



BeyondTrust

Privilege Management for Unix & Linux Policy Language Guide

Table of Contents

Privilege Management for Unix & Linux Language Guide	14
Privilege Management for Unix & Linux Overview	15
Privilege Management for Unix & Linux Components	15
Privilege Management for Unix & Linux Task Processing	16
Create Policy Files in Privilege Management for Unix & Linux	18
Default Policy	19
Role Based Policy	21
Database Schema	21
User Groups	21
Host Groups	22
Command Groups	22
Roles	24
Role "Auth" Attribute	26
Role Based Policy, Change Management Events	29
Role Based Policy Entitlement Reports	30
Policy File Format	38
Variable Scope	38
Syntax Checking	38
Environment Variable Processing Considerations	39
Security Policy Scripting Language Definition	41
Variables and Data Types	41
Variables	41
Variable Scope	41
Variable Data Types	42
Constants	45
Operators	46
Arithmetic Operators	47
Logical Operators	51
Relational Operators	53
Special Operators	56
Expressions	59

Program Statements	60
Executable Program Statements	60
Non-Executable Program Statements	74
Functions and Procedures	75
function Statement	75
procedure Statement	76
Other Programming Considerations	77
Boolean True and False Variables	77
Format Commands	77
Regular Expression Patterns	80
Wildcard Search Characters	82
Special Characters	82
Privilege Management for Unix & Linux Variables	84
Task Information Variables	85
argc	90
argv	90
bkgd	91
browserhost	92
browserip	92
clienthost	93
command	93
cwd	94
env	95
execute_via_su	96
group	97
groups	98
host	98
localmode	99
logcksum	100
logpid	101
logservers	102
mastertimelimit	102
mastertimeout	103

nice	104
noexec	105
optimizedrunmode	106
pblocaldnoglob	107
pbrisklevel	107
pidmessage	108
requestuser	109
rlimit_as	109
rlimit_core	110
rlimit_cpu	111
rlimit_data	112
rlimit_fsize	113
rlimit_locks	114
rlimit_memlock	115
rlimit_nofile	116
rlimit_nproc	117
rlimit_rss	118
rlimit_stack	119
false	120
hour	120
i18n_date	121
i18n_day	121
i18n_dayname	122
i18n_hour	122
i18n_minute	122
i18n_month	123
selinux	123
runchroot	124
runcksum	125
runcksumlist	126
runconfirmmessage	126
runconfirmpasswdservice	127
runconfirmuser	128

runeffectivegroup	129
runeffectiveuser	130
runenablerlimits	130
runmd5sum	131
runmd5sumlist	132
runenvironmentfile	133
runpamsessionsservice	134
runpamsetcred	135
runpid	135
runptyflags	136
runsecurecommand	136
runtimelimit	137
runtimeout	138
runutmpuser	139
shellallowedcommands	140
shellcheckbuiltins	140
shellcheckredirections	141
shellforbiddencommands	142
shellloginincludefiles	143
shellreadonly	143
shellrestricted	144
solarisproject	145
submithost	146
submithostip	146
submitpid	146
taskpid	147
tasktynname	147
timezone	148
ttynname	148
umask	149
user	149
Command Line Parsing Variables	151
optarg	151

opterr	152
optind	153
optopt	153
optreset	154
optstrictparameters	154
Logging Variables	156
event	157
eventlog	157
exitdate	158
exitstatus	158
exittime	159
forbidkeyaction	159
forbidkeypatterns	160
i18n_exitdate	161
i18n_exittime	161
iolog	162
logmaximumfailures	162
lognopassword	163
logomit	164
logstderr	165
logstderrlimit	165
logstdin	166
logstdinlimit	167
logstdout	167
logstdoutlimit	168
passwordloggingprompts	169
System Variables	170
date	172
day	172
dayname	172
false	173
hour	173
i18n_date	174

i18n_day	174
i18n_dayname	175
i18n_hour	175
i18n_minute	175
i18n_month	176
i18n_time	176
i18n_year	177
lineinfile	177
linenum	178
lognoreconnect	178
masterhost	179
minute	179
month	180
noreconnect	180
outputredirect	181
pbclientcertificateissuer	181
pbclientcertificatesubject	182
pbclientkerberosuser	182
pbclientmode	183
pbclientname	183
pblogdreconnection	184
pbrunreconnection	185
pbversion	185
pid	186
ptyflags	186
status	186
submittimeout	187
subprocuser	187
time	188
true	188
uniqueid	189
year	189
Host Identification Variables	191

masterlocale	193
runlocale	194
submitlocale	194
pbguidmachine	194
pbguidnodename	195
pbguidrelease	195
pbguidsysname	195
pbguidversion	196
pbkshmachine	196
pbkshnodename	196
pbkshrelease	197
pbkshsysname	197
pbkshversion	197
pblocaldcertificateissuer	198
pblocaldcertificatesubject	198
pblocaldmachine	199
pblocaldnodename	199
pblocaldrelease	199
pblocaldsysname	199
pblocaldversion	200
pblogdcertificateissuer	200
pblogdcertificatesubject	200
pblogdmachine	201
pblogdnodename	201
pblogdrelease	201
pblogdsysname	202
pblogdversion	202
pbmasterdcertificateissuer	202
pbmasterdcertificatesubject	203
pbmasterdmachine	203
pbmasterdnodename	203
pbmasterdrelease	204
pbmasterdsysname	204

pbmasterdversion	204
pbrunmachine	204
pbrunnodename	205
pbrunrelease	205
pbrunsysname	205
pbrunversion	205
pbshmachine	206
pbshnodename	206
pbshrelease	207
pbshsysname	207
pbshversion	207
X11 Session Capture Variables	208
xwincookie	208
xwinproto	208
xwindisplay	208
xwinforward	209
xwinreconnect	209
Built-in Functions and Procedures	211
Advanced Control and Audit	212
Important Considerations	212
aca	214
Enablesessionhistory	217
Date and Time Functions	219
datecmp	219
strftime	220
timebetween	220
File and Path Functions	222
access	222
basename	223
dirname	223
logmktemp	224
mktemp	225
stat	226

Format and Conversion Functions	228
atoi	228
sprintf	228
Input/Output Functions and Procedures	230
fprintf	230
input	231
inputnoecho	232
print	232
printf	233
printnl	234
printvars	235
readfile	236
syslog	236
LDAP Functions	238
ldap_attributes	239
ldap_bind	239
ldap_dn2ufn	240
ldap_entry_count	241
ldap_explodedn	242
ldap_firstentry	243
ldap_getdn	243
ldap_getvalues	244
ldap_init	245
ldap_nextentry	245
ldap_open	246
ldap_search	247
ldap_unbind	248
List Functions	249
append	249
insert	250
join	251
length	252
range	252

replace	253
search	254
split	255
Miscellaneous Functions and Procedures	257
egrep	258
fgrep	258
glob	259
grep	260
iologcloseaction	261
iologcloseactionrunhost	262
ipaddress	263
isset	263
policytimeout	264
quote	266
remotesystem	266
runtimewarn	268
runtimewarnlog	269
system	269
unset	270
NIS Functions	272
innetgroup	272
inusernetgroup	273
Policy Environment Functions and Procedures	274
getlistsetting	274
getnumericsetting	275
getstringsetting	275
getyesnosetting	276
policygetenv	277
policysetenv	278
policyunsetenv	278
String Functions	280
charlen	280
gsub	281

length	282
pad	282
sub	283
substr	284
tolower	285
toupper	285
Task Control Procedures	287
setkeystrokeaction	287
Task Environment Functions and Procedures	289
keystrokeactionprofile	289
getenv	290
keepenv	291
setenv	291
unsetenv	292
Command Line Parsing Functions	294
getopt	294
getopt_long	295
getopt_long_only	296
User and Password Functions	299
getfullname	299
getgroup	300
getgrouppasswd	301
getgroups	301
gethome	302
getshell	303
getstringpasswd	303
getuid	304
getuserpasswd	305
ingroup	306
submitconfirmuser	307
PAM Policy Functions	309
getuserpasswdpam	309
submitconfirmuserpam	310

Persistent Variable Functions and Procedures	313
listpersistentvars	313
setpersistentvar	314
getpersistentvarint	315
getpersistentvarstring	315
getpersistentvarlist	316
delpersistentvar	317
Glossary	318

Privilege Management for Unix & Linux Language Guide

This guide provides detailed information regarding the security policy file programming language for the BeyondTrust Privilege Management for Unix & Linux software. This language is used to create security policy files that are used by Privilege Management for Unix & Linux to:

- Control the tasks a user or group of users may perform
- Control the systems from which a task may be submitted
- Control the systems from which a task may be run
- Determine when a specific task may be run (day and time)
- Determine where a task may be run from
- Determine if secondary security checks, such as passwords or checksums, are required to run a task
- Determine if one or more supplemental security programs are run before a task is started



Note: This guide assumes that the user has a basic understanding of Unix or Linux system administration and some experience with a scripting or other computer language. It is recommended that you have experience in these areas before you attempt to create or modify security policy files



Note: Privilege Management for Unix & Linux Basic refers to the product formerly known as PowerBroker for Sudo. Privilege Management for Unix & Linux refers to the product formerly known as PowerBroker for Unix and Linux.



Note: Specific font and line-spacing conventions are used to ensure readability and to highlight important information, such as commands, syntax, and examples.

Sample Policy Files

When you install Privilege Management for Unix & Linux, you can choose to copy sample Privilege Management for Unix & Linux policy files to the installation host. These sample policy files include detailed explanations of what they do. You can use these files to learn how policy files are typically written for various scenarios. The directory that these sample files are copied to is determined by the GUI library directory option that you specify during installation. By default, this directory is `/usr/local/lib/pbuilder`. A `readme_samples` text file in that directory includes a brief description of each sample file.

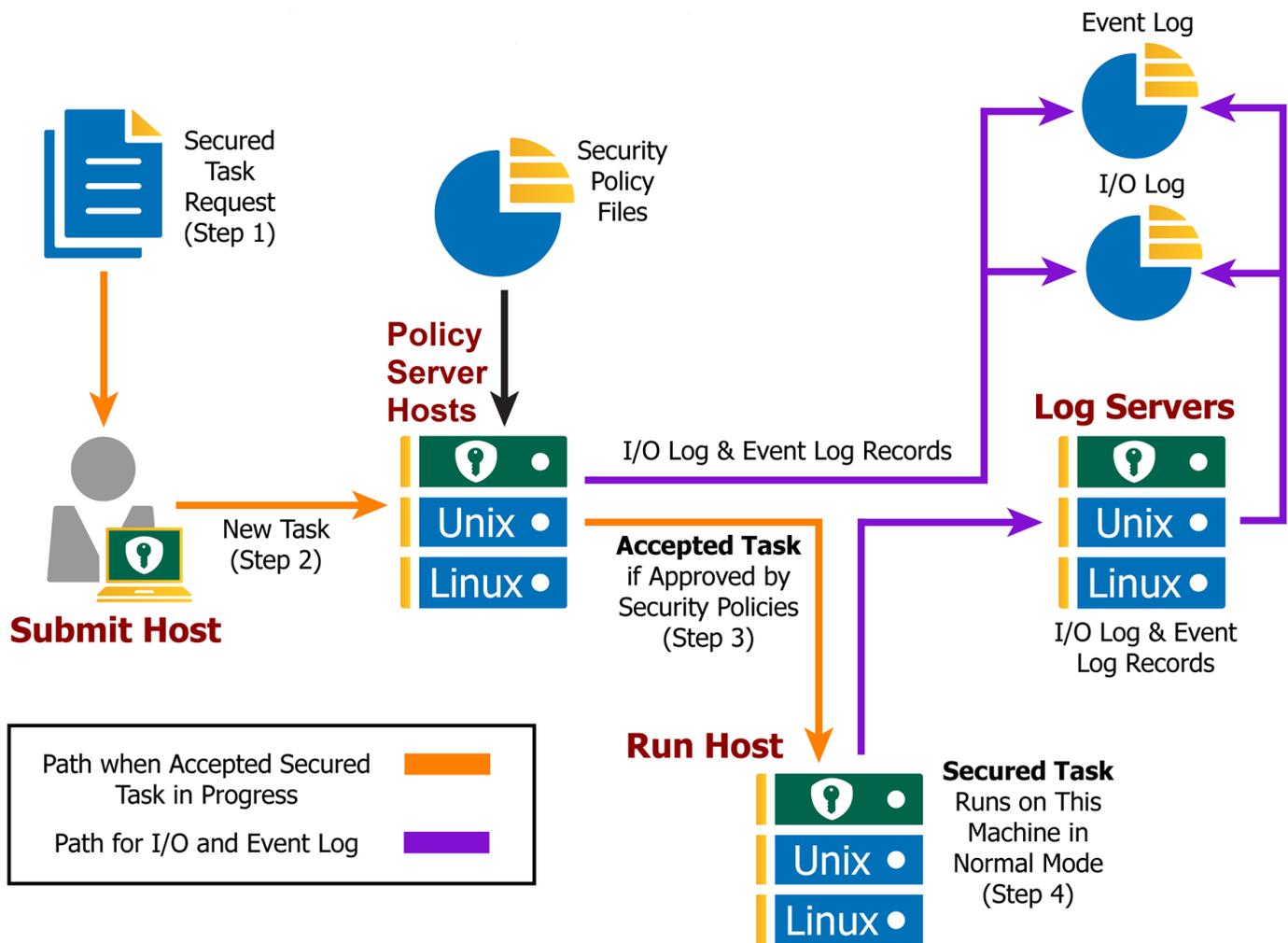
Privilege Management for Unix & Linux Overview

To write effective Privilege Management for Unix & Linux security policy files, it is helpful to understand how Privilege Management for Unix & Linux works. A typical Privilege Management for Unix & Linux configuration consists of the following primary components: **pbrun**, **pbrunmasterd**, **pblockd**, and **pblogd**. Each of these components is described below. It is possible to install all of these components on a single machine or distribute them among different machines. For optimal security, the Policy Server host and log hosts should be separate machines that are isolated from normal activity.

Privilege Management for Unix & Linux Components

As shown in the figure below, the machine from which a task is submitted is referred to as the submit host. The machine on which security policy file processing takes place is referred to as the Policy Server host. The machine on which a task actually executes is referred to as the run host. The machine on which event log records and I/O logs are written is referred to as the log host. (Although the use of **pblogd** is highly recommended, it is an optional component.)

How Privilege Management for Unix & Linux Works



Privilege Management for Unix & Linux Task Processing

In the context of Privilege Management for Unix & Linux, there are two types of task requests: secured and unsecured.

Secured task requests must undergo security validation processing before they can be run. Privilege Management for Unix & Linux must process these tasks.

Unsecured tasks do not undergo security validation processing. These tasks do not represent a potential threat to the system and so do not fall under a company's security policy implementation. The operating system handles unsecured tasks. Privilege Management for Unix & Linux is not involved in the processing of unsecured tasks.

Secured Task Submission to SSH-Managed Devices - pbssh

Secured tasks can also be submitted through **pbssh**. **pbssh** is the Privilege Management component used to access SSH-managed devices where Privilege Management is not installed (routers, firewalls, Windows devices, or Unix/Linux devices where Privilege Management is not installed). **pbssh** connects to the target device using the SSH configuration.

Task Submission - pbrun

All secured tasks must be submitted through **pbrun**, the Privilege Management for Unix & Linux component that receives task requests. A separate **pbrun** process starts for each submitted secured task request. Any task that needs to undergo Privilege Management for Unix & Linux security processing (that is, a secured task) must be submitted through **pbrun**. A company's security policy implementation may be compromised if the use of **pbrun** for secured tasks is not enforced.



Note: ***pbrun** must be installed on any machine from which a user can submit a secured task request.*

Security Policy File Processing - pbmasterd

pbmasterd is responsible for applying the security rules as defined in the Privilege Management for Unix & Linux security policy files that make up a company's network security policy. In other words, it is **pbmasterd** that performs security verification processing to determine if a request is accepted (that is, allowed to execute) or rejected (that is, not allowed to execute), based on the logic in the Privilege Management for Unix & Linux security policy files. If a request is rejected, then the result is logged and processing terminates. If a request is accepted, then it is immediately passed to **pblocald** for execution.

If the **pblogd** component (below) is not used, then **pbmasterd** waits for the **pblocald** process to complete. If **pblogd** is used, then **pbmasterd** terminates after the request is passed to **pblocald**. A separate **pbmasterd** process starts for each secured task request that is submitted.



Note: *During security verification processing, the first "accept" or "reject" condition that is encountered causes security policy file processing to terminate immediately. No further security verification processing is performed.*

Task Execution - pblocald

pblocald is normally responsible for executing task requests that have passed security verification processing and have been accepted by **pbmasterd** on the run host (when the run host is a different host than the submit host). After a task request is accepted, it is immediately passed from **pbmasterd** to **pblocald**. By default, **pblocald** executes the task request as the user that is specified in the policy variable **runuser**. This is typically a privileged user such as **root**, a database administrator, or a web server administrator. All task input and output information is piped back to **pbrun**. In addition, **pblocald** logs pertinent task information to the Privilege Management for Unix & Linux Event Log using **pbmasterd** or **pblogd**. This depends on how Privilege Management for Unix & Linux has been deployed. The run host can also record task keystroke information to a Privilege Management for Unix & Linux I/O log and again through **pbmasterd** or **pblogd**. Again, this depends on how Privilege Management for Unix & Linux has been deployed.

Task Execution - pbrun

When the run host and submit host are on the same machine, **pbrun** can directly execute a secured task. This optimizes out the extra network connections to **pblocald**.

Logging - pblogd

pblogd is responsible for writing event and I/O log records. **pblogd** is an optional Privilege Management for Unix & Linux component. If **pblogd** is not installed, then **pbmasterd** writes log records directly to the appropriate log files rather than passing them off to **pblogd**. In addition, without **pblogd** installed, **pbmasterd** must wait for the **pblocald** process to complete. If the **pblogd** component is used, then **pbmasterd** normally terminates when task execution starts and **pblocald** sends its log records directly to **pblogd**.

Using **pblogd** optimizes Privilege Management for Unix & Linux processing by:

- Centralizing the writing of log records in a single, dedicated component
- Eliminating the need for the **pbmasterd** process to wait for task execution to complete

Create Policy Files in Privilege Management for Unix & Linux

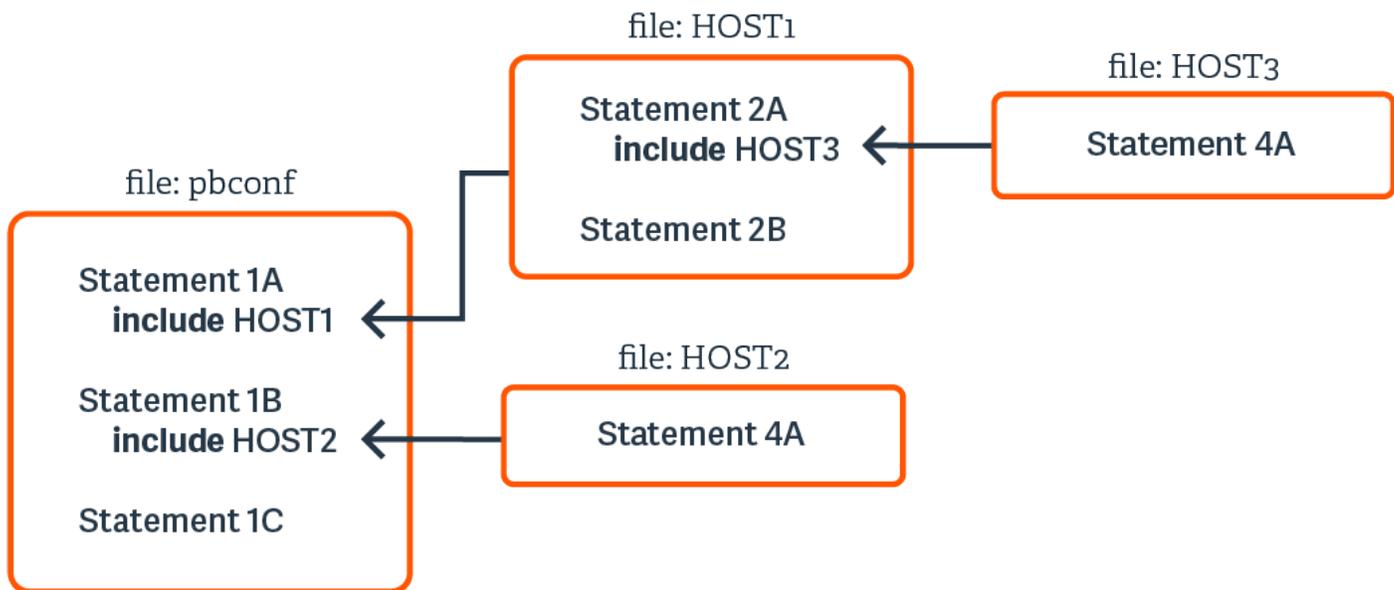
A security policy file is a collection of instructions that define the system security rules that Privilege Management for Unix & Linux applies during task verification processing. These instructions are written using Privilege Management for Unix & Linux security policy scripting language.

The default name of the primary Privilege Management for Unix & Linux security policy file is **pb.conf**. This file is analogous to the **main()** function in a C program. It is possible to add security policy scripting language statements directly to **pb.conf** or to use security policy subfiles. Security policy subfiles are separate, individual security policy files invoked at runtime using the **include** statement (using the syntax **include "subfilename"**).



Note: It is strongly recommended that you use security policy subfiles.

Conceptually, the **include** statement can be thought of as a placeholder.



At run time, Privilege Management for Unix & Linux replaces **include** statements with the actual contents of the specified include file. This process occurs in computer memory and does not alter the physical files in any way.

The use of security policy subfiles enables you to organize a site's security policy implementation in a modular fashion. Using this method, each security policy subfile can focus on a specific area of security policy implementation. This compartmentalizes security policy implementation, making it much easier to maintain and enhance over time.

A common way to organize security profile files is by type of user and system access requirements.

root should own the security policy files and their permissions should be set to **400** or **600**. Place the files in the same directory (**/opt/pbul/policies** is recommended) for convenience. The **/opt/pbul/policies** directory is the default location. A different directory can be specified with the **policydir** setting in the **pb.settings** file. To insure security policy file integrity, Privilege Management for Unix & Linux will not process a security policy file if users other than **root** has security permissions that allow them to modify or delete the file. In other words, only **root** should have read/write permissions for these files, and the directories in which these files are stored should have security permissions that prevent users other than **root** from reading, modifying, or deleting them.

Security policy files are usually created with a standard text editor. They are saved as plain text files. By default, Privilege Management for Unix & Linux uses a `.conf` file name suffix for security policy files, but this is not a requirement.

When naming security policy files, any file suffix may be used, or the suffix may be omitted. Starting with v9.0, a new Role Based Policy mechanism has been implemented that allows administrators to maintain their policy in a database with an option 'change management' functionality.

Default Policy

Starting with version 8.0, a default policy will be installed by default if an existing policy does not exist. The files `pbul_policy.conf` and `pbul_functions.conf` will be created in a `/opt/pbul/policies` directory (from v9.4.3+ and in `/etc/pb` prior to v9.4.3) by default. `pbul_policy.conf` will then be included in the main policy (by default `/opt/pbul/policies/pb.conf` from v9.4.3+ and `/etc/pb.conf` prior to v9.4.3).

This default policy contains the following roles:

Helpdesk Role

Enabled by default, when invoking `pbrun helpdesk` it allows any user in `HelpdeskUsers` (default `root`) to initiate a Helpdesk Menu as root on any host in `HelpdeskHosts` (default `submithost` only). Helpdesk Menu of actions contains:

- List of processes (`ps -ef`)
- Check if a machine is up (`ping <host>`)
- List current users on this host (`who -H`)
- Display Host's IP settings (`ifconfig -a`)

PBTest

Enabled by default, for all users on all hosts, `pbrun pbtest` allows checking connectivity and policy.

Controlled Shells

Enabled by default, allows users in `ControlledShellUsers` (by default the `submituser`), for runhosts in `ControlledShellHosts` (by default only `submithost`), to enable iologging for `pbksh/pbsh`. iologs are created by default in `"/tmp/pb.<user>.<runhost>.<YYYY-MM-DD>.[pbksh|pbsh].XXXXXX"`. This role has a list of commands (empty by default) to elevate privileges for, as well as a list of commands (empty by default) to reject.

Admin role

Enabled by default, allows users in `AdminUsers` (by default `root`) to run any command on runhosts in `AdminHosts` (by default only `submithost`).

Demo role

Disabled by default, allows users in `DemoUsers` (default all users) to run commands in `DemoCommands` (default `id` and `whoami`) as `root` on any host in `DemoHosts` (default all hosts).

The policy ends by allowing all users to run any command as themselves without any privilege escalation.

This policy is meant to be used as a starting point for your own policy. You can enable or disable any of the roles listed above by simply setting the corresponding `"Enable<rolename>Role"` to `true` or `false`. Or you can completely delete the policy and use your own. If you choose to continue with the default policy as a starting point, you can add more users, hosts and commands to the various lists used for each role, for example you can take the

ControlledShellRole further by adding users to **ControlledShellUsers**, and hosts to **ControlledShellHosts**, and commands to **ControlledShellRejectedCmds** and **ControlledShellPrivilegedCmds**.

Splunk role

Disabled by default. If enabled, only when **pbrun** is invoked, enables iologging (creating iologs in **/pbiologs**), sets default ACA rule, enables aca session history and sets iologcloseaction to a script sending records to Splunk.

Role Based Policy

Role Based Policy has been implemented to simplify the definition of policy for administrators. Policies are kept within structured records in a database, simplifying maintenance, decreasing system load, increasing throughput, and providing a comprehensive REST API to integrate policy management with existing customer systems and procedures, including simplified bulk import/export of data. Once the customers' data is held within the Role Based Policy database it is much easier to provide management information, such as user entitlement reports. The policy data is grouped into users, hosts, commands, time/dates and roles detailed in the schema below.

Database Schema

User Groups

User Groups define groups of users and/or wildcard patterns that match usernames:

```
CREATE TABLE usergrp
(id INTEGER PRIMARY,
name TEXT UNIQUE,
description TEXT,
disabled INTEGER CHECK(disabled BETWEEN 0 AND 1), -- 0=enabled, 1=disabled
type CHAR(1) CHECK (type IN ('I','E')), -- I=internal, E=external
extinfo TEXT -- external lookup info
);

CREATE TABLE userlist (
id INTEGER REFERENCES usergrp(id),
user TEXT, -- "glob" wildcard
PRIMARY KEY(id,user)
);
```

Each User Group has multiple User List entries that specify names and/or wildcards that will match both submit and run user names when matched by the Role.

USER GROUP

1. User Group ID (key, unique)
2. User Group Name (unique)
3. User Group Description
4. Disabled
5. Group Type (Internal/external)
6. External Group connection info (encoded - json ?)

MEMBERSHIP LIST

1. User Group ID (foreign key)
 2. Member String (glob/regex)
- } Composite Key

Host Groups

Host Groups define groups of hosts and/or wildcard patterns that match hostnames:

```
CREATE TABLE hostgrp (
  id INTEGER PRIMARY,
  name TEXT UNIQUE,
  description TEXT,
  disabled INTEGER CHECK(disabled BETWEEN 0 AND 1), -- 0=enabled, 1=disabled
  type CHAR(1) CHECK (type IN ('I','E')), -- I=Internal, E=external
  extinfo TEXT -- external lookup info
);
CREATE TABLE hostlist (
  id INTEGER REFERENCES hostgrp(id),
  host TEXT, -- "glob" wildcard
  PRIMARY KEY(id,host)
);
```

Each Host Group has multiple Host List entries that specify names and/or wildcards that will match both submit and run host names when matched by the Role.

HOST GROUP

1. Host Group ID (key, unique)
2. Host Group Name (unique)
3. Host Group Description
4. Disabled
5. Group Type (internal/external)
6. External Group connection info (encoded - json ?)

HOST LIST

- | | | |
|---|---|---------------|
| <ol style="list-style-type: none"> 1. Host Group ID (foreign key) 2. Host String (glob/regex) | } | Composite Key |
|---|---|---------------|

Command Groups

Command Groups define groups of commands and/or wildcard patterns that match commands:

```
CREATE TABLE cmdgrp (
  id INTEGER PRIMARY,
  name TEXT UNIQUE,
  description TEXT,
  disabled INTEGER CHECK(disabled BETWEEN 0 AND 1)-- 0=enabled, 1=disabled
);
CREATE TABLE cmdlist (
  id INTEGER REFERENCES cmdgrp(id),
```

```
cmd TEXT, -- "glob" wildcard
rewrite TEXT, -- new command (see below)
PRIMARY KEY (id,cmd)
);
```

Each Command Group has multiple Command List entries that specify commands and/or wildcards that will match the submitted command name when matched by the Role, and a rewrite column to rewrite the command that will executed. The rewrite is in a similar format to Bourne/Bash shell arguments, ie \$0, \$1, etc, \$* and \$#. Rewrite will use the original command to substitute arguments into the new rewritten command.

COMMAND GROUP

1. Command Group ID (key, unique)
2. Command Group Name (unique)
3. Command Group Description
4. Disabled

COMMAND LIST

1. Command Group ID (foreign key)
 2. Command String (glob/regex)
 3. Command re-write
- } Composite Key

Time/Date Groups

Time/Date Groups define groups of times/dates and/or wildcard patterns that match times/dates:

```
CREATE TABLE tmdategrp (
id INTEGER PRIMARY,
name TEXT UNIQUE,
description TEXT,
disabled INTEGER CHECK(disabled BETWEEN 0 AND 1)-- 0=enabled, 1=disabled
);

CREATE TABLE tmdatelist (
id INTEGER REFERENCES tmdategrp(id),
tmdate TEXT, -- json format - see below
PRIMARY KEY (id,tmdate)
);
```

Each Time/Date Group has multiple Time/Date List entries that specify times/dates and/or wildcards that will match the submitted command name when matched by the Role, and a rewrite column to rewrite the command that will executed. Each individual time/date is specified in JSON format, and can be one of two different formats: From/To specific date range - both from and to are specified in epoch seconds:

```
'{ "range" : { "from" : 1415851283, "to": 1415887283 } }'
```

Day of the Week - each day is specified as an array of hours.

Each hour is a number representing 15 minute intervals defined as a binary mask:

```
1 1 1 1
  ^ 0 to 14 minutes of the hour
  ^-- 15 to 29 minutes of the hour
  ^---- 30 to 44 minutes of the hour
  ^----- 45 to 59 minutes of the hour
Therefore the values range from 0 to 15:
{'
  "mon" : [0,0,0,0,0,0,0,15,15,15,15,15,15,15,15,15,15,3,0,0,0,0,0,0],
  "tue" : [0,0,0,0,0,0,0,15,15,15,15,15,15,15,15,15,15,3,0,0,0,0,0,0],
  "wed" : [0,0,0,0,0,0,0,15,15,15,15,15,15,15,15,15,15,3,0,0,0,0,0,0],
  "thu" : [0,0,0,0,0,0,0,15,15,15,15,15,15,15,15,15,15,3,0,0,0,0,0,0],
  "fri" : [0,0,0,0,0,0,0,15,15,15,15,15,15,15,15,15,15,3,0,0,0,0,0,0],
  "sat" : [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
  "sun" : [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
}'
```

TIME/DATE GROUP

1. Time/Date Group ID (key, unique)
2. Time/Date Group Name (unique)
3. Time/Date Group Description
4. Disabled

TIME/DATE LIST

1. Time/Date Group ID (foreign key)
 2. Time/Date/Day (json encoded)
- } Composite Key

Roles

Roles are the entities that tie all the other entities together to define a Role.

```
CREATE TABLE role
(id INTEGER PRIMARY,
name TEXT UNIQUE,
rorder INTEGER, -- rule order for matching
description TEXT,
disabled INTEGER CHECK(disabled BETWEEN 0 AND 1), -- 0=enabled, 1=disabled
risk INTEGER CHECK(risk >= 0),
action CHAR(1) CHECK (action IN ('A', 'R')), -- A=Accept, R=Reject
iolog TEXT, -- iolog template
script TEXT -- pbparse script
tag TEXT DEFAULT NULL -- Arbitrary tag that will allow grouping of roles
```

```
comment TEXT DEFAULT NULL      -- Arbitrary comment field that can contain anything
message TEXT DEFAULT NULL      -- Accept/reject message (templated)
variables TEXT DEFAULT NULL    -- Contains JSON formatted Policy Script variables to set (templated)
varmatch TEXT DEFAULT NULL     -- Contains JSON formatted Policy Script variables to match
auth TEXT DEFAULT NULL         -- Contains JSON formatted array of authentication methods (templated)
rpt INTEGER DEFAULT 1          -- 1=on, 0=off, include Role in Entitlement Report
);

CREATE TABLE roleusers (
  id INTEGER REFERENCES role(id),
  users INTEGER REFERENCES usergrp(id),
  type CHAR(1) CHECK (type IN ('S','R')), -- S=Submit, R=Run User
  PRIMARY KEY (id,users,type)
);

CREATE TABLE rolehosts (
  id INTEGER REFERENCES role(id),
  hosts INTEGER REFERENCES hostgrp(id),
  type CHAR(1) CHECK (type IN ('S','R')), -- S=Submit, R=Run User
  PRIMARY KEY (id,hosts,type)
);

CREATE TABLE rolecmds (
  id INTEGER REFERENCES role(id),
  cmds INTEGER REFERENCES cmdgrp(id),
  PRIMARY KEY (id,cmds)
);

CREATE TABLE roletmdates (
  id INTEGER REFERENCES role(id),
  tmdates INTEGER REFERENCES tmdategrp(id),
  PRIMARY KEY (id,tmdates)
);
```

Each Role has multiple Users, Hosts, Commands and Time/Dates. When the Policy Engine matches against Roles, complete records are selected from the database as fully populated Roles, sorted by the Role attribute "rorder". Once the first record has been matched, the attributes of the Role are applied to the session, and the Policy Engine will accept or reject the session. The iolog template is the normal script format log file, for example `/var/log/io_log.XXXXXXX`. The script is a full Privilege Management for Unix & Linux script that will be called if the Role has been accepted. This script can carry out extra processing to authorize the session (and can therefore override the accept/reject status with an implicit command), and can carry out extended environment configuration as would normal Privilege Management for Unix & Linux script.

ROLE

1. Role Group ID (key, unique)
2. Role Name (unique)
3. Role Order
4. Role Description
5. Disabled
6. Risk
7. Accept/Reject
8. I/O Log Template
9. Script
10. Tg
11. Comment
12. Message Template
13. Policy Script Variables
14. Varmatch
15. Authentication Methods
16. Include in Entitlement Report

SUBMIT/RUN USER LIST

1. Role ID (foreign key)
 2. User Group ID (foreign key)
 3. Type (submit/run)
- } Composite Key

COMMAND LIST

1. Role ID (foreign key)
 2. Command Group ID (foreign key)
- } Composite Key

SUBMIT/RUN HOST LIST

1. Role ID (foreign key)
 2. Host Group ID (foreign key)
 3. Type (submit/run)
- } Composite Key

TIME/DATE LIST

1. Role ID (foreign key)
 2. Time/Date Group ID (foreign key)
- } Composite Key

Role "Auth" Attribute

A new column holding a JSON formatted configuration provide the flexibility of the multiple authentication methods that script policy currently employ. The applicable functions will then be called by Role Based Policy authorization functions in a similar way as the script based policy.

A new database column, formatted in JSON provides extra authentication options. The column is a JSON array of methods that will be called in order, and will REJECT when the first one fails. Each array element is a JSON object with a "method" and attributes:

```
{ "method" : "getstringpasswd", "passwd" : <string>, "prompt": "<string>", "message": "<string>", "rejectMessage": "<string>", "tries": <num> }
```

passwd	base64 encoded SHA256 password to match
prompt	the prompt string
message	message to display if the authentication fails
rejectMessage	the "reject" message that is logged against the event
tries	the number of password attempts

```
{ "method" : "getuserpasswd", "user": <string>, "fname" : <string>, "prompt": "<string>", "message": "<string>", "rejectMessage": "<string>", "tries": <num>, "period" : <num> }
```

user	username to check
fname	the unique filename used to cache the password authentication
prompt	the prompt string
message	message to display if the authentication fails
rejectMessage	the "reject" message that is logged against the event
tries	the number of password attempts
period	the maximum duration before the user has to re-authenticate

```
{ "method" : "getuserpasswdpam", "user": <string>, "service" : <string>, "fname" : <string>, "prompt": "<string>", "message": "<string>", "rejectMessage": "<string>", "tries": <num>, "period" : <num> }
```

user	username to check
service	the PAM service string
fname	the unique filename used to cache the password authentication
prompt	the prompt string
message	message to display if the authentication fails

rejectMessage	the "reject" message that is logged against the event
tries	the number of password attempts
period	the maximum duration before the user has to re-authenticate

```
{ "method" : "submitconfirmuser", "user":<string>, "fname" : <string>, "prompt": "<string>",
  message": "<string>", "rejectMessage": "<string>", "tries": <num>, "period" : <num> }
```

user	username to check
fname	the unique filename used to cache the password authentication
prompt	the prompt string
message	message to display if the authentication fails
rejectMessage	the "reject" message that is logged against the event
tries	the number of password attempts
period	the maximum duration before the user has to re-authenticate

```
{ "method" : "submitconfirmuserpam", "user":<string>, "service" : <string>, "fname" : <string>,
  "prompt": "<string>", message": "<string>", "rejectMessage": "<string>", "tries": <num>, "period" :
  <num> }
```

user	username to check
service	the PAM service string
fname	the unique filename used to cache the password authentication
prompt	the prompt string
message	message to display if the authentication fails
rejectMessage	the "reject" message that is logged against the event
tries	the number of password attempts
period	the maximum duration before the user has to re-authenticate

There are also three other variables (namely `runconfirmuser`, `runconfirmmessage`, `runconfirmpasswdservice`) that affect re-authentication. However, because these are Policy Script variables as opposed to functions, these are implemented in a similar way. In this respect, these variables should be set in the "variables" column, and are templated in a similar manner, for example:

```
{ "runconfirmuser" : "%user%" }
```

Matching Privilege Management for Unix & Linux Variables for a Role

A new JSON formatted column has been introduced that will allow the matching of roles based upon variables submitted by the client, for example `pbclientmode`. Matched values are wildcarded using normal `glob(3)` rules. The format of the object is similar to:

```
{ "varmatch" : { "pbclientmode" : "pbrun", "year" : "201[678]" }}
```

Role Based Policy, Change Management Events

There are two different approaches to maintaining the Role Based Policy database. The first, simple method is to access the tables using `pbdbutil` at the command line. Each change will be individual, and instantaneous, and will be "live" immediately. Although for smaller organizations this will be adequate, larger organizations will be a more controlled procedural access method.

Role Based Policy database "change transactions" can be enabled using the `pb.setting rbptransactions`. Once enabled, before changes can be made, the administrator must begin a change transaction, specifying a reason why the change is being made. This is logged and the whole Role Based Policy database is then locked for update - only that administrator can continue to make changes. These changes will NOT be mirrored in the "live" authorization process and can continue to be made by that administrator alone, and when completed can be "committed" or "rolled back". Once the changes are committed they are all applied to the database as one update, and a changemanagement event is generated. If the changes are rolled back, they will be discarded and nothing will change.

If, for whatever reason, a change transaction is begun, and that administrator leaves it open and fails to close the transaction, any other administrator with access can force the rollback of the changes. Once again, this requires a reason specifying, and will log a change management event. The change transactions are necessary once the GUI policy updates are implemented to force database integrity. See the section below for Change Transaction Command Line options.

To enable the logging of change management events each client needs the `pb.setting changemanagementevents` **yes** and log servers will need to defined the `eventdb <path>` and need the REST `pbrest` service running.

The following settings are used and need to be set when Role Based Policy and Change management is implemented and used:

`policydb <path>`

- The path to the Role Based Policy Database.
- There is no default for this setting.

`pbresturi <string>`

- The partial REST url string between the hostname and /REST
- There is no default for this setting.

`pbrestport <port#>`

- The REST port
- Default value is the base port + 6

rolebasedpolicy <yes/no>

- Enabled/Disable Role Based Policy checking
- The default is **no**

eventdb <path>

- The path to the Change Management Event Database
- There is no default for this setting.

rbptransactions <yes/no>

- Enable the use of Role Based Policy Transactions to ensure integrity
- The default is **no**

changemanagementevents <yes/no>

- Enable/Disable the logging of Change Management Events when maintaining databases
- The default is **no**

pbresttimeskew <num>

- The maximum time in seconds that hosts are mis-matched by (it is recommended that the customer uses a time synchronization service)
- The default is **60 seconds**

Role Based Policy Entitlement Reports

Privilege Management for Unix & Linux v10.1.0 introduced Role Based Policy Entitlement Reports. These reports are available to the user from the **pbrun** command using **-e**, or to the administrator as an overall report using **pbdbutil --rbp -R**. They provide a comprehensive report on what users can access commands on which hosts, and when they are allowed to run them.

pbdbutil: Role Based Policy Options

The pbdbutil Role Based Policy options introduced in Privilege Management for Unix & Linux v10.1.0 are described below.

```
pbdbutil --rbp [<options>] [ <file> <file> ...]
-R { json param } Report user entitlements from the database
  -R Add option to display commands
    -R Add option to display time/date restrictions
      -R Add option to display additional role options
-E { json param } List user entitlements data from the database
  where { json param } is one or more of:
  "submituser" : "user1" Specify submit user or wildcard
  "submithost" : "host1" Specify submit host or wildcard
  "runuser"    : "user1" Specify run user or wildcard
  "runhost"    : "host1" Specify run host or wildcard
  "command"   : "command" Specify command or wildcard
```

pbrun Options

Privilege Management for Unix & Linux v10.1.0 introduced the following options that are available only when Role Based Policy is enabled:

<code>pbrun -e</code>	Will return the entitlement report for the current user at level 1
<code>pbrun -e 1</code>	Will return the entitlement report for the current user at level 1
<code>pbrun -e 4</code>	Will return the entitlement report for the current user at level 4
<code>pbrun --entitlement=4</code>	Will return the entitlement report for the current user at level 4

Examples of Entitlement reports:

```

Level 1 report
=====
Privilege Management for Unix & Linux Role Based Policy Entitlement Report - Level 1
-----
Date/Time: 2018-06-18 09:07:23
User: root
Belongs to the following Roles:
Admin
=====
Role Order:      1
Name:            Admin
Description:     Super users and admins
Action:         allowed
Tag:
Membership:     Admins
Submit Host(s): Any PBUL Host
Run Host(s):    Any PBUL Host
Commands may be executed as user(s): root,admin,user*
Please use the '-u' flag to select user at run time.
eg: pbrun -u runuser command [arguments]
User may request the following commands using pbrun:
/bin/find *,/usr/bin/ls,/bin/ls,/bin/cat *,/bin/ls *,/usr/bin/ls *,/usr/bin/rm *,
/usr/bin/cat *,/usr/bin/find *,/sbin/shutdown *,/bin/more *,/bin/id,/usr/bin/more *,
/usr/bin/mount *,/bin/ln *,/bin/mount *,/bin/rm *,/usr/sbin/shutdown *,
/usr/bin/ln *,/usr/bin/id,/sbin/ifconfig *,/usr/sbin/ifconfig *
=====
    
```

Privilege Management for Unix & Linux Role Based Policy Entitlement Report - Level 2

Date/Time: 2018-06-18 09:07:28

User: root

Belongs to the following Roles:

Admin

```

=====
Role Order:      1
Name:           Admin
Description:     Super users and admins
Action:         allowed
Tag:
Risk:           1
Membership:     Admins
Submit Host(s): Any PBUL Host
Run Host(s):    Any PBUL Host
Commands may be executed as user(s): root,admin,user*
Please use the '-u' flag to select user at run time.
eg: pbrun -u runuser command [arguments]
User may request the following commands using pbrun:
Command Group:  User Commands
Description:    Common UNIX Commands
/bin/ls          executes: /bin/ls
/bin/ls *        executes: /bin/ls *
/usr/bin/ls      executes: /usr/bin/ls
/usr/bin/ls *    executes: /usr/bin/ls *
/bin/cat *       executes: /bin/cat *
/usr/bin/cat *   executes: /usr/bin/cat *
/bin/find *      executes: /bin/find *
/usr/bin/find *  executes: /usr/bin/find *
/bin/more *      executes: /bin/more *
/usr/bin/more *  executes: /usr/bin/more *
/bin/rm *        executes: /bin/rm -i $*
/usr/bin/rm *    executes: /usr/bin/rm -i $*
/bin/ln *        executes: /bin/ln *
/usr/bin/ln *    executes: /usr/bin/ln *
/bin/id          executes: /bin/id
/usr/bin/id      executes: /usr/bin/id
Command Group:  Admin Commands
Description:    Common Superuser Commands
/sbin/shutdown * executes: /sbin/shutdown *
/usr/sbin/shutdown * executes: /usr/sbin/shutdown *
/bin/mount *     executes: /bin/mount *
/usr/bin/mount * executes: /usr/bin/mount *
/sbin/ifconfig * executes: /sbin/ifconfig *
/usr/sbin/ifconfig * executes: /usr/sbin/ifconfig *

```

Level 3 report

=====

Privilege Management for Unix & Linux Role Based Policy Entitlement Report - Level 3

Date/Time: 2018-06-18 09:07:30

User: root

Belongs to the following Roles:

Admin

=====

Role Order: 1

Name: Admin

Description: Super users and admins

Action: allowed

Tag:

Risk: 1

Membership: Admins

Submit Host(s): Any PBUL Host

Run Host(s): Any PBUL Host

Commands may be executed as user(s): root,admin,user*

Please use the '-u' flag to select user at run time.

eg: pbrun -u runuser command [arguments]

User may request the following commands using pbrun:

Command Group: User Commands

Description: Common UNIX Commands

/bin/ls executes: /bin/ls

/bin/ls * executes: /bin/ls *

/usr/bin/ls executes: /usr/bin/ls

/usr/bin/ls * executes: /usr/bin/ls *

/bin/cat * executes: /bin/cat *

/usr/bin/cat * executes: /usr/bin/cat *

/bin/find * executes: /bin/find *

/usr/bin/find * executes: /usr/bin/find *

/bin/more * executes: /bin/more *

/usr/bin/more * executes: /usr/bin/more *

/bin/rm * executes: /bin/rm -i \$*

/usr/bin/rm * executes: /usr/bin/rm -i \$*

/bin/lm * executes: /bin/lm *

/usr/bin/lm * executes: /usr/bin/lm *

/bin/id executes: /bin/id

/usr/bin/id executes: /usr/bin/id

Command Group: Admin Commands

Description: Common Superuser Commands

/sbin/shutdown * executes: /sbin/shutdown *

/usr/sbin/shutdown * executes: /usr/sbin/shutdown *

/bin/mount * executes: /bin/mount *

/usr/bin/mount * executes: /usr/bin/mount *

/sbin/ifconfig * executes: /sbin/ifconfig *

/usr/sbin/ifconfig * executes: /usr/sbin/ifconfig *

Date and Time restrictions for Role 'Admin':

Time/Date Group: Any Time

Description: Any Time

Monday: 01:00am to 12:14pm

Tuesday: 01:00am to 12:14pm

Wednesday: 01:00am to 12:14pm

Thursday: 01:00am to 12:14pm

Friday: 01:00am to 12:14pm
Saturday: 01:00am to 12:14pm
Sunday: 01:00am to 12:14pm

Level 4 report

=====
Role Based Policy Entitlement Report - Level 4

Date/Time: 2018-06-18 09:07:32

User: root

Belongs to the following Roles:

Admin

=====
Role Order: 1
Name: Admin
Description: Super users and admins
Action: allowed
Tag:
Risk: 1
Membership: Admins
Submit Host(s): Any PBUL Host
Run Host(s): Any PBUL Host
Commands may be executed as user(s): root,admin,user*
Please use the '-u' flag to select user at run time.
eg: pbrun -u runuser command [arguments]
User may request the following commands using pbrun:
Command Group: User Commands
Description: Common UNIX Commands
/bin/ls executes: /bin/ls
/bin/ls * executes: /bin/ls *
/usr/bin/ls executes: /usr/bin/ls
/usr/bin/ls * executes: /usr/bin/ls *
/bin/cat * executes: /bin/cat *
/usr/bin/cat * executes: /usr/bin/cat *
/bin/find * executes: /bin/find *
/usr/bin/find * executes: /usr/bin/find *
/bin/more * executes: /bin/more *
/usr/bin/more * executes: /usr/bin/more *
/bin/rm * executes: /bin/rm -i \$*
/usr/bin/rm * executes: /usr/bin/rm -i \$*
/bin/ln * executes: /bin/ln *
/usr/bin/ln * executes: /usr/bin/ln *
/bin/id executes: /bin/id
/usr/bin/id executes: /usr/bin/id
Command Group: Admin Commands
Description: Common Superuser Commands
/sbin/shutdown * executes: /sbin/shutdown *
/usr/sbin/shutdown * executes: /usr/sbin/shutdown *
/bin/mount * executes: /bin/mount *
/usr/bin/mount * executes: /usr/bin/mount *
/sbin/ifconfig * executes: /sbin/ifconfig *
/usr/sbin/ifconfig * executes: /usr/sbin/ifconfig *
Date and Time restrictions for Role 'Admin':
Time/Date Group: Any Time

```
Description:      Any Time
Monday:  01:00am to 12:14pm
Tuesday:  01:00am to 12:14pm
Wednesday: 01:00am to 12:14pm
Thursday:  01:00am to 12:14pm
Friday:   01:00am to 12:14pm
Saturday:  01:00am to 12:14pm
Sunday:   01:00am to 12:14pm
Additional Role Options:
Additional Authentication Required: no
Session Recording Enabled: yes
Extended Script Policy: no
Custom accept/reject message: no
```

```
Level 1 report, with "command" filter
pbdbutil -P --rbp -R '{ "command":"/usr/bin/*"}'
```

```
=====
Privilege Management for Unix & Linux Role Based Policy Entitlement Report - Level 1
-----
```

```
Date/Time: 2018-06-18 09:09:10
```

```
User: *
```

```
Belongs to the following Roles:
```

```
Admin,users
```

```
=====
Role Order:      1
Name:            Admin
Description:     Super users and admins
Action:         allowed
Tag:
Risk:           1
Membership:     Admins
Submit Host(s): Any PBUL Host
Run Host(s):    Any PBUL Host
Commands may be executed as user(s): root,admin,user*
Please use the '-u' flag to select user at run time.
eg: pbrun -u runuser command [arguments]
User may request the following commands using pbrun:
/usr/bin/ls,/usr/bin/mount *,/usr/bin/ls *,/usr/bin/cat *,/usr/bin/find *,
/usr/bin/rm *,/usr/bin/ln *,/usr/bin/more *,/usr/bin/id
```

```
=====
Role Order:      4
Name:            users
Description:     Normal users
Action:         allowed
Tag:
Membership:     Users
Submit Host(s): nfs.company.com,build.company.com,staging.company.com
Run Host(s):    nfs.company.com,build.company.com,staging.company.com
Commands will execute as user: user*
User may request the following commands using pbrun:
/usr/bin/ls,/usr/bin/ls *,/usr/bin/find *,/usr/bin/cat *,/usr/bin/ln *,
/usr/bin/rm *,/usr/bin/more *,/usr/bin/id
```

Level 4 report with "command" filter

=====

Privilege Management for Unix & Linux Role Based Policy Entitlement Report - Level 4

Date/Time: 2018-06-18 09:09:26

User: *

Belongs to the following Roles:

Admin,users

=====

Role Order: 1

Name: Admin

Description: Super users and admins

Action: allowed

Tag:

Risk: 1

Membership: Admins

Submit Host(s): Any PBUL Host

Run Host(s): Any PBUL Host

Commands may be executed as user(s): root,admin,user*

Please use the '-u' flag to select user at run time.

eg: pbrun -u runuser command [arguments]

User may request the following commands using pbrun:

Command Group: Admin Commands

Description: Common Superuser Commands

/usr/bin/mount * executes: /usr/bin/mount *

Command Group: User Commands

Saturday: 01:00am to 12:14pm

Description: Common UNIX Commands

/usr/bin/ls executes: /usr/bin/ls

/usr/bin/ls * executes: /usr/bin/ls *

/usr/bin/cat * executes: /usr/bin/cat *

/usr/bin/find * executes: /usr/bin/find *

/usr/bin/more * executes: /usr/bin/more *

/usr/bin/rm * executes: /usr/bin/rm -i \$*

/usr/bin/ln * executes: /usr/bin/ln *

/usr/bin/id executes: /usr/bin/id

Date and Time restrictions for Role 'Admin':

Time/Date Group: Any Time

Description: Any Time

Monday: 01:00am to 12:14pm

Tuesday: 01:00am to 12:14pm

Wednesday: 01:00am to 12:14pm

Thursday: 01:00am to 12:14pm

Friday: 01:00am to 12:14pm

Saturday: 01:00am to 12:14pm

Sunday: 01:00am to 12:14pm

Additional Role Options:

Additional Authentication Required: no

Session Recording Enabled: yes

Extended Script Policy: no

Custom accept/reject message: no

=====

Role Order: 4

Name: users

Description: Normal users

```
Action:          allowed
Tag:
Risk:            1
Membership:      Users
Submit Host(s): build.company.com,nfs.company.com,staging.company.com
Run Host(s):    build.company.com,nfs.company.com,staging.company.com
Commands will execute as user: user*
User may request the following commands using pbrun:
Command Group:  User Commands
Description:    Common UNIX Commands
/usr/bin/ls          executes: /usr/bin/ls
/usr/bin/ls *        executes: /usr/bin/ls *
/usr/bin/cat *       executes: /usr/bin/cat *
/usr/bin/find *      executes: /usr/bin/find *
/usr/bin/more *      executes: /usr/bin/more *
/usr/bin/rm *        executes: /usr/bin/rm -i $*
/usr/bin/ln *        executes: /usr/bin/ln *
/usr/bin/id          executes: /usr/bin/id
Date and Time restrictions for Role 'users':
Time/Date Group: Working Week
Description:       Working Week
Monday:  01:00am to 12:14pm
Tuesday: 01:00am to 12:14pm
Wednesday: 01:00am to 12:14pm
Thursday: 01:00am to 12:14pm
Friday:  01:00am to 12:14pm
Saturday: none
Sunday:  none
Additional Role Options:
Additional Authentication Required: no
Session Recording Enabled: no
Extended Script Policy: no
Custom accept/reject message: no
```

Policy File Format

In most cases, the order of the instructions in a security policy file is not important. The user's security requirements determine the rules that the file contains.

User-Written Functions and Procedures

To help simplify Security Policy implementation, the Privilege Management for Unix & Linux Security Policy scripting language enables the Security Administrator to write custom functions and procedures (that is, user-written functions and procedures).



Note: For the remainder of this discussion, the term "function" refers to both user-written functions and procedures. The differences between the two are discussed in "Functions and Procedures" on page 75.

Think of functions as stand-alone units of security code that perform specific programming tasks. After a function is written, the function can be invoked from within any security policy file to perform its specific task or function. It is a good idea to write functions for repetitive programming tasks. Doing so enables the policy instructions to be written once and utilized in multiple places.

Another benefit of using functions is that any needed changes can be made in only one place. By centralizing the logic for a repetitive type task in one place (that is, a single function), all of the security policy files that call the function automatically benefit from any updates that are made to the function. The following figure illustrates the basic structure of a function.

When a user-written function is used within a security policy file, the code for that function is placed at the top of the security policy file that first references it. In other words, the overall structure of a security policy file is all user-written functions first, followed by security policy code.

file: pb.conf

```
function FUCNTIONNAME(arguments)
{
    function statements go here
    _____
    FUCNTIONNAME=value
}
```



For more information on creating functions and procedures, please see "Functions and Procedures" on page 75.

A good way to manage and organize user-written functions is to logically group all functions that perform similar types of tasks in a security policy file. Now, add **include** statements for each of these sub files to the beginning of the **pb.conf** file. These **include** statements should come before anything else. When this is done, the functions that are contained within these sub files can be called from within any security policy file.

Variable Scope

Security Policy variables are global. In other words, after a variable has been implicitly defined, it can be referenced from any security policy file. The use of a variable is not limited to the security policy file in which it was implicitly defined (that is, used for the first time).

If a variable is implicitly created in one security policy file and referenced by another, both files access and modify the same variable.

Syntax Checking

Always check the syntax of a security policy file before putting it into production. If a request encounters a security policy file syntax error, then the task that causes the error is immediately rejected. The "reject" event is logged in the Privilege Management for Unix & Linux Event Log.

Syntax checking is done with **pbcheck**, a Privilege Management for Unix & Linux utility program. It performs two functions:

- Security policy file syntax validation
- Simulates security processing for test task requests to determine if that task request would be accepted or rejected during production processing



For more information on how to use **pbcheck**, please see the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

Policy Debugging

Policies can be debugged via the **pbadmin --poldbg** command.



For more information, please see the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

Environment Variable Processing Considerations

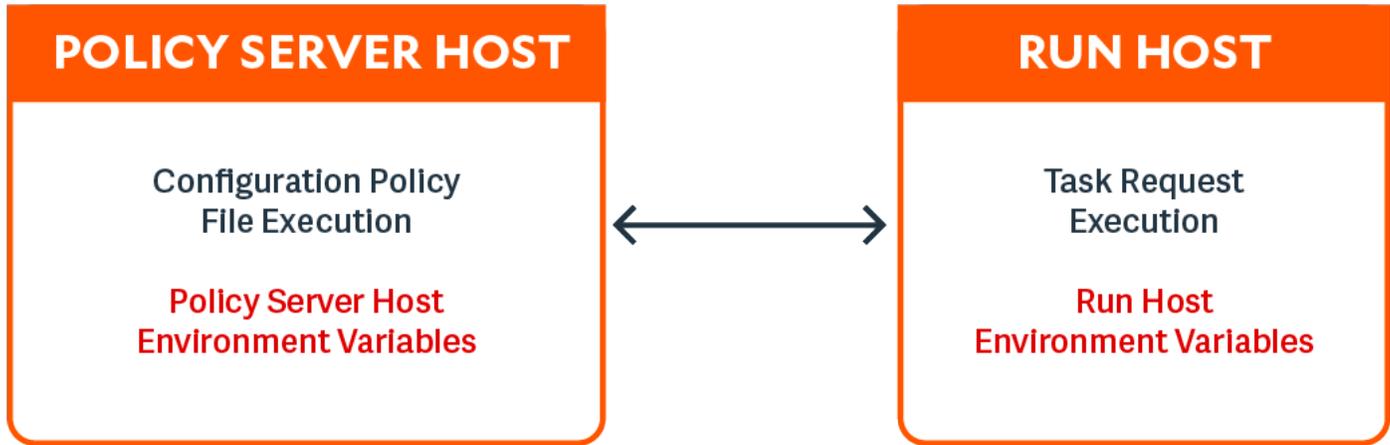
As discussed earlier, it is possible to install **pbrun**, **pbmasterd**, **pblocald**, and/or **pblogd** on different machines (that is, the submit host, Policy Server host, run host, and log host may represent different physical machines). When this is the case, each of these separate machines can have its own set of users, groups, and environment variables, which can differ from host to host.



Note: If **pbrun**, **pbmasterd**, and/or **pblocald** are installed on different machines, then the environment variables on those machines can contain different values.

For instance, a user might have one home directory on the submit host and another on the run host. In another example, a user group list on Policy Server host can be different from the same user group list on the run host. This situation might arise if the Policy Server host is not an NIS client or has fewer entries in its **/etc/passwd** file.

As shown in the following figure, security policy file processing always takes place on the Policy Server host machine, while task execution takes place on the run host machine. When the Policy Server host and run host represent different machines, by default, it is the user and group information on the Policy Server host machine that is accessed during Security Profile File processing. If it is necessary to access users or groups only on the run host machine, then special pass-through values must be used. When these values are encountered during Security Profile File processing, **pbmasterd** passes through the value to the run host machine to be resolved when the task is run.



Note: The `execute_via_su` mechanism enables the runhost's environment for the runuser, overriding the run environment that the policy on the Policy Server has set up. Note also that the `runenvironmentfile` feature can also be used to add runhost specific environment variables.



For more detailed information on using pass-through values, please see "[Task Information Variables](#)" on page 85.

Support for Multiple-Byte Character Sets

The Privilege Management for Unix & Linux policy language supports the processing of UTF-8 encoded multiple-byte character strings. In addition, several variables (indicated by `i18n_` in their names) format UTF-8 encoded date and time values according to the operating system's locale settings.

Security Policy Scripting Language Definition

The security policy scripting language is an interpreted programming language. Its syntax is similar to the C language. Like C, it is case-sensitive. This chapter contains detailed information about using the Privilege Management for Unix & Linux security policy scripting language.

Variables and Data Types

A variety of variables and data types are available in the Privilege Management for Unix & Linux security policy scripting language. These are described in the following sections.

Variables

Privilege Management for Unix & Linux uses predefined system variables to store both system and task-specific information. These variables are a valuable resource to the Security Administrator because they can be accessed and manipulated from within security policy files with the security policy scripting language. The information in these variables can play a critical role in determining whether a task request should be accepted or rejected. System variables can also be used to set runtime properties, including logging options, for a specific task request.

In addition to predefined system variables, the Security Administrator can create and manipulate user-defined variables to assist with security policy file processing. User-defined variables are implicitly defined, meaning the interpreter automatically allocates storage for a user-defined variable the first time that variable is referenced. In the Privilege Management for Unix & Linux security policy scripting language, there is no need to formally declare a variable before using it. Consequently, the language does not provide a mechanism for explicitly defining a variable type. A variable's type is implicitly defined by the information that is stored in that variable. After a variable has stored a specific type of information, it cannot store information of a different data type.

Observe the following rules when creating user-defined variables:

- Variable names can be any length.
- The first character of a variable name must be a letter or an underscore character. The remaining characters can be letters, numerals, or underscores.
- Variable names are case sensitive. For example, the variable names **currentuser** and **CurrentUser** represent two different and unique variables.

Example:

```
MyVariable = "123"; # Create a user-defined variable.  
LoopCounter = 1; # Create a user-defined variable.  
_CurrentUser = "Tom"; # Create a user-defined variable.  
runuser = "SysAdm"; # Set a predefined system variable.
```

Variable Scope

With the exception of function parameters, all Privilege Management for Unix & Linux variables are global in scope. (In this context, the function name inside a function behaves like a function parameter.) This means that if a user-defined variable is implicitly defined in a security policy file and referenced in another security policy file, both files access the same variable.

Function parameters, also called function arguments, do not work differently from other variables. Function argument storage for a specific Security Policy function is deleted when that Security Policy function completes execution.

Variable Data Types

The data type, or type of information that is stored in a variable, determines the type of operations you can perform on the variable. Privilege Management for Unix & Linux supports the following data types:

- Character strings
- Integers
- LDAP connections
- LDAP messages
- List of character strings

Character String

The character string, or string, data type is a sequence of zero or more characters, enclosed by single or doublequotation marks. It is important to note that arithmetic functions cannot be performed on character strings. For instance, the character string **"123"** cannot be used in an arithmetic operation although it contains numeric characters. As another example, the character string **"12"** is not the same as the number **"12"**. A value that is enclosed in quotation marks is always stored as a character string. In other words, the security policy scripting language interpreter treats numeric values and numeric character strings differently. They are not interchangeable.

The following table lists character string examples and how they are interpreted.

Example	Interpreted As
"abc"	Character string.
""	Empty character string.
"0123456789"	Numeric character string.
'abc'	Character string.

Integer

Integers are numeric values used to perform arithmetic operations. It is important to note that the value **12**, which is a numeric value, is not the same as the value **"12"**, which is a character string. The security policy scripting language interpreter treats numeric values and numeric character strings differently. They are not interchangeable.

The integer data type can store any integer value (that is, the set of both positive and negative whole numbers). An octal number (base 8) is specified by prefixing the octal value with a leading zero (for example, **022**). A hexadecimal number (base 16) is specified by preceding the hexadecimal value with **"0x"** (for example, **0x5A**).

The following table lists the valid integer characters.

Basic	Valid Characters
Octal	0, 1, 2, 3, 4, 5, 6, 7
Decimal	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The policy language does not support fractional (or floating-point) values. Integer values cannot include characters such as commas, dollar signs, or decimal points.

The integer values **0** and **1** have special meaning within the security policy scripting language. The integer value of **0** represents the Boolean **false** value. The integer value of **1** represents the Boolean **true** value.



For more information on Boolean values, please see ["Boolean True and False Variables" on page 77](#).

The following table provides several examples on the use of integer variables.

Example	Result
RejectCount = 0;	Set RejectCount to 0
UserLimit = 10;	Set UserLimit to 10
OctNumber = 022;	Set an octal variable to 18
HexNumber = 0x7a;	Sets an integer to a hexadecimal value of 122

LDAP Connection

The LDAP connection is a special data type that is used solely for passing parameters to and from the Privilege Management for Unix & Linux LDAP functions.



For more information on Privilege Management for Unix & Linux LDAP functions, please see ["LDAP Functions" on page 238](#).

LDAP Message

The LDAP message is a special data type. It is used only to pass parameters to and from the Privilege Management for Unix & Linux LDAP functions.



For more information on Privilege Management for Unix & Linux LDAP functions, please see ["LDAP Functions" on page 238](#).

List of Character Strings

A list of character strings, also called a list, is an ordered group of character strings, separated by commas and surrounded by curly braces {}. It has the syntax:

```
{ string-one, string-two, ...}
An empty list is represented as { }
Assignment to a list has the syntax:
name = { string-one, string-two, ...}
Assignment to an element of a list can be done by:
name[1] = "string-three"
```

Think of a list as a one-dimensional array consisting of zero or more elements (refer to the example). A list can contain only character string data (that is, a list cannot contain integer values, LDAP related types, or other lists).

Individual list elements are accessed using an index number. Square brackets enclose the index number and postfix the list name (see the following example).

Index numbering starts at 0. This means that the first element in a list has an index of 0, the second element has an index of 1, and so on. For example, the fifth element in a list has an index number of 4.

For example, the list

```
UserList = {"JWhite", "BSmith", "CDent"};
```

results in the following:

```
UserList[0] is "JWhite"  
UserList[1] is "BSmith"  
UserList[2] is "CDent"
```

As a second example, assume we have the list

```
TrustedUsers = {"JWhite", "BSmith");  
User1 = TrustedUsers [0];  
User2 = TrustedUsers [1];  
MyString = { "a", "b", "c" }[1];
```

In this list,

```
User1 = TrustedUsers [0]; sets User1 to "JWhite"  
User2 = TrustedUsers [1]; sets User2 to "BSmith"  
MyString = { "a", "b", "c" }[1]; sets MyString = "b"
```

Constants

A constant is a value that is not modified during security policy file execution. The following table contains examples of the different constant types.

Table 4. Constant Examples

Constant Type	Examples
Integer Constant	12, 54, -100, 08, 0x1a
List Constant	{"user1", "user2", "user3"}
String Constants	"12", "ABCD"

Operators

An operator is a symbol that performs a specific mathematical, relational, or logical function. The security policy scripting language supports the types of operators that are listed in the following table.

Operator Type	Symbols
Arithmetic Operators	*, /, +, -, %, ++, --, +=, -=, *=, /=, %=
Logical Operators	&&, , !
Relational Operators	>, >=, <, <=, ==, !=
Special Operators	(), [], +, ?:, in, ,

Every operator has an intrinsic precedence order associated with it. The precedence order determines the evaluation order for expressions containing more than one operator. The operator with the highest precedence evaluates first. In most cases, operators of the same precedence are evaluated left to right. The following table lists the operator precedence.

Precedence	Operator	Associativity
Highest	{ }	Left to right
	() []	Left to right
	in	Left to right
	!++--	Right to left
	- (unary)	Left to right
	*/%	Left to right
	+ -	Left to right
	<><=>=	Left to right
	==!=	Left to right
	&&	Left to right
		Left to right
	?:	Right to left
	+=-=*/%=	Right to left
Lowest	,	Left to right

For example, following the rules of operator precedence, the statement

```
5 + 6 - 3 * 4 + 8 / 4
is resolved as:
Step 1: 3*4 = 12
Step 2: 8/4 = 2
```

```
Step 3: 5 + 6 - (12) + (2)
Result: 1
```

Modifying the operator precedence order as shown here can change the result produced in the example above.

```
(5 + 6 - 3) * (4 + 8) / 4
The statement is resolved as follows:
Step 1: 5 + 6 - 3 = 8
Step 2: (4 + 8) = 12
Step 3: 8 * 12 / 4
Result: 24
```

Arithmetic Operators

The Privilege Management for Unix & Linux security policy scripting language supports the arithmetic operators shown in the following table.

Operator	Description
++	Prefix autoincrement
--	Prefix autodecrement
++	Postfix autoincrement
--	Postfix autodecrement
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
+=	Addition self assignment
-=	Subtraction self assignment
*=	Multiplication self assignment
/=	Division self assignment
%=	Modulus self assignment

The subtraction, addition, multiplication and division operators perform arithmetic operations. The default evaluation order for arithmetic operators is:

- Multiplication, division, and modulus division, left to right.
- Addition and subtraction, left to right.

In the example,

```
result = 6 * 4 / 2 - 4 + 2;
```

result would contain the integer value **10**.

Prefix Autoincrement Operator

Description

The prefix autoincrement operator (**++**) adds one to a variable and returns the result.

Example

```
a = 3;
```

```
b = ++a;
```

In this example, both **a** and **b** are 4.

Prefix Autodecrement Operator

Description

The prefix autodecrement operator (**--**) subtracts one from a variable and returns the result.

Example

```
a = 3;
```

```
b = --a;
```

In this example, both **a** and **b** are 2.

Postfix Autoincrement Operator

Description

The postfix autoincrement operator (**++**) returns the value of a variable and adds one to the variable.

Example

```
a = 3;
```

```
b = a++;
```

In the example, **a** is 4 and **b** is 3.

Postfix Autodecrement Operator

Description

The postfix autodecrement operator (`--`) returns the value of a variable and subtracts one from the variable.

Example

```
a = 3;
```

```
b = a--;
```

In the example, **a** is 2 and **b** is 3.

Addition Operator

Description

The addition operator (`+`) adds two numbers.

Example

```
result = 5 + 3;
```

Subtraction Operator

Description

The subtraction operator (`-`) subtracts two numbers.

Example

```
result = 5 - 3;
```

Multiplication Operator

Description

The multiplication operator (`*`) multiplies two numbers.

Example

```
result = 5 * 3;
```

Division Operator

Description

The division operator (/) divides two numbers.

Example

```
result = 5 / 3;
```

Modulus Operator

Description

The modulus operator (%) returns the remainder of integer division.

Example

```
result = 5 % 3;
```

In this example, **result** contains the integer value **2**. Dividing **5** by **3** yields a result of **1** and a remainder of **2**. The remainder portion of the answer, in this case **2**, becomes the result of the modulus division operation.

Addition Self-assignment Operator

Description

The addition self-assignment operator (+=) adds a value to a variable and stores the result in the variable.

Example

```
a += 3;
```

In this example, **3** is added to **a** and the result is stored in **a**.

Subtraction Self-assignment Operator

Description

The subtraction self-assignment operator (-=) subtracts a value from a variable and stores the result in the variable.

Example

```
a -= 4;
```

In this example, **4** is subtracted from **a** and the result is stored in **a**.

Multiplication Self-assignment Operator

Description

The multiplication self-assignment operator (`*=`) multiplies a variable by a value and stores the result in the variable.

Example

```
a *= 5;
```

In this example, **a** is multiplied by **5** and the result is stored in **a**.

Division Self-assignment Operator

Description

The division self-assignment operator (`/=`) divides a variable by a value and stores the result in the variable.

Example

```
a /= 6;
```

In this example, **a** is divided by **6** and the result is stored in **a**.

Modulus Self-assignment Operator

Description

The modulus self-assignment operator (`%=`) divides a variable by a value and stores the modulus in the variable.

Example

```
a %= 5;
```

In this example, **a** is divided by **5** and the remainder is stored in **a**.

Logical Operators

The Privilege Management for Unix & Linux security policy scripting language supports a standard set of logical operators.

Operator	Action
&&	<p>AND</p> <p>In Privilege Management for Unix & Linux versions 3.2 and earlier, logical expressions containing the && operator were evaluated before determining the result.</p> <p>Beginning with Privilege Management for Unix & Linux version 3.5, logical expressions containing the && operator stop evaluation when a false value is found.</p>

	<p>OR</p> <p>In Privilege Management for Unix & Linux versions 3.2 and earlier, logical expressions containing the operator were evaluated before determining the result.</p> <p>Beginning with Privilege Management for Unix & Linux version 3.5, logical expressions containing the operators stop evaluation when a true value is found.</p>
!	<p>NOT</p>

AND Operator

Description

The **AND** operator (**&&**) considers the relationship between two values. Both values must be **true** for a **true** result to be returned. If both values are **true**, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

In Privilege Management for Unix & Linux 3.2 and earlier, all parts of logical expressions containing **&&** operators are evaluated before determining the result.

Beginning with Privilege Management for Unix & Linux 3.5, logical expressions containing **&&** operators are evaluated from left to right until their truth can be determined (like in the C language).

Example

```
if (UserOkay && Bkup) accept;
```

If both **UserOkay** and **Bkup** are non-zero, the current task request is accepted.

OR Operator

Description

The **OR** operator (**||**) considers the relationship between two values. At minimum, one of the two values must be **true** for a **true** result to be returned. If either the first or second value is **true**, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

In Privilege Management for Unix & Linux 3.2 and earlier, all parts of logical expressions that contain || operators were evaluated before determining the result.

Beginning with Privilege Management for Unix & Linux 3.5, logical expressions that contain || operators are evaluated from left to right until their truth can be determined (like in the C language).

Example

```
if (UserOkay || Bkup) accept;
```

If either **UserOkay** or **Bkup** are non-zero, the current task request is accepted.

NOT Operator

Description

The **NOT** operator (**!**) takes the inverse of a value. If a value is **false**, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

```
if (!UserOkay) reject;
```

If **UserOkay** is equal to **0**, the current task request is rejected.

Relational Operators

The Privilege Management for Unix & Linux security policy scripting language supports a standard set of relational operators.

Operator	Description
==	Equal To
>	Greater Than
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To
!=	Not Equal To

Equal To Operator

Description

The **Equal** operator (**==**) compares two values. If the first value is equal to the second value, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

```
if (UserCount == 10) reject;
```

If **UserCount** is equal to **10**, the current task request is rejected.

Greater Than Operator

Description

The **Greater Than** (**>**) operator compares two values. If the first value is greater than the second value, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

```
if (UserCount > 10) reject;
```

If **UserCount** is greater than **10**, the current task request is rejected.

Greater Than or Equal To Operator

Description

The Greater Than or Equal To (**>=**) operator compares two values. If the first value is greater than or equal to the second value, then an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

In this example, if **UserCount** is greater than or equal to **10**, then the current task request is rejected.

```
if (UserCount >= 10) reject;
```

Less Than Operator

Description

The **Less Than** operator (**<**) compares two values. If the first value is less than the second value, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

```
if (UserCount < 10) reject;
```

If **UserCount** is less than **10**, the current task request is rejected.

Less Than or Equal To Operator

Description

The **Less Than or Equal** operator (**<=**) compares two values. If the first value is less than or equal to the second value, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

```
if (UserCount <= 10) accept;
```

If **UserCount** is less than or equal to **10**, the current task request is accepted.

Not Equal To Operator

Description

The **Not Equal To** operator (`!=`) compares two values. If the first value is not equal to the second value, an integer value of **1 (true)** is returned. Otherwise, an integer value of **0 (false)** is returned.

Example

```
if (UserCount != 10) reject;
```

If **UserCount** is not equal to **10**, the current task request is rejected.

Special Operators

The Privilege Management for Unix & Linux security policy scripting language supports the special operators.

Operator	Description
+	Concatenation
[]	List index
in	List member
()	Precedence (that is, parentheses)
?:	Ternary conditional
,	Evaluates terms from left to right; returns the value of the last expression

Concatenation Operator

Description

The **Concatenation** operator **+** is used to concatenate a series of one or more strings. It should not be confused with the **Addition** operator used in arithmetic expressions. Although both of these operators are represented by the **+** symbol, the **Addition** operator works only on integer values.

The Concatenation operator concatenates, or appends, one item to another item. If a series of strings are concatenated, they are returned in a newly created string.

Example

```
FirstName = "Sandy";
LastName = "White";
UserName = FirstName + " " + LastName;
```

UserName would contain the character string **"Sandy White"**.

List Index Operator

Description

The **List Index** operator **[]**, also referred to as square brackets, is used to specify a list element index number. The value of a specific list element is returned.

The first element in a list always has an index number of **0**, and the second list element has an index of **1**, etc. The general formula for calculating an index number is **index number = element number - 1**.

Example

```
UserList = {"Adm1", "Adm2", "Adm3", "Adm4", "Adm5"};
CurrentUser = UserList[3];
```

CurrentUser contains the character string **"Adm4"**.

Example

```
UserList[1] = "Adm10";  
Userlist[1] is set to "Adm10".
```

List Member Operator

Description

This list member operator, **in**, searches the specified list for the given string. If the string is present in the list, the result is **true (1)**. If the string is not present, it returns **true (0)**. Shell-style wildcards can be used in the string argument. The syntax for using this operator is **result = string in list**;

Example

```
AdminList = {"Adm1", "Adm2", "Adm3", "root", "sys"};  
runuser = (user == "sysadmin")? "root" : "sys";  
test1 = "Adm1" in AdminList; # True  
test2 = "sys" in AdminList; # True - matches sys in AdminList  
test3 = "system" in AdminList; # False  
test4 = "Adm" in AdminList; # False - only a partial match  
# single character
```

Each string is tested to see if it is a member of a list.

Precedence Operator

Description

The **Precedence** operator (**()**), also referred to as parentheses, is used to modify the default operator precedence. In other words, parenthesis characters force a specific expression evaluation order.

Example

```
result = (6 + 4) * 2 - 4;
```

result contains the integer value **16**.

Example

```
result = 6 + 4 * 2 - 4;
```

The Precedence operators are removed, and the **result** contains the integer value **10**.

Ternary Conditional Operator

Description

The **Ternary** operator, represented by `?;`, is a special operator that provides a compact alternative to **if** statements where only an expression is required.

The Ternary operator has the syntax:

```
result = condition ? if-true-expression : if-false-expression;
```

The ternary operator works as follows:

- If **condition** evaluates to **true**, then the **if-true-expression** is returned.
- If **condition** evaluates to **false**, then the **if-false-expression** is returned.

The Ternary operator can be used as an alternative to simple **if** statements. The **condition** corresponds to the **if condition**. The **if-true-expression** corresponds to the assignment in the true part of the **if** statement, and the **if-false-expression** corresponds to the else part of the **if** statement.

Example

```
runuser = (user == "sysadmin") ? "root" : "sys";
```

If **user** is equal to **sysadmin**, then **root** is returned. Otherwise, **sys** is returned.

Another way to accomplish the same thing would be to use the following **if** statement:

```
if (user == "sysadmin")
runuser = "root";
else
runuser = "sys";
```

Comma Operator

Description

The **Comma** operator (`,`) causes expressions to be evaluated from left to right and returns the value of the last expression. This operator is primarily used in loops.

Example

```
for (a=0, b=1, c=2; a < 0 ; a++) <any statement>;
```

The Comma (`,`) operator causes the assignment of the three variables **a**, **b**, and **c** at a spot which looks for a single expression.

Expressions

An expression is a combination of constants, variables, and operators. Expressions are evaluated according to operator precedence rules. Most expressions follow the general rules of Algebra in regards to operator precedence. The following is an example of an expression:

```
TotalTasks = RejectedCount + AcceptedCount;
```

In Privilege Management for Unix & Linux 3.2 and earlier, expressions and variables could not be used interchangeably.

Beginning with Privilege Management for Unix & Linux 3.5+, assignments can be performed anywhere expressions are found.



For more information on operator precedence, please see "Constants" on page 1.

Program Statements

There are two types of program statements in the Privilege Management for Unix & Linux security policy scripting language, **Executable** and **Non-executable**.

Executable Program Statements

Executable program statements allows security administrators to define and implement security rules. These types of statements have two major functions:

- Set the environment in which security profile files run
- Control the logic flow within security policy files

The following table summarizes the executable program statements:

Statement	Description
accept	Terminates security policy file processing and passes control to pblocald . Version 4.0 and earlier: statements do not support ACL. Version 5.0 and later: statements support ACL.
Assignment	Used to assign a value to a variable.
break	Terminates the processing of cases within a loop and exits the loop Version 3.2 and earlier: statements are limited to ending a case clause in a switch statement Version 3.5 and later: statements are expanded for use within loops
continue	Allows the remaining loop body to be skipped. Returns to the next iteration of the loop
do-while	Creates do-while loops which follow the C language syntax
for	C-style for . Used to create for loops which follow the C language syntax
for-in	Creates loops that execute the loop body for each element in an argument list
function	Stand-alone subroutines that are used to modularize a company's security policy file
if	Determines which program statement to execute next based on whether an expression is true or false
include	Passes the flow of control to another file
procedure	Stand-alone subroutines used to modularize a company's security policy files
readonly	Freezes the value of a variable so it cannot be changed by a security policy file
reject	Immediately terminates security policy file checking and cancels the current job request before it can execute Version 4.0 and earlier: statements do not support ACL. Version 5.0 and later: statements support ACL.
switch	Provides a way to execute a specific set of program statements based on an expression value

while	Builds while loops which follow the C language syntax
--------------	--

Type your executable program statements in lowercase because the security policy scripting language interpreter is case sensitive. For example, the word **If** is recognized as a variable name by the interpreter whereas the word **if** is recognized as an executable program statement.

Some general rules for creating program statements are as follows:

- Terminate program statements with a semicolon (;)
- A single statement can be multiple lines.
- Multiple statements can be included on one line if each statement terminates with a semicolon
- Enclosing groups of program statements within curly brackets creates a compound statement. Each statement within the group must terminate with a semicolon.

Executable program statements have a special meaning to the security policy scripting language interpreter. Therefore, you cannot use them for other purposes. For instance, using an executable program statement as a variable name generates an error.

Many administrators desire a non-programmatic way of using Privilege Management for Unix & Linux. To accomplish this goal, the Privilege Management for Unix & Linux Policy Language was extended in Privilege Management for Unix & Linux version 5.0 to include an **Access Control List** structure. This structure extends the **accept** and **reject** statements to provide a simple non-programmatic way of specifying access data. It can be used exclusively to provide control, or it can be used in combination with the rest of the Privilege Management for Unix & Linux Policy Language to provide greater control.



For more information on expressions, please see "[Expressions](#)" on page 59. For more information about function and procedure statements, please see "[Functions and Procedures](#)" on page 75.

accept Statement

- **Version 4.0 and earlier:** **accept** statement does not support ACL
- **Version 5.0 and later:** **accept** statement supports ACL

Description

When an **accept** statement is encountered, security policy file processing terminates immediately, **pblocald** starts, and the secured task is executed by **pblocald**.

Syntax

All versions:

```
accept;
```

Version 5.0 and later:

```
accept [from ["user"][, ["submithost"][, ["command"]
[, ["runhost"]]]]] [when conditional-expression]
[with optional-statements-before-execution];
```

Definition

- **user** is a user name, list of user names, or left blank to imply any user.
- **submithost** is a submit host name, list of submit hosts, or left blank to imply any submit host.
- **command** is a command, list of commands, or left blank to imply any command.
- **runhost** is a run host, list of run hosts, or left blank to imply any run host.
- **conditional-expression** is an expression that evaluates **true** or **false**.
- **optional-statements-before-execution** is one or more Privilege Management for Unix & Linux Policy Language statements that executes before the requested command is executed. For multiple statements, separate each statement with a comma.

Example

All versions:

```
if (user == "HelpDesk1") accept;
```

If **user** is equal to **HelpDesk1**, the task is accepted and allowed to execute. Security policy file processing immediately terminates. **pblocald** starts, and the information is sent from the Policy Server for **pblocald** to start the executable specified in the variable **runcommand**. It is run by **pblocald** with the arguments specified in the **runargv** variable and run as the user specified in the **runuser** variable. Other run variables can be set.

Version 5.0 and later:

- Accept all commands for **user1** from any submit host and for any run host:

```
accept from "user1";
```

- Accept all commands for **user1** when the request comes from submit host **host1** for any run host:

```
accept from "user1", "host1";
```

- Accept the **date** command from **user1** from any submit host and for any run host:

```
accept from "user1", "date";
```

- Accept all commands from **user3**, from any submit host and for any run host, when the time is between 9:00 A.M. and 5:00 P.M.:

```
accept from "user3" when timebetween(900, 1700);
```

- Accept a **sh** command from **user1** or **user3**, from any submit host and for any run host, and turn on I/O logging:

```
accept from {"user1", "user3"}, "sh" with iolog = "/var/log/pb.iolog.sh";
```

- Accept all commands from all users, from any submit host and for any run host, when the time is between 9:00 A.M. and 5:00

P.M.:

```
accept when timebetween(900, 1700);
```

See Also

```
reject
```

Assignment Statement

Description

An assignment statement assigns a value to a variable. An assignment can be used whenever an expression is expected, and multiple assignments can be done in a single statement.

In Privilege Management for Unix & Linux 3.2 and earlier, assignments were not expressions and could not be cascaded.

Beginning with Privilege Management for Unix & Linux 3.5+, assignments are expressions and can be cascaded anywhere an expression occurs.

Syntax

```
list[n] = expression;
```

An expression can be a constant, variable, or complex equation.

```
var1 = var2 = var3 ... = value;
```

var1, **var2**, and **var3** are assigned values.

Example

```
IntegerString = "1234";  
StringList = {"User1", "User2", "User3"};  
Counter = 1;  
TotalUsers = 5;  
CurrentUsers = 3;  
InactiveUsers = TotalUsers - CurrentUsers;  
userString = user;  
runuser = "root";  
list1 = {"a1", "a2", "a3"};  
list2 = list1;  
list2[0] = "11"
```

The following occurs:

```
InactiveUsers is set to 2 (5 - 3)  
userString = user; sets userString to the submitting user.  
runuser = "root"; sets runuser to root.
```

```
list2[0] = "l1" causes list1 to still be {"a1", "a2", "a3"}, list2 has the value of {"l1", "a2", "a3"}
```

Example

```
a = b = c = d = 0;
```

The variables **a**, **b**, **c**, and **d** are cascaded and assigned the same value (**0**).

break Statement

Description

The **break** statement exits loops and terminates cases. In Privilege Management for Unix & Linux 3.2 and earlier, the break statement was used only to end a case clause in a **switch** statement.

Beginning with Privilege Management for Unix & Linux 3.5, the break statement is used within loops as well as to end a clause in a **switch** statement.

Syntax

```
break;
```

Example

```
for (a = 1 ; a <= 10; a++) {  
  if (a > 5) break;  
  print (a);  
}
```

The statement prints the numbers between 1 through 5.

See Also

```
continue, do-while, for, for-in, while
```

continue Statement

Description

The **continue** statement is used in the body of a C-style **for**, **while**, or **do-while** statement to skip the rest of statements in the body.

Syntax

```
continue;
```

Example

```
for (a = 1 ; a <= 10; a++) {  
  if (a % 2 != 0) continue;  
  print (a);  
}
```

The statement prints the even numbers between 1 and 10.

See Also

break, do-while, for, for-in, while

do-while Statement

Description

The C-style **do-while** statement is used to execute a loop. The body that follows the while statement can be a single statement or set of statements inside braces ({ and }). This statement is executed as follows:

1. The body is executed.
2. If a **break** statement is encountered in the body, the loop terminates.
3. The test expression is evaluated.
4. If the test expression is **false** (0), the loop terminates.
5. If the test expression is **true** (non-zero), steps 1 through 4 are repeated until a **break** statement is encountered or the test expression becomes **false**.

The body is always executed at least once.

Syntax

```
do body while (test_expression);
```

Example

```
a = 1;  
do print(a++);  
while (a <= 10);
```

The statement prints the numbers 1 through 10.

See Also

break, continue, for, for-in, while

for Statement

Description

The **for** statement provides a mechanism to loop through or to repeat a series of program statements. In Privilege Management for Unix & Linux 2.8 and earlier, the **for** statement always terminated with an **end** statement. This is no longer necessary in Privilege Management for Unix & Linux 3.0+.

Syntax

```
for ControlValue = StartValue to StopValue [step Increment]
{executable program statements}
```

The **for** statement works in the following manner:

1. The first time through the **for** statement, **ControlValue** is set to **StartValue**.
2. **ControlValue** is immediately compared to **StopValue**.
3. After an execution of the **for** statement has been completed and all associated program statements have been executed, **StartValue** is incremented by the **step** value.
4. If a **step** value was not specified, a default **step** value of **1** is used. **ControlValue** is again compared to **StopValue** and the result of this comparison determines if the **for** statement executes again.

The comparison of **ControlValue** to **StopValue** works as follows:

1. When the **Increment** value is positive, the **for** statement is executed as long as **ControlValue** \leq **StopValue** evaluates to **true**.
2. When the **Increment** value is negative, the **for** statement is executed as long as **ControlValue** \geq **StopValue** evaluates to **true**.
3. When the **Increment** value is **0**, the **for** statement executes forever. An **accept** or **reject** is required to break out of the loop.
4. If an **Increment** is not specified, **1** is used as the increment value.



Note: The **for** statement loop condition is tested at the top of the loop, and there is no guarantee the **for** loop will execute.

Example

In the **for** statement

```
for LoopCounter = 0 to 10 step 1
{counter = counter + 1;
counter2 = counter2 + 2;
}
```

The statement continues to loop as long as **LoopCounter** is less than or equal to **10**.

Example

```
for LoopCounter = 0 to -5 step -1
{counter = counter + 1;
```

```
counter2 = counter2 + 2;  
}
```

the **for** statement continues to loop as long as **LoopCounter** is greater than or equal to **-5**.

C-style for Statement

Description

The C-style **for** statement is used to execute a loop. The body which follows the **for** statement can be either a single statement or set of statements inside braces (**{** and **}**). This statement executes as follows:

1. The **start_expression** is evaluated.
2. The **test_expression** is evaluated.
3. If the **test_expression** is **false** (0), execution ends.
4. If the **test_expression** is **true** (non-zero), the body is executed.
5. If a **break** statement is encountered in the body, the loop terminates.
6. The **step_expression** is evaluated.

Repeat steps 2 through 6 until the **test_expression** is **false**, or a **break** statement is encountered.

If the **test_expression** is **false** the first time it is tested, then the step expression and body are not executed.

Syntax

```
for (start_expression; test_expression; step_expression ) body
```

Example

```
for (a=1; a <= 5; a+=1) print(a);
```

The statement prints the numbers from 1 to 5 until the test expression is **false**.

See Also

```
break, continue, do-while, for, for-in, while
```

for-in Statement

Description

The **for-in** statement is used to execute a loop for each element in a list. The body that follows the list can be either a single statement, or set of statements inside braces (**{** and **}**). This statement executes as follows:

1. A variable is set to the first or next element of the list.
2. The body executes. If a **break** statement is encountered in the body, the loop terminates.
3. Steps 1 and 2 are repeated while there are elements left in the list or until a **break** statement is encountered.

When the loop is complete, the **variable** contains the last value assigned to it.

Syntax

```
for variable in list body;
```

Example

```
for name in {"one", "two", "three"}  
print(name);
```

The statement prints each element in the list.

See Also

```
break, continue, do-while, for, while
```

if Statement

Description

The **if** statement is used to make a decision based on whether an expression evaluates to **true** or **false**. The decision determines what program statement is executed next. When **expression** evaluates to a non-zero value (**true**), the executable program statement immediately following the expression executes. When **expression** evaluates to **0 (false)**, the executable program statement immediately following the **else** statement is executed. When the chosen executable statement finishes, control flows to the next statement after the **if** statement. The **else** component of the **if** statement is optional.

Only one executable program statement can be inserted after the **if** expression or **else** statement. If multiple executable program statements are required, enclose them in curly braces **{}** to make a single compound statement.

Syntax

```
if (expression)  
executable program statement;  
else  
executable program statement;
```

Example

```
# Make an accept or reject decision based on  
# CurrentUserType  
if (CurrentUserType == 1)  
{  
    # if CurrentUserType is equal to 1, do these statements  
    RunCheck = true;  
    accept;  
}else  
{  
    # if CurrentUserType is not equal to 1, perform these statements:
```

```

RunCheck = false;
reject;
}
    
```

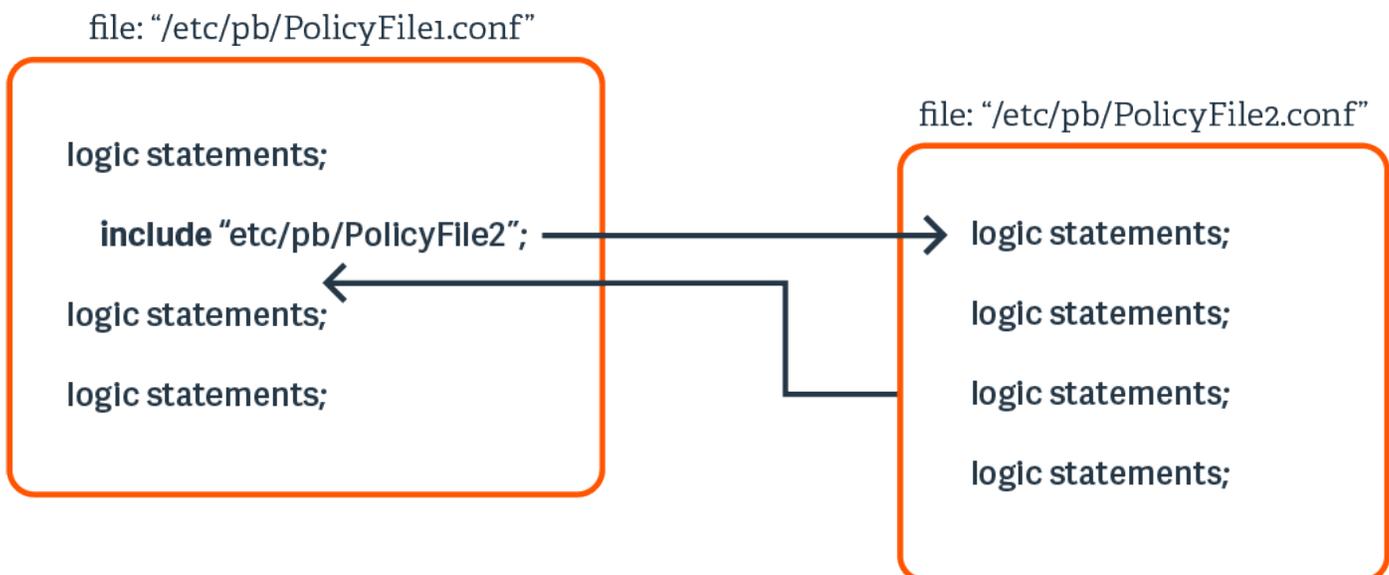
See Also

switch

include Statement

Description

The **include** statement is very powerful. It enables a security policy file to embed another security policy file called a security policy subfile. When an **include** statement is encountered, the flow of control jumps to the included file. When the included file has completed execution, the flow of control returns to the statement immediately following the **include** statement in the original file. The following figure demonstrates this concept.



When specifying **file-name**, the specified file name must be either a string enclosed in quotation marks or a variable that contains a string. If a relative or absolute path is not specified, Privilege Management for Unix & Linux looks for the file in the default security policy file directory. If a relative path name is specified, it is treated as relative to the security policy file directory that is specified in the **policydir** setting in **pb.settings**.

Syntax

```
include file-name;
```

where **file-name** can be a variable containing a string or a string constant enclosed in quotation marks.

Example

```
include "/opt/pbul/policies/SupportStaffPolicies.conf";  
include "/opt/pbul/policies/"+user+".conf";
```



Note: Use `stat()` to verify the existence of a file before adding an include statement that calls the file. Security policy subfile specifications that contain a variable may not be checked by `pbcheck` when checking the including file.

readonly Statement

Description

The **readonly** statement freezes a variable. After a variable is marked as read only, a security policy file cannot change its value. In essence, the variable ceases to behave as a variable and becomes a constant.

The **readonly** statement has a global scope.

Syntax

```
readonly { "variable1" [, "variable2", ...] };
```

Example

Do not allow changes to the following variables:

```
readonly { "CurrentUser", "CurrentCommand", "TargetHost" };
```

reject Statement

- **Version 4.0 and earlier:** reject statements do not support ACL.
- **Version 5.0 and later:** reject statements support ACL.

Description

The **reject** statement immediately terminates security policy file checking and cancels the current job request without allowing it to execute. Depending on the parameters that are selected, the user sees a default message, custom reject message, or no message.

In Privilege Management for Unix & Linux 5.0, the Privilege Management for Unix & Linux Policy Language was extended to include an **Access Control List** structure. This structure extends the **accept** statement to provide a simple non-programmatic way of entering access data.

Syntax

Version 4.0 and earlier:

```
reject ["reject-text"];
```

Version 5.0 and later:

```
reject ["reject-text"] [from ["user"][, ["submithost"]
[, ["command"][, ["runhost"]]]]]
[when conditional-expression];
```

- **reject-text** is the text to display to the user.
- **user** is a user name, list of user names, or left blank to imply any user.
- **submithost** is a submit host name, list of submit hosts, or left blank to imply any submit host.
- **command** is a command, list of commands, or left blank to imply any command.
- **runhost** is a run host, list of run hosts, or left blank to imply any run host.
- **conditional-expression** is an expression that evaluates **true** or **false**.

reject Statement Display Text

The **reject** statement has an optional **reject-text** expression in its argument. The meaning of the expression is as follows:

blank	Not specifying a parameter results in the display of the default <i>request rejected by Policy Server...</i> message.
""	An empty string suppresses the default <i>request rejected by Policy Server...</i> message.
"string"	Replaces the default <i>request rejected by Policy Server...</i> message with a message specified by string .

Examples
Version 4.0 and earlier:

```
if (user == "User1") reject;
```

If the current user is **User1**, reject the task request and immediately terminate security policy file processing.

```
reject;
```

The **reject** statement has no parameter, causing the default *request rejected by Policy Server...* message to appear.

```
reject "";
```

The **reject** statement used with the **null ("")** argument. This suppresses the default *request rejected by Policy Server...* message.

```
reject "You may not do that";
```

The **reject** statement is used with string parameter **"You may not do that"**, resulting in the message **"You may not do that"** being displayed.

Version 5.0 and later:

```
reject from "user4";
```

Reject all commands from **user4**, from any submit host, and for any run host.

```
reject when timebetween (1700, 900);
```

Reject all commands, from any user and any submit host, and for any run host, when the time is between 5:00 P.M. and 9:00 A.M.

```
reject "Permission denied" from {"user5", "user6"},,, "host5";
```

Reject all commands from **user5** or **user6**, from any submit host, for run host **host5**, with the display message *Permission denied*.

See Also

```
accept
```

switch Statement

Description

The **switch** statement provides a way to execute a specific set of program statements based on an expression value. Each set of program statements has a value associated with them. A **case** statement represents this value. If the **switch** statement expression matches a case statement, then the logic that is associated with that **case** statement executes.

When a switch expression-case statement match is found, execution begins at the statement immediately following the **case** statement. Execution continues through each statement following the **case** statement until a break statement is encountered. The **break** statement forces an immediate exit from the switch statement.

When a **break** statement is encountered, execution immediately jumps to the first statement following the end of the switch statement. The **break** statement is optional.

If an expression / **case** statement match is not found, the logic associated with the **default** case executes. The **default case** is optional.



Note: The case labels must evaluate as strings.

Syntax

```
switch (string-expression)
{
  case string1:
    statement1a; [statement1b; ...] [break;]
  case string2:
    statement2a; [statement2b; ...] [break;]
  default:
    default-stmt1; [default-stmt2; ...] [break;]
}
```

statement1a, **statement1b**, **statement2a**, **statement2b**, **default-stmt1**, and **defaultstmt2** all represent executable program statements.

Example

Check to see if the current user name is valid. Valid users are **admin** and **helpdesk**. If the user is not valid, reject the request.

```
switch (user)
{
  case "admin":
    hostmachine = "AdminHost"; break;
  case "helpdesk":
    hostmachine = "HelpDeskHost";break;
  default:
    reject;
}
```

See Also

[if](#)

while Statement

Description

The **while** statement is used to execute a loop. The body that follows the **while** statement can be a single statement or set of statements inside braces ({ and }). This statement executes as follows:

1. The **test_expression** is evaluated.
2. If the **test_expression** is **false** (0), the loop terminates.
3. If the **test_expression** is **true** (non-zero), the body executes.
4. If a **break** statement is encountered in the body, the loop terminates.

Repeat steps 1 through 4 until the **test_expression** is **false** or a **break** statement is encountered.

If the **test_expression** is **false** the first time it is tested, the body is not executed.

Syntax

```
while (test_expression) body
```

Example

```
a = 1;
while (a <= 10) {
  print(a);
  a += 1;
}
```

The statement prints the numbers 1 through 10 while a <=10.

See Also

```
break, continue, do-while, for, for-in
```

Non-Executable Program Statements

A Non-executable Program Statement helps organize security policy files. Because non-executable program statements have a special meaning to the security policy scripting language interpreter, they are not used for any other purpose. For instance, using a non-executable program statement as a variable name generates an error.

The non-executable program statement consists of the Comment statement.

Comment Statement

Description

Comment statements document the inner workings of individual security policy files. Comment text is nonexecutable code that is ignored by the interpreter during execution.

Comment statements must begin with the # character and continue to the end of the current line. No end character is necessary. This type of comment statement may not span multiple lines.

Syntax

```
# Comment text goes here.
```

Example

```
# This is a comment statement
```

Functions and Procedures

The Security Policy Scripting Language supports both **functions** and **procedures**. Functions and procedures are stand-alone subroutines that help modularize a company's security policy files. Functions and procedures are programming building blocks that execute specific tasks. These functions and procedures can be called whenever there is a need to perform that task. Functions and procedures are especially useful for repetitive type tasks.

The difference between functions and procedures is that functions return values while procedures do not.

Privilege Management for Unix & Linux functions and procedures do not support the same notion of scope as C functions. In other words, after a variable is implicitly defined, any function can use it. Its use is global and not limited to the function where it was originally defined.

If a variable is implicitly created in one function and referenced by another function, both functions can access and modify the same variable. The same holds true for procedures.

Privilege Management for Unix & Linux provides a number of built-in functions and procedures to help automate the process of creating security policy files.



For more information, please see "Built-in Functions and Procedures" on page 1

When adding user-written functions to a security policy file, the code for inline functions is placed at the top of the security policy file that first uses the function. Beginning with Privilege Management for Unix & Linux 3.0, **end** statements are no longer required for functions, procedures, and loops. However, Privilege Management for Unix & Linux still supports policy files that use end statements.



For more information on using user-written functions and procedures, please see "User and Password Functions" on page 1..

function Statement

Description

A function name can be any length. Its name can consist of any alpha or numeric characters, but it must start with an alphabetic character or an underscore.

The method of returning a value from a function is similar to that used in Pascal. The value is returned in a variable with the same name as the function.

A function must return a value. Otherwise, an error occurs.

Syntax

```
function FunctionName (argument-list)
{
statements;
FunctionName = expression;
}
```

Example

```
function square (x)
{
square = x * x;
}
```

See Also

procedure

procedure Statement

Description

A procedure name can be any length. It can consist of any alpha, underscore, or numeric characters, but it must start with an alphabetic character or an underscore.

Procedures do not return a value. If a value is returned, an error occurs.

Syntax

```
procedure ProcedureName (argument-list)
{
statements;
}
```

Example

```
procedure print_message(message)
{
print(message);
}
```

See Also

function

Other Programming Considerations

This section describes other programming considerations. These consist of:

- Boolean **true** and **false** variables
- Format commands
- Regular expression patterns
- Wildcard search characters
- Special characters

Boolean True and False Variables

Many program statements rely upon conditional tests to determine the next program statement to execute. The **if** program statement is an example.

Conditional tests generally evaluate to either a **true** or **false** value. Although any positive, non-zero integer can represent a **true** value, the integer **1** is normally used. The integer **0** represents a **false** value.

The following are some Boolean true and false variable examples:

- `LoopControl = false; #sets LoopControl to 0`
- `LoopControl = true; #sets LoopControl to 1`

Format Commands

Format commands insert values into character strings known as variable substitution. These commands specify where to insert the character string and how to format it. Format commands begin with a percent (%) sign followed by a format code. There are two categories of format commands: **Character** format and **Time** format.

Character Format Commands

The **printf()** function AND **fprintf** and **printf** procedures use character format commands. The following table describes the commands.

Character	Format Command
<code>%d</code>	Decimal value
<code>%i</code>	Integer value
<code>%o</code>	Octal value
<code>%s</code>	String of characters
<code>%u</code>	Unsigned decimal value
<code>%x</code>	Character hexadecimal value without a leading zero and with letters in lowercase (that is, 0x87a4)
<code>%X</code>	Character hexadecimal value without a leading zero and with letters in uppercase (that is, 0X87A4)
<code>%%</code>	Percent sign

Example

This demonstrates how character format commands work. Given the following character string,

```
I have x dogs, y cats, and z fish
```

The character format commands can be used to insert actual numeric values for x, y and z. This is done as follows:

```
printf ("I have %d dogs, %d cats, and %d fish", DogCount, CatCount, \FishCount);
```

DogCount, **CatCount** and **FishCount** are variables containing numeric values.

Example

The interpreter sequentially replaces each format command with one of the provided variables.

The replacement is done in sequential order. The first format command gets the first variable, and the second format command gets the second variable, etc.

Format commands can also use field modifiers to specify field width and whether to left justify a field.

Minimum Field-Width Modifier

An integer placed between the percent sign and the command character determines the minimum width of a field. By default, the pad character is a blank. To pad with zeros instead of spaces, place a zero before the minimum field-width specifier.

For example, **%04d** pads an integer value with zeros if the integer value is less than four digits in length.

Maximum Field-Width Modifier

A decimal point, followed by a maximum field width determines the maximum width of a field. If the value is longer than the specified maximum length, the value truncates on the right.

For example, **%2.4d** generates a field with a minimum length of two digits and a maximum length of four characters.

Left-Justification Field Modifier

By default, all output is right-justified. To left-justify a field, place a minus sign directly after the percent sign.

For example, **%-2.4d** generates a left-justified field with a minimum length of two digits and a maximum length of four digits.

Time Format Commands

The **strftime()** function uses time format commands. The following table describes the commands.



Note: Time format commands can vary based on the operating system. It is recommended that you consult the **strftime** manual pages for your local **pbrasterd** system.

Character	Command
%a	The abbreviated weekday name according to the current locale
%A	The full weekday name according to the current locale
%b	The abbreviated month name according to the current locale

%B	The full month name according to the current locale
%c	The preferred date and time representation for the current locale
%C	The century number (year/100) as a two-digit integer
%d	The day of the month as a decimal number (range 01 - 31)
%D	Equivalent to %m/%d/%y
%e	Like %d , the day of the month as a decimal number, but space replaces a leading zero
%E	Modifier. Use alternative format
%g	Like %G but without the century, (that is, with a 2-digit year, 00-99)
%G	The ISO 8601 year with century as a decimal number. The four-digit year that corresponds to the ISO week number (see %V). This has the same format as %y except that if the ISO week number belongs to the previous or next year, that year is used instead.
%h	Equivalent to %b
%H	The hour as a decimal number using a 24-hour clock (00-23)
%I	The hour as a decimal number using a 12-hour clock (01-12)
%j	The day of the year as a decimal number (001-366)
%k	The hour (24-hour clock) as a decimal number (0-23). A blank precedes single digits. See also %H .
%l	The hour (12-hour clock) as a decimal number (1-12). A blank precedes single digits. See also %I .
%m	The month as a decimal number (01-12)
%M	The minute as a decimal number (00-59)
%n	A new line character
%O	Modifier. Use alternative format
%p	Either AM or PM according to the given time value or the corresponding strings for the current locale. Noon is PM and midnight is AM.
%P	Like %p but in lowercase: am or pm or a corresponding string for the current locale
%r	The time in AM or PM notation
%R	The time in 24-hour notation (%H:%M). For a version that includes seconds, see %T .
%s	The number of seconds since the Epoch
%S	The second as a decimal number (00-61)
%t	A tab character
%T	The time in 24-hour notation (%H:%M:%S)
%u	The day of the week as a decimal (1-7) with Monday being 1

%U	The week number of the current year as a decimal number (00-53) starting with the first Sunday as the first day of week 01
%V	The ISO 8601:1998 week number of the current year as a decimal number (01-53) where week 1 is the first week that has at least four days in the current year and Monday as the first day of the week
%w	The day of the week as a decimal (0-6) with Sunday being 0
%W	The week number of the current year as a decimal number (00-53) starting with the first Monday as the first day of week 01
%x	The preferred date representation for the current locale without the time
%X	The preferred time representation for the current locale without the date
%y	The year as a decimal number without a century (00-99)
%Y	The year as a decimal number including the century
%z	The time zone as hour offset from GMT
%Z	The time zone name or abbreviation
%+	The date and time in date(1) format
%%	A % character

The time format commands work in the same manner as character format commands.

Regular Expression Patterns

The Privilege Management for Unix & Linux Security Policy Scripting Language supports extended regular pattern matching. Use these for pattern searches as well as forbidden and warning keystroke patterns.



For more information on regular expressions, please see the following:

- "grep" on page 1
- "egrep" on page 1

Pattern	Example	Description
.		Matches any character
	abc.d	Match the string abc followed by any single character then a d
[]		Defines the beginning and end of a character class
	[jJ]*	Match an uppercase or lowercase j followed by any number of characters
	[a-z]	Match any lowercase characters a through z
^		Not character (when used inside square brackets)
	[^a-z]	Match any character except lowercase characters a through z

*		Match zero or more occurrences of the last pattern
	abc*	Matches the string ab followed by zero or more c 's
?		Match zero or one occurrences of the last pattern
	abc?	Match either ab or abc
+		Match one or more occurrences of the last pattern
	abc+	Match the string ab followed by one or more c 's
{m}		Match exactly m occurrences of the last pattern
	abc{3}	Match the string abccc
{m,}		Match m or more occurrences of the last pattern
	abc{3,}	Match abccc , abccccc , etc
{m,n}		Match at least m , but no more than n , occurrences of the last pattern
	abc{3,5}	Match abccc , abccccc , or abccccc
()		Group several characters or patterns together and treat as a single group
	a(bc)+	Match abc , abcbc , abcbcbc , and so forth
		Match either of two patterns
	ab c	Match either ab or ac
^		Match beginning of line (when outside square brackets)
	^abc	Match abc only if it appears at the beginning of a line
\$		Match end of line
	abc\$	Match abc only if it appears at the end of a line
[:alnum:]		Matches alphanumeric characters
[:alpha:]		Matches alpha characters
[:blank:]		Matches spaces or tabs
[:boundary:]		Matches a word's boundaries
[:cntrl:]		Matches control characters
[:digit:]		Matches decimal digits
[:graph:]		Matches graphical characters
[:lower:]		Matches lowercase characters
[:print:]		Matches printable characters

<code>[:punct:]</code>	Matches punctuation marks
<code>[:space:]</code>	Matches any white space
<code>[:upper:]</code>	Matches uppercase characters
<code>[:xdigit:]</code>	Matches hexadecimal digits

Wildcard Search Characters

The Privilege Management for Unix & Linux Security Policy Scripting Language supports the standard set of shell-style, wildcard search characters. These are used for searches by the `in` operator and for forbidden and warning keystroke patterns.

Character	Example	Description
<code>*</code>		Matches any number of characters. Case is not considered.
	<code>j*</code>	Match <code>j</code> followed by any number of characters.
	<code>j*e</code>	Match a string starting with <code>j</code> and ending with <code>e</code> , with any number of characters between <code>j</code> and <code>e</code> .
<code>?</code>		Matches any single character. Case is not considered
	<code>j?</code>	Match <code>j</code> followed by any single character
	<code>j?e</code>	Match a string starting with <code>j</code> and ending with <code>e</code> , with any single character between <code>j</code> and <code>e</code>
<code>[]</code>		Match characters. Case is considered
	<code>[jJ]*</code>	Match upper or lowercase <code>j</code> followed by any number of characters
	<code>[a-z]</code>	Match any lowercase characters <code>a</code> through <code>z</code>
		Not character
	<code>[^a-z]</code>	Match any character except lowercase characters <code>a</code> through <code>z</code>

Special Characters

The security policy scripting language supports a standard set of special characters. Use special characters in place of characters that are impossible to enter using the keyboard or have other meanings in policy language strings. These characters can be used in the same way as any other single character, and they should be enclosed in either single or double quotation marks.

Character	Command
<code>\a</code>	Alert
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\r</code>	Carriage return

<code>\t</code>	Tab character
<code>'</code>	Single quotation mark
<code>"</code>	Double quotation mark
<code>\</code>	Backslash

Example

```
Tab = '\t';
```

This sets the variable with the **Tab** character.

```
StringExample = "start a new line \n";
```

This adds a new line character at the end of the string.

Privilege Management for Unix & Linux Variables

Privilege Management for Unix & Linux uses its own set of predefined variables to store information. These can be broken down into the following general categories:

- Task information variables
- Command line parsing variables
- Logging variables
- System variables
- Host identification variables
- X11 session capture variables

The Privilege Management for Unix & Linux variables are a valuable resource to security administrators because some of them can be queried from within security policy files. The information in Privilege Management for Unix & Linux variables can play a critical role in determining whether a specific request should be accepted or rejected. Privilege Management for Unix & Linux variables can also be used to set run time properties for a task request.

Task Information Variables

Privilege Management for Unix & Linux uses task information variables to store information about a specific task request. Using the security policy scripting language, a security administrator can query this information and use it to make security decisions about a task request. These values are logged in the event logs and I/O logs.



Note: The run variables do not apply to **pbssh**. If these run variables are present in the policy, they will not have any effect on **pbssh** and will be ignored.

The following table lists these variables.

Table 17. Task Information Variables

Task Information Variable	Run Version of Variable	Description
argc	---	Number of arguments that are supplied with the current command
argv	runargv	Argument values that are associated with the current command
bkgd	runbkgd	Controls if background command ignores HUP signals
browserhost	---	The host name of the machine that connected to pbguid .
clienthost	---	The name of the client (submit) host as resolved on the client host. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
command	runcommand	Name of the current command
cwd	runcwd	Full path of the current working directory
env	runenv	List of environment variables that are associated with the current task
group	rungroup	Name of user's primary group
groups	rungroups	List of all groups the current user belongs to
host	runhost	Name of the machine that the task will execute on
---	runhostip	IP address of the runhost
localmode	runlocalmode	Controls if the secured task replaces pbrun on the submit host, for local tasks. pblocald is not invoked. Note that with the exception of pbsh and pbksh , localmode is deprecated in favor of optimized run mode.
---	logcksum	Indicates which checksum value will be added to the event log.
mastertimelimit		Specifies a time limit, between pbmasterd and pblocald , for a task request. Version 4.0 and earlier: variable not available Version 4.0 and later: variable available
mastertimeout		Specifies the amount of idle time in seconds, between pbmasterd and

		pblocald . Version 4.0 and earlier: variable not available Version 4.0 and later: variable available
---	logservers	A list of log hosts for pblocald to use for event and I/O logging. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
nice	runnice	Nice values for the secured task
optimizedrunmode	runoptimizedrunmode	Controls whether optimized run mode is allowed for this task
---	pblocaldnoglob	Stops pblocald from expanding arguments to the target program
---	pbrisklevel	Risk rating that will be passed to BeyondInsight.
---	pidmessage	Optional message to issue when a job starts
requestuser	---	The user that is specified in the pbrun -u argument.
rlimit_as	runrlimit_as	Control the maximum memory that is available to a process. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_core	runrlimit_core	Control the maximum size of a core file. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_cpu	runrlimit_cpu	Control the maximum size CPU time of a process Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_data	runrlimit_data	Control the maximum size of a process' data segment Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_fsize	runrlimit_fsize	Control the maximum size of a file Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_locks	runrlimit_locks	Control the maximum number of file locks for a process Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_memlock	runrlimit_memlock	Control the maximum number of bytes of virtual memory that can be locked Version 3.5 and earlier: variable not available

		Version 4.0 and later: variable available
rlimit_nofile	runlimit_nofile	Control the maximum number of files a user may have open at a given time. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_nproc	runlimit_nproc	Control the maximum number of process a user may run at a given time Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_rss	runlimit_rss	Control the maximum size of a process' resident set (number of virtual pages resident at a given time) Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
rlimit_stack	runlimit_stack	Control the maximum size of the process stack Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
selinux		Indicates whether pbrun is confined by SELinux. Version 5.2 and earlier: variable not available Version 6.0 and later: variable available
---	runchroot	Name of the special file system root directory; see the chroot manual page for more information
---	runcksum	Contains a checksum value for the current task
---	runcksumlist	Contains a list of checksum values for the current task
---	runconfirmmessage	Password prompt that is used by pblocald for a final verification of the user
---	runconfirmuser	Controls if final verification requires a password
---	runeffectivegroup	Controls the effective group ID (egid) of the requested job
---	runeffectiveuser	Controls the effective user ID (euid) of the requested job
---	runenablerlimits	When true , use the runlimit_* variables to set up ulimits for the secured task. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	runenvironmentfile	Specifies an environment file that contains environment variables to be incorporated into the run environment Version 5.2 and earlier: variable not available Version 6.0 and later: variable available

---	runptyflags	Flags that are used internally for pty settings; reserved for internal use
---	runsecurecommand	Check that the runcommand is writable only by root or the runuser. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	runmd5sum	Contains an MD5 checksum for the current task.
---	runmd5sumlist	Contains a list of MD5 checksum values for the current task.
---	runtimelimit	The number of seconds that the job may execute
---	runtimeout	Maximum allowed idle time
---	runutmpuser	utmp user name
---	shellallowedcommands	Contains a list of strings that contain commands that may be run without any further authorization Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	shellcheckbuiltins	If true , directs the shell to check shell built-in commands as if they were standard commands Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	shellcheckredirections	If true , directs the shell to authorize I/O redirections; if false , always allows I/O redirection. Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	shellforbiddencommands	Contains a list of strings that specify commands for pbksh and pbsh to reject without consulting a Privilege Management for Unix & Linux policy server daemon Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	shellloginincludefiles	Controls if the contents of included (sourced) shell scripts should be recorded in the I/O logs Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
---	shellreadonly	Contains a list of environment variables that pbsh and pbksh set to read-only at startup time Version 3.5 and earlier: variable not available Version 4.0 and later: variable available

	shellrestricted	Controls if Privilege Management for Unix & Linux shells run in restricted mode Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
solarisproject	runsolarisproject	Specifies a Solaris project that the secured task should be associated with on a Solaris 9 or higher runhost Version 6.0 and earlier: variable not available Version 6.1 and later: variable available
submithost	---	Name of the machine from which the current request was submitted
submithostip	---	IP address of the machine from which the current request was submitted
taskpid	---	The PID of the secured task launched by pbrun
taskttyname	---	Name of the tty device associated with the secured task. This variable is only available after the secured task is launched and cannot be used in the policy. This is a read-only variable. Version 6.2.0 and earlier: variable available Version 6.2.6 and later: variable available
timezone	---	Standard representation of timezone on submithost
ttyname	---	Name of the tty device from which the current request was submitted
umask	runumask	The user's umask values
user	runuser	Specifies the user ID that is associated with the login name of the user that submitted the current task.

Within Privilege Management for Unix & Linux, each secured task has its set of task information variables. Other secured task requests do not share the information in these variables.

Two copies of task information variables are created and maintained for each task request that Privilege Management for Unix & Linux processes. One set is read-only. These read-only variables contain the original, unmodified information about a task request. The other set, known as run variables, have the exact same information as their corresponding read-only versions; however, their values can be modified. The information in the modifiable variables is the information that Privilege Management for Unix & Linux actually uses to execute a request once it is accepted. The modifiable task information variables have the same names as their read-only counterparts except they have the prefix **run**.



Note: These run variables do not apply to **pbssh**. If these run variables are present in the policy, they will not have any effect on **pbssh** and will be ignored.

There are some special pass-through values that are available for the run versions of some task information variables. These special values are needed when the policy server host and run host represent different systems. In this scenario, processing some functions may fail because the values for those variables need to be retrieved from the run host system rather than the policy server host. The following functions are affected: **gethome()**, **getgroup()**, **getgroups()**, and **getshell()**.

Value	Description	Example
-------	-------------	---------

!g!	Return the run user's run group on run host.	<code>rungroup = "!g!";</code>
!G!	Return all groups that the run user belongs to on run host	<code>rungroups = {"!G!"};</code>
!~!	Return the run user's home directory on run host	<code>runcwd = "!~!";</code>
!!!	Return the run user's default shell on run host	<code>runcommand = "!!!";</code>

 For more information on when and how to use special run variable values, please see "[Environment Variable Processing Considerations](#)" on page 39.

 For more information on the `gethome()`, `getgroup()`, `getgroups()`, and `getshell()` functions, please see "[Built-in Functions and Procedures](#)" on page 211.

argc

Data Type

Integer, read-only

Description

The **argc** variable contains the number of arguments that are supplied with the current command. The command name is treated as an argument. Thus, the actual number of user supplied arguments, not including the command name itself, is **argc - 1**.

There is not a run version of this variable.

Valid Values

A positive integer

See Also

`argv`, `runargv`, `command`, `runcommand`

argv

Run Version

runargv



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

List. **argv** is read-only. **runargv** is modifiable.

Description

The **argv** and **runargv** variables contain the list of argument values that are associated with the current command. The first argument value, with index **0**, is the name of the command. Use the run version of this variable to change an argument value.

Syntax

```
runargv = list;
```

Valid Values

A list in which the first element contains the name of the current command, as entered by the submitting user. The remaining list elements contain the command arguments, as entered by the submitting user. **argv** is a read-only variable whose value comes from the **pbrun** command line. The default value of **runargv** is the value of **argv**.

Example

```
runargv = {"uname", "-a"};
```

See Also

```
argc, command, runcommand
```

bkgd

Run Version

runbkgd



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Boolean. **bkgd** is read-only. **runbkgd** is modifiable.

Description

The **bkgd** and **runbkgd** variables indicate whether to run a task in the background with HUP signals ignored. Privilege Management for Unix & Linux sets both variables when the user executes **pbrun** with a **-b** switch. To change whether a task actually runs in the background with HUP signals ignored, set the **runbkgd** variable.



Tip: In this context, the function name inside the function behaves like a function parameter.

When its parent process terminates, HUP refers to the Hang-Up signal that is sent to a child process by the operating system. If the child process was set to ignore HUP signals, the child process continues to run even though its parent process was terminated.



Tip: This feature can be useful for applications running in the background.

Syntax

```
runbkgd = boolean;
```

Valid Values

true	Ignore HUP signals
false	Do not ignore HUP signals

runbkgd is read-only and defaults to **true** when **pbrun -b** is used. Otherwise, it defaults to **false**. **runbkgd** defaults to the value of **bkgd**.

Example

```
runbkgd = true;
```

browserhost

Data Type

String, read-only

Description

The host name of the machine connected to **pbguid**. This is usually a browser or a proxy.

Valid Values

A string as described above

See Also

```
browserip
```

browserip

Data Type

String, read-only

Description

The IP address of the machine connected to **pbguid**. This is usually a browser or a proxy.

Valid Values

A string as described above

See Also

```
browerhost
```

clienthost

- **Version 3.5 and earlier:** **clienthost** variable is not available.
- **Version 4.0 and later:** **clienthost** variable is available.

Data Type

String, read-only

Description

The name of the client (submit) host as resolved on the client host.

Valid Values

A string as described above

See Also

```
host, submithost
```

command

Run Version

runcommand



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

String. **command** is read-only. **runcommand** is modifiable.

Description

The **command** and **runcommand** variables contain the name of the current command request. If specified, command arguments are stored in **runargv** and are not stored in **command** or **runcommand**. To change the current command, set the **runcommand** variable.



Note: Setting the run version of this variable also sets **runargv[0]**; however, setting **runargv** does not set **runcommand**.

Syntax

```
runcommand = string;
```

Valid Values

A string containing the name of the current task request command as entered by the submitting user. **command** is a read-only variable. **runcommand** defaults to the value of **command**.

Example

```
runcommand = "/bin/ls";
```

See Also

```
argc, argv, runargv
```

cwd

Run Version

runcwd



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

String. **cwd** is read-only. **runcwd** is modifiable.

Description

The **cwd** and **runcwd** variables contain the full path of the working directory on the submit host from which the current task request is being initiated. To cause the requested program to execute in a different directory on a run host, set the **runcwd** variable. Depending on how Privilege Management for Unix & Linux was deployed, submit host and run host might be different machines with different directory structures.



Note: If Privilege Management for Unix & Linux cannot set this variable and **enforceRunCwd** is set to **No**, the task request will run in the **/tmp** directory on the run host.

Syntax

```
runcwd = string;
```

Valid Values

A string specifying the run host working directory for the current task request. **cwd** is a read-only variable. Also, **cwd** is the directory from which the command originated. **runcwd** defaults to **cwd**.

Example

```
runcwd = "/home/username";
```

See Also

```
runchroot
```

env

Run Version

runenv



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

List. **env** is read-only. **runenv** is modifiable.

Description

The **env** and **runenv** variables contain the name and value pairs of each Unix or Linux environment variable that is present when the current task request was submitted. Each environment variable is stored as an element within **env**. Each of these elements has the format **NAME=Value**, where **NAME** is the name of the environment variable and **Value** is the value that is stored in that variable.

The value of an environment variable is modified by setting **runenv**.

The **getenv()**, **setenv**, **keepenv**, and **unsetenv** functions and procedures can access the values within **env**.



For more information on these functions, please see "[Task Environment Functions and Procedures](#)" on page 289.

Syntax

```
runenv = list of strings;
```

Valid Values

A list in which each element has the format **NAME=value** where **NAME** is the name of the Unix or Linux environment variable and **value** is the value stored in that variable. This list defaults to the run time environment of the **pbrun** command.

See Also

```
getenv( ), keepenv, logomit, setenv, unsetenv
```

execute_via_su

Data Type

Boolean

Description

The run environment for the secured task is normally dictated by the Privilege Management for Unix & Linux Policy Server policy. It may be desirable to have the runhost dictate the run environment for the secured task. Privilege Management for Unix & Linux version 7.1 and above can use the `su -` command to create a login shell for the secured task, thus allowing the login mechanism to setup the run environment. The Privilege Management for Unix & Linux Policy Server host keyword **execute_via_su** in `/etc/pb.settings` will globally enable using `su -` to execute the secured task. This keyword can be overridden by the policy variable with the same name **execute_via_su**. The **execute_via_su** variable's initial value is based on the keyword setting's value. When **execute_via_su** is used, any run environment setup in the policy will affect the execution of `su -` rather than the execution of the secured task. This includes the use of `runcwd`, `setenv()`, `keepenv()`, etc as well as `!g!`, `!G!`, etc. Entitlement reports will not indicate that `su -` is used, however the Accept events in the event log will show that `su -` was used to invoke the secured task. This feature will not work for runusers whose login is disabled (for example, using `/sbin/nologin` or `/bin/false`).

Settings Keyword	Policy Variable	Result uses su -?
unset	unset	no
	TRUE	YES
	FALSE	no
No	unset	no
	TRUE	YES
	FALSE	no
Yes	unset	YES
	TRUE	YES
	FALSE	no

Valid Values

- 0
- 1
- true
- false

Default:

unset

See Also

```
runcommand, runuser, runargv, runenvironmentfile, setenv(), keepenv()
```



For more information, please see "[Environment Variable Processing Considerations](#)" on page 39.

group

Run Version

rungroup



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

String. **group** is read-only. **rungroup** is modifiable.

Description

The **group** and **rungroup** variables contain the name of the submitting user's primary group. To temporarily change the submitting user's primary group, set the **rungroup** variable.



Note: If the **rungroup** does not exist on the run host, the run host will refuse to execute the command.

Syntax

```
rungroup = string;
```

Valid Values

A string that contains the name of the submitting user's primary group. **group** is a read-only variable. The default value of **rungroup** defaults to the value of **group**.

Example

```
rungroup = "bin";
```

See Also

```
groups, rungroups, getgroup(), getgrouppasswd(), getgroups(), innetgroup(), inusernetgroup(),  
runffectivegroup
```

groups

Run Version

rungroups



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

List. **groups** is read-only. **rungroups** is modifiable.

Description

The **groups** and **rungroups** variables contain the list of groups the submitting user belongs to. To temporarily modify the list of groups, set the **rungroups** variable.

If one of the **rungroups** does not exist on the run host, the run host issues a warning before executing the command.

Syntax

```
rungroups = list;
```

Valid Values

The **groups** variable contains the name of each group the submitting user belongs to on the submit host.

The value of the **rungroups** variable defaults to the value of the **groups** variable.

Example

```
rungroups = {"bin", "wheel"};
```

See Also

```
group, rungroup, getgroup(), getgrouppasswd(), getgroups(), innetgroup(), inusernetgroup()
```

host

Run Version

runhost



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

String. **host** is read-only. **runhost** is modifiable.

Description

submithost is the name of the machine that executed **pbrun**. **host** is the value that is passed to **pbrun** with the **-h** switch. If a **-h** switch is not used, then the value of **host** is taken from **submithost**. If the value of **runhost** is not explicitly set in the policy, then its value comes from **host**.

Setting **runhost** in the policy has no effect when the task is run in local mode (that is, when **pbrun** is executed with the **-l** option, or if the **runlocalmode** policy variable is set to **true**).

Syntax

```
runhost = string;
```

Valid Values

A string that contains the fully-qualified name of the run host machine. **host** is a read-only default value and is the name of the submit host. The default value of **runhost** is the value of **host**.

Example

```
runhost = "tad";
```

See Also

```
ipaddress(), localmode, runlocalmode, masterhost, pid, requestuser, runconfirmuser, subprocuser, submithost, submithostip, uniqueid
```

localmode

Run Version

runlocalmode



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Boolean. **localmode** is read-only. **runlocalmode** is modifiable.

Description

The **localmode** and **runlocalmode** variables indicate if the submitting user specified that the current task request run in local mode. When a task runs in local mode, **pbmasterd** returns control to **pbrun** rather than **pblocald**. After the task is accepted, **pbrun** replaces itself with the current task request. The result is that **localmode** cannot be used with Advanced Control and Audit (ACA), and the current task request is processed without the benefit of any further event logging (the exit status is not logged) or keystroke actions.

Regarding **pbrun**, the **localmode** mechanism is deprecated in favor of Optimized Run Mode, where all features are available.

The Privilege Management shells **pbsh** and **pbksh** normally operate in **localmode**. This can be disabled by setting **runlocalmode=false**.

Privilege Management for Unix & Linux sets the **localmode** variables when the user executes **pbrun** with a **-l** switch, or when the **runlocalmode** variable is set to **true** in the policy.

Syntax

```
runlocalmode = boolean;
```

Valid Values

true	Run local mode. The default value is true if pbrun -l is used, false otherwise.
false	Disable local mode.

localmode is a read-only variable with a value of **true** if **pbrun -l** is used, **false** otherwise.

runlocalmode defaults to **localmode**. If the **allowlocalmode** setting is **false**, then **runlocalmode** is set to read-only and has a value of **false**.

Example

```
runlocalmode = false;
```

See Also

```
bkgd, runbkgd, noreconnect, pblocald, pbrun, allowlocalmode
```

logcksum

- **Version 7.5 and earlier:** **logcksum** variable not available
- **Version 8.0 and later:** **logcksum** variable available

Data Type

String, modifiable

Description

When **runcksum**, **runcksumlist**, **runmd5sum**, or **runmd5sumlist** are present in the policy, the run host verifies that the checksum of the **runcommand** matches the values specified in those variables. The **logcksum** variable allows the checksum of the **runcommand** to be recorded in the event log for analysis.

There is no read-only version of this variable.

Syntax

```
logcksum = string_value
```

Valid Values

cksum	Save the runtime-generated application checksum in the cksum variable and record it in the event log. This is the value that would be compared to the runcksum or runcksumlist user-defined policy variable (if available).
md5	Save the runtime-generated application MD5 checksum in the md5sum variable and record it in the event log. This is the value that would be compared to the runmd5sum or runmd5sumlist user-defined policy variable (if available).
all	Record both runtime-generated checksum values (cksum and md5sum variables) in the event log.

Example

```
logcksum = "cksum";
```

```
logcksum = "md5";
```

```
logcksum = "all";
```

See Also

```
runcksum, runcksumlist, runmd5sum, runmd5sumlist
```

logpid

Data Type

Number, read-only

Description

The **logpid** variable contains the PID of the logserver daemon logging the accept.

This read-only variable is not available during the processing of the policy, because it is created after the policy performs an accept. This variable is available in the event log.

There is no run version of this variable.

Valid Values

A number that contains a PID.

This is a read-only variable.

See Also

```
pid, runpid, submitpid, taskpid
```

logservers

Data Type

List

Description

A list of log hosts for **pblocald** to use for event and I/O logging. The policy variable overrides the settings keyword when the **logservers** keyword in the settings file is enabled. In other words,

```
/etc/pb.settings:  
.  
.  
logservers name0  
/opt/pbul/policies/pb.conf:  
...logservers={"name1", "name2"};  
...
```

The logservers that are used are **name1** and **name2**.

Syntax

```
logservers = {list};
```

Example

```
logservers = {"name1", "name2"};
```

mastertimelimit

- **Version 4.0 and earlier:** **mastertimelimit** variable not available
- **Version 5.0.1 and later:** **mastertimelimit** variable available

Data Type

Integer, modifiable

Description

The **mastertimelimit** variable specifies a time limit, in seconds, between **pbmasterd** and **pblocald**, for a task request. If the job does not finish within the specified number of seconds, then it is terminated.

mastertimelimit is similar to **mastertimeout**, but it is based on total time rather than idle time.

mastertimelimit is similar to **runtime-limit**, from the **pbmasterd** point of view, and is useful only when there is no log server.



Note: The **mastertimelimit** variable is not honored in local mode.

Syntax

```
mastertimelimit = number;
```

Valid Values

- **number:** Enable time limit checking.
- **0:** Disable time limit checking. This value is the default.

Example

```
mastertimelimit = 3600;
```

See Also

```
mastertimeout, runtime-limit, runtimeout, submittimeout
```

mastertimeout

- **Version 4.0 and earlier:** **mastertimeout** variable not available
- **Version 5.0.1 and later:** **mastertimeout** variable available

Data Type

Integer, modifiable

Description

The **mastertimeout** variable specifies the amount of idle time, in seconds, between **pbmasterd** and **pblocald**. If the job is idle for the specified number of seconds, then it is terminated. **mastertimeout** is similar to **runtimeout**, from the **pbmasterd** point of view, and is useful only when there is no log server.



Note: The **mastertimeout** variable is not honored in local mode.

Syntax

```
mastertimeout = number;
```

Valid Values

- **number**: Enable idle checking.
- **0**: Disable idle checking. This value is the default.

Example

```
runtimeout = 3600;
```

See Also

```
mastertimelimit, runtimelimit, runtimeout, submittimeout
```

nice

Run Version

runnice



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Integer. **nice** is read-only. **runnice** is modifiable.

Description

The **nice** and **runnice** variables contain the **nice** value for the current task request. The **nice** value controls task execution priority. To modify task execution priority, set **runnice**.



For more information on **nice** values, refer to the Unix or Linux manual pages.

Syntax

```
runnice = number;
```

Valid Values

An integer value that represents a task execution priority. This variable has no default value.

Example

```
runnice = 20;
```

See Also

Unix or Linux manual page for the **nice** command

noexec

Data Type

Integer. **noexec** is modifiable.

Description

This variable does not apply to **pbssh**. If it is present in the policy, and set to **1**, **pbrun**, **pblocald**, **pbsh**, and **pbksh** will attempt to prevent the secured task from performing an exec to launch a new program (for example, prevent vi's shell escape `!/bin/bash`).

This mechanism uses the **LD_PRELOAD** or equivalent mechanism to load a Privilege Management for Unix & Linux shared library that intercepts the exec family of library calls.

The **noexec** feature requires Privilege Management for Unix & Linux 8.5.0 **runhosts**. Any previous version of **runhost** will silently ignore the **noexec** feature.



Note: Care should be used when assigning enabling **noexec** for shell scripts (these normally exec other programs).

Restrictions

- The **noexec** feature is not supported on Mac OS X systems.
- The **noexec** feature works only for binaries that are dynamically linked, on operating systems that support the **LD_PRELOAD** or equivalent mechanism.
- The **noexec** feature supports **setuid** programs only on Linux and Solaris run hosts.
- The **noexec** feature cannot execute shell scripts that lack the **#!/path/shell** specification.
- The **noexec** feature currently does not support the Privilege Management for Unix & Linux **execute_via_su** feature.
- HP-UX 11.11 requires linker patch PHSS_22535 or newer.

Syntax

```
noexec=1;
```

Valid Values

Valid values are **0** and **1**. This variable has default value of **0**.

Example

```
noexec=1;
```

See Also

Unix/Linux manual page for the **ld.so** (Linux), **ld.so.1** (Solaris), **ld** (HP-UX), **dld.sl** (HP-UX) commands

optimizedrunmode

- **Version 4.0 and earlier:** **optimizedrunmode** variable not available
- **Version 5.0 and later:** **optimizedrunmode** variable available
- **Version 6.0 and later:** **runoptimizedrunmode** variable available

Run Version

runoptimizedrunmode



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Boolean. **optimizedrunmode** is read-only. **runoptimizedrunmode** is modifiable.

Description

optimizedrunmode indicates whether the task can be executed using Privilege Management for Unix & Linux's optimized run mode feature. A value of **true** indicates that optimized run mode has not been disabled for this task by command line switch or Privilege Management for Unix & Linux settings.

Setting **runoptimizedrunmode** to **false** can be used to prevent a task from being executed using Privilege Management for Unix & Linux's optimized run mode feature. Note that if optimized run mode was disabled in the Policy Server host's settings file, the submit host's settings file, or by a command line option on either **pbrun** or **pbmastord**, then setting **runoptimizedrunmode** to **true** will have no effect.

Syntax

```
runoptimizedrunmode = Boolean;
```

Valid Values

true	Non-zero. Enable optimized run mode.
false	Zero. Disable optimized run mode

Example

```
runoptimizedrunmode = false;
```

See Also



For information about optimized run mode and related settings, please see the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

pblocaldnoglob

Data Type

Boolean, modifiable

Description

pblocaldnoglob stops **pblocald** from expanding arguments to the target program. By setting this variable to a non-zero value, you can duplicate the way version Privilege Management for Unix & Linux 2.6 (and earlier) passed arguments.

There is no read-only version of this variable.

Syntax

```
pblocaldnoglob = boolean;
```

Valid Values

true	Non-zero. Stop pblocald from expanding arguments to the target program
false	Zero. Allow pblocald to expand arguments to the target program. This setting is the default.

Example

```
pblocaldnoglob = true;
```

pbrisklevel

Data Type

Number, modifiable

Description

The **pbrisklevel** variable specifies a risk rating that will be passed to BeyondInsight. The data will be displayed in the BeyondInsight Privilege Management for Unix & Linux grid and agent details grid.

There is no read-only version of this variable.

Syntax

```
pbrisklevel = number;
```

Valid Values

- A whole number in the range of 0 - 9
 - 9 means highest risk
 - 0 means no risk

Default Value

If **pbrisklevel** is not explicitly set in the policy, the risk level setting will default to zero (0).

Example

```
pbrisklevel = 3;
```

pidmessage

Data Type

String, modifiable

Description

The **pidmessage** variable contains an optional string that causes the process ID of the task on the run host to print out at the start of the task.

There is no read-only version of this variable.



Note: If Privilege Management for Unix & Linux is running as local mode, it ignores **pidmessage**.

Syntax

```
pidmessage = string;
```

Valid Values

Any string. The default value is empty.

Example

The following example produces something similar to *This is job: sparky 9876* before the target command runs.

```
pidmessage = "This is job: ";
```

requestuser

Data Type

String, read-only

Description

The **requestuser** variable contains the value that is specified by the **pbrun -u** argument. When a user runs **pbrun** with the **-u** username option, the value is placed in **requestuser**. The policy then determines whether or not to honor the request. If the **-u** command option is not used, then **requestuser** contains the same value as user.

There is no run version of this variable.

Valid Values

A string as described above

See Also

```
pbrun, user, runuser
```

rlimit_as

- **Version 3.5 and earlier:** **rlimit_as** and **runrlimit_as** variables not available
- **Version 4.0 and later:** **rlimit_as** and **runrlimit_as** variables available

Run Version

runrlimit_as



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **rlimit_as** is read-only, **runrlimit_as** is modifiable.

Description

These variables control the maximum memory available to a process in bytes as a 32-bit number. These variables are equivalent to **vmem** on some systems. **rlimit_as** is the read-only value for the user who invoked Privilege Management for Unix & Linux.

runlimit_as is the modifiable value for the target secured task.



Note: To enable **runlimit_as** functionality, set **runenablerlimits** to a value of 1.

Syntax

```
runlimit_as = number;
```

Valid Values

Vary according to platform

Example

```
runlimit_as = 1000;
```

See Also

```
rlimit_core, runlimit_core, rlimit_cpu, runlimit_cpu, rlimit_data, runlimit_data, rlimit_fsize, runlimit_fsize, rlimit_locks, runlimit_locks, rlimit_memlock, runlimit_memlock, rlimit_nofile, runlimit_nofile, rlimit_nproc, runlimit_nproc, rlimit_rss, runlimit_rss, rlimit_stack, runlimit_stack
```

rlimit_core

- **Version 3.5 and earlier:** **rlimit_core** and **runlimit_core** variables not available
- **Version 4.0 and later:** **rlimit_core** and **runlimit_core** variables available

Run Version

runlimit_core



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **rlimit_core** is read-only. **runlimit_core** is modifiable.

Description

These variables control the maximum size of a core file in bytes as a 32-bit number. **rlimit_core** is the read-only value for the user who invoked Privilege Management for Unix & Linux. **runlimit_core** is the modifiable value for the target secured task.



Note: To enable **runlimit_core** functionality, set **runenablerlimits** to a value of 1.

Syntax

```
runrlimit_core = number;
```

Valid Values

Vary according to platform

Example

```
runrlimitcore = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_cpu

- **Version 3.5 and earlier:** `rlimit_cpu` and `runrlimit_cpu` variables not available
- **Version 4.0 and later:** `rlimit_cpu` and `runrlimit_cpu` variables available

Run Version

runrlimit_cpu



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. `rlimit_cpu` is read-only. `runrlimit_cpu` is modifiable.

Description

These variables control the maximum size CPU time of a process in seconds as a 32-bit number. `rlimit_cp` is the read-only value for the user who invoked Privilege Management for Unix & Linux. `runrlimit_cpu` is the modifiable value for the target secured task.



Note: To enable `runrlimit_cpu` functionality, set `runenablerlimits` to a value of 1.

Syntax

```
runlimit_cpu = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_cpu = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimitdata, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_data

- **Version 3.5 and earlier:** **rlimit_data** and **runrlimit_data** variables not available
- **Version 4.0 and later:** **rlimit_data** and **runrlimit_data** variables available

Run Version

runrlimit_data



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **rlimit_data** is read-only. **runrlimit_data** is modifiable.

Description

These variables control the maximum size of a process' data segment as a 32-bit number. **rlimit_data** is the read-only value for the user who invoked Privilege Management for Unix & Linux. **runrlimit_data** is the modifiable value for the target secured task.



Note: To enable **runrlimit_data** functionality, set **runenablerlimits** to a value of 1.

Syntax

```
runrlimit_data = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_data = 100;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_fsize

- **Version 3.5 and earlier:** **rlimit_fsize** and **runrlimit_fsize** variables not available
- **Version 4.0 and later:** **rlimit_fsize** and **runrlimit_fsize** variables available

Run Version

runrlimit_fsize



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **rlimit_fsize** is read-only. **runrlimit_fsize** is modifiable.

Description

These variables control the maximum size of a file in bytes as a 32-bit number. **rlimit_fsize** is the read-only value for the user who invoked Privilege Management for Unix & Linux. **runrlimit_fsize** is the modifiable value for the target secured task.



Note: To enable **runrlimit_fsize** functionality, set **runenablerlimits** to a value of **1**.

Syntax

```
runrlimit_fsize = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_fsize = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_locks

- **Version 3.5 and earlier:** **rlimit_locks** and **runrlimit_locks** variables not available
- **Version 4.0 and later:** **rlimit_locks** and **runrlimit_locks** variables available

Run Version

runrlimit_locks



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **rlimit_locks** is read-only. **runrlimit_locks** is modifiable.

Description

These variables control the maximum number of file locks for a process as a 32-bit number. **rlimit_locks** is the read-only value for the user who invoked Privilege Management for Unix & Linux. **runrlimit_locks** is the modifiable value for the target secured task.



Note: To enable **runrlimit_locks** functionality, set **runenablerlimits** to a value of 1.

Syntax

```
runrlimit_locks = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_locks = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_memlock

- **Version 3.5 and earlier:** `rlimit_memlock` and `runrlimit_memlock` variables not available
- **Version 4.0 and later:** `rlimit_memlock` and `runrlimit_memlock` variables available

Run Version

runrlimit_memlock



Note: This run variable does not apply to `pbssh`. If it is present in the policy, it will not have any effect on `pbssh` and will be ignored.

Data Type

Number. `rlimit_memlock` is read-only. `runrlimit_memlock` is modifiable.

Description

These variables control the maximum number of bytes of virtual memory that may be locked at a given time as a 32-bit number. `rlimit_memlock` is the read-only value for the user who invoked Privilege Management for Unix & Linux. `runrlimit_memlock` is the modifiable value for the target secured task.



Note: To enable `runrlimit_memlock` functionality, set `runenablerlimits` to a value of 1.

Syntax

```
runrlimit_memlock = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_memlock = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_nofile

- **Version 3.5 and earlier:** `rlimit_nofile` and `runrlimit_nofile` variables not available
- **Version 4.0 and later:** `rlimit_nofile` and `runrlimit_nofile` variables available

Run Version

runrlimit_nofile



Note: This run variable does not apply to `pbssh`. If it is present in the policy, it will not have any effect on `pbssh` and will be ignored.

Data Type

Number. `rlimit_nofile` is read-only. `runrlimit_nofile` is modifiable.

Description

These variables control the maximum number of files a user may have open at a given time as a 32-bit number. `rlimit_nofile` is the read-only value for the user who invoked Privilege Management for Unix & Linux. `runrlimit_nofile` is the modifiable value for the target secured task.



Note: To enable `runrlimit_nofile` functionality, set `runenablerlimits` to a value of 1.

Syntax

```
runrlimit_nofile = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_nofile = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_nproc

- **Version 3.5 and earlier:** `rlimit_nproc` and `runrlimit_nproc` variables not available
- **Version 4.0 and later:** `rlimit_nproc` and `runrlimit_nproc` variables available

Run Version

runrlimit_nproc



Note: This run variable does not apply to `pbssh`. If it is present in the policy, it will not have any effect on `pbssh` and will be ignored.

Data Type

Number. `rlimit_nproc` is read-only. `runrlimit_nproc` is modifiable.

Description

These variables control the maximum number of process a user may run at a given time as a 32-bit number. `rlimit_nproc` is the read-only value for the user who invoked Privilege Management for Unix & Linux. `runrlimit_nproc` is the modifiable value for the target secured task.



Note: To enable `runrlimit_nproc` functionality, set `runenablerlimits` to a value of 1.

Syntax

```
runrlimit_nproc = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_nproc = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_rss, runrlimit_rss, rlimit_stack, runrlimit_stack
```

rlimit_rss

- **Version 3.5 and earlier:** **rlimit_rss** and **runrlimit_rss** variables not available
- **Version 4.0 and later:** **rlimit_rss** and **runrlimit_rss** variables available

Run Version

runrlimit_rss



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **rlimit_rss** is read-only. **runrlimit_rss** is modifiable.

Description

These variables control the maximum size of a process' resident set (number of virtual pages that are resident at a given time) as a 32-bit number. **rlimit_rss** is the read-only value for the user who invoked Privilege Management for Unix & Linux. **runrlimit_rss** is the modifiable value for the target secured task.



Note: To enable **runrlimit_rss** functionality, set **runenablerlimits** to a value of **1**.

Syntax

```
runrlimit_rss = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_rss = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_stack, runrlimit_stack
```

rlimit_stack

- **Version 3.5 and earlier:** `rlimit_stack` and `runrlimit_stack` variables not available
- **Version 4.0 and later:** `rlimit_stack` and `runrlimit_stack` variables available

Run Version

runrlimit_stack



Note: This run variable does not apply to `pbssh`. If it is present in the policy, it will not have any effect on `pbssh` and will be ignored.

Data Type

Number. `rlimit_stack` is read-only. `runrlimit_stack` is modifiable.

Description

These variables control the maximum size the process stack in bytes as a 32-bit number. `rlimit_stack` is the read-only value for the user who invoked Privilege Management for Unix & Linux. `runrlimit_stack` is the modifiable value for the target secured task.



Note: To enable `runrlimit_stack` functionality, set `runenablerlimits` to a value of `1`.

Syntax

```
runrlimit_stack = number;
```

Valid Values

Vary according to platform

Example

```
runrlimit_stack = 1000;
```

See Also

```
rlimit_as, runrlimit_as, rlimit_core, runrlimit_core, rlimit_cpu, runrlimit_cpu, rlimit_data, runrlimit_data, rlimit_fsize, runrlimit_fsize, rlimit_locks, runrlimit_locks, rlimit_memlock, runrlimit_memlock, rlimit_nofile, runrlimit_nofile, rlimit_nproc, runrlimit_nproc, rlimit_rss, runrlimit_rss
```

false

Data Type

Boolean, read-only

Description

The **false** variable is a read-only variable with a predefined value of **0**.

Many program statements rely upon conditional tests to determine what program statement should be executed next. The if statement is an example of this. Conditional tests evaluate to either a **true** value or a **false** value. In the security policy scripting language, a **true** value is represented by any positive, non-zero integer, but is usually represented by the integer value **1**. A **0** represents **false**.

Because **true** and **false** values are used so frequently within security policy files, the variable **true** may be used in place of a numeric value **1** and the variable **false** may be used in place of a **0** value when evaluating a conditional expression or initializing a variable.

Valid Values

0. Constant, cannot be changed

See Also

```
true
```

hour

Data Type

Integer, read-only

Description

The **hour** variable contains the current hour, taken from the Policy Server host, in HH format.

Valid Values

An integer ranging from 0 - 23 (inclusive) from the Policy Server host

See Also

```
date, day, dayname, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_date

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_date** variable contains the current date, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a date.

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_day

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_day** variable contains the current date, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a day value.

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_dayname

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_dayname** variable contains the current day of the week, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a value for the day of the week

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_hour

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_hour** variable contains the current hour, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains an hour value

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_minute

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_minute** variable contains the minute portion of the current time, taken from the Policy Server host. It is formatted according to the operating system's locale settings. The month, day, date, and year variables can be used together to determine the current date, per the Policy Server host. The hour and minute variables can be used together to determine the current time, per the Policy Server host.

Valid Values

A UTF-8 encoded string that contains a minute value

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour,
i18n_month, i18n_time, i18n_year
```

i18n_month

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_month** variable contains the current month, taken from the Policy Server host. It is formatted according to the operating system's locale settings. The month, day, date, and year variables can be used together to determine the current date per the Policy Server host. The hour and minute variables can be used together to determine the current time per the Policy Server host.

Valid Values

A UTF-8 encoded string that contains the month value

selinux

- **Version 5.2 and earlier:** **selinux** variable not available
- **Version 6.0 and later:** **selinux** variable available

Data Type

Integer, read-only

Description

The **selinux** variable indicates whether the **pbrun** client that is requesting the secured task is running confined in the SELinux environment. This variable is not present when the submit host is not integrated with SELinux. You can use the **isset()** function to determine if **pbrun** is running confined.

Valid Values

An integer, as described above. If **pbrun** is running unconfined, the variable is not present.

Example

```
if (isset("selinux")
{
print ("SELINUX: ", selinux);
}
```

runchroot

Data Type

String, modifiable

Description

The **runchroot** variable contains the name of the user's root directory. A secured task can access only those files that reside within that root directory. To change the root directory for the current task, set **runchroot**.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

To use Privilege Management for Unix & Linux with the directory that is specified in the **runchroot** variable, the following files must be copied into that directory:

Files	Target Directory
/etc/pb.settings	runchroot/etc
Key files in /etc (if using Privilege Management for Unix & Linux encryption)	runchroot/etc
/usr/lib/symark/pb/* (if using Kerberos, SSL, or LDAP)	runchroot/usr/lib/symark/pb

In addition, if the **pbrunlog** setting has a value, you must create a corresponding directory under the directory that is specified in **runchroot**. For example, if **pbrunlog** is set to **/var/log/pbrun.log**, then create a **runchroot/var/log** directory.

Syntax

```
runchroot = string;
```

Valid Values

A string that contains a valid absolute path specification. The default value is empty, which implies that the entire run host's file system is accessible.

Example

```
runchroot = "/usr/local/newroot";
```

See Also

```
cwd, runcwd
```

runcksum

Data Type

String, modifiable

Description

The **runcksum** variable stores a checksum value. By default, **runcksum** is an empty string. Populate it by running the Privilege Management for Unix & Linux utility program **pbsum**, which generates application and file checksum values.

Use checksum values to determine if a file or application has changed by establishing a baseline checksum and then comparing that baseline checksum against a checksum that is generated during security policy file processing. If the checksum values are different, then the file or application has changed since generation of the baseline checksum, and Privilege Management for Unix & Linux will refuse to run it.

Application checksum values can be used to determine if a virus has infected an application or if the file has been changed.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runcksum = string;
```

Valid Values

A string that contains a checksum value that is generated by **pbsum**. The default value is empty, which specifies no checksum checking.

Example

```
runcksum = "2f9777ff";
```

See Also

```
pbsum
```

runcksumlist

Data Type

List

Description

The **runcksumlist** variable contains a list of checksum values. By default, **runcksumlist** is an empty list. Populate it by running the Privilege Management for Unix & Linux utility program **pbsum**, which generates application and file checksum values.

Use checksum values to determine if the target files or applications have changed by establishing baseline checksum values and then comparing those baseline checksum values against a checksum that is generated during security policy file processing. If the checksum value that was generated during security policy file processing does not match any of the values in **runcksumlist**, then the file or application has changed since generation of the baseline checksum, and Privilege Management for Unix & Linux will refuse to run it.

Application checksum values can be used to determine if a virus has infected an application or if the file has been changed.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runcksumlist = list of checksum values;
```

Valid Values

A list of strings that represents checksum values generated by **pbsum**. The default value is empty, which specifies no checksum checking.

Example

```
runcksumlist={"b3b156bc", "59bf4a99"};
```

See Also

`pbsum`, `runcksum`

runconfirmmessage

Data Type

String, modifiable

Description

The **runconfirmmessage** variable contains the prompt that is displayed when the submitting user is required to enter a password. If a prompt is not set in **runconfirmmessage**, then the following default prompt is used: *type in the user's password*.

The Privilege Management for Unix & Linux variable **runconfirmuser** determines if a password is required.

There is no read-only version of this variable.

Syntax

```
runconfirmmessage = string;
```

Valid Values

A string containing a user-password prompt. The default value is empty, which defaults to *type in the user's password*.

Example

```
runconfirmmessage = "Please enter the password for pat";
```

See Also

```
runconfirmuser
```

runconfirmpasswdservice

Data Type

String, modifiable

Description

The **runconfirmpasswdservice** variable stores the name of the PAM password service which will be used to perform password authentication and account management for the user named by the **runconfirmuser** variable. It overrides **pampasswdservice** in **pb.settings** of the run host.

There is no read-only version of this variable.

Syntax

```
runconfirmpasswdservice = pam_password_service;
```

Valid Values

A string that contains a name of a valid PAM password service that is present on the run host. There is no default value. If this variable is not defined, the server setting **pampasswdservice** (if set) will be used.

Example

```
runconfirmpasswdservice = "pbul_pam_stack";
```

See Also

```
runconfirmuser, runhost
```



For information about **pampasswordservice**, please see the [Privilege Management for Unix & Linux System Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

runconfirmuser

Data Type

String, modifiable

Description

The **runconfirmuser** variable controls whether or not a user must correctly enter a password before the current task request is executed. When this variable is set, the submitting user is prompted for the password that is associated with the run host user name that is set in this variable.

The variable **runconfirmmessage** determines the password prompt that is displayed to the user after the policy is finished, but before the run host starts the command request. When setting **runconfirmuser**, it is a good idea to set **runconfirmmessage**.

If the user fails in three attempts to submit the correct password, the secured task request is not executed. Because the secured task has already been accepted, the Privilege Management for Unix & Linux event log records an exit status of *ConfirmUser <username> failed*.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runconfirmuser = user;
```

Valid Values

A string that contains a user name that is present on the run host (as specified in the **runhost** variable), for which a password must be supplied before the current task request can be run. The default value is empty, which indicates this password check will not be performed.

Example

```
runconfirmuser = "sandy";
```

See Also

```
runconfirmmessage, runhost
```

runeffectivegroup

Data Type

String, modifiable

Description

runeffectivegroup provides control over the effective group ID (egid) of the secured task. Setting this to a group name makes that group the effective group for the task. If **runeffectivegroup** is not set, then the value of **rungroup** specifies the effective group.

Any change to the **rungroup** variable resets **runeffectivegroup** to the same value. If you want **runeffectivegroup** to be different from **rungroup**, then set **runeffectivegroup** after **rungroup**.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runeffectivegroup = group;
```

Valid Values

A string that contains a valid group name. The default value is the value of **rungroup**.

Example

```
runeffectivegroup = "bin";
```

See Also

```
pblogdreconnection, pbrunreconnection, rungroup, runuser
```

runeffectiveuser

Data Type

String, modifiable

Description

runeffectiveuser provides control over the effective user ID (euid) of the requested job. Setting this variable to a user name makes that user the effective user for the job. If it is not set, the value of **runuser** specifies the effective user.

Any change to the **runuser** variable resets **runeffectiveuser** to the same value. If you want **runeffectiveuser** to be different from **runuser**, then set **runeffectiveuser** after **runuser**.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runeffectiveuser = string;
```

Valid Values

A string containing a valid user name. The default value is the value of **runuser**.

Example

```
runeffectiveuser = "bin";
```

See Also

```
pblogdreconnection, pbrunreconnection, runeffectivegroup
```

runablerlimits

- **Version 3.5 and earlier:** **runablerlimits** variable not available
- **Version 4.0 and later:** **runablerlimits** variable available

Data Type

Boolean

Description

This variable determines if the **runrlimit** variables will be used on the run host. This variable must be set to a value of **1** to enable the functionality of the following variables: **rlimit_as**, **rlimit_core**, **rlimit_cpu**, **rlimit_data**, **rlimit_fsize**, **rlimit_locks**, **rlimit_memlock**, **rlimit_nofile**, **rlimit_nproc**, **rlimit_rss**, **rlimit_stack**.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runenablerlimits = boolean;
```

Valid Values

true	Use the runrlimit_* values on the run host.
false	Ignore the runrlimit_* values and use the run host native ulimits . The default is false .

Example

```
runenablerlimits = true;
```

See Also

```
rlimit_*, runrlimit_*
```

runmd5sum

Data Type

String, modifiable

Description

The **runmd5sum** variable stores an MD5 checksum value. By default, **runmd5sum** is an empty string. Populate it by running the Privilege Management for Unix & Linux utility program **pbsum -m <file names>**, which generates the application and file MD5 checksum values.

Use checksum values to determine if a file or application has changed by establishing a baseline checksum and then comparing that baseline checksum against a checksum that is generated during security policy file processing. If the checksum values are different, then the file or application has changed since the generation of the baseline checksum, and Privilege Management for Unix & Linux will refuse to run it.

Application checksum values can be used to determine if a virus has infected an application or if the file has been changed.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runmd5sum = string;
```

Valid Values

A string containing a checksum value generated by **pbsum**. The default value is empty, which specifies no checksum checking.

Example

```
runmd5sum = "dda5b3a11ac4e203190fbf0643722a05";
```

See Also

`pbsum`

runmd5sumlist

Data Type

List

Description

The **runmd5sumlist** variable contains a list of MD5 checksum values. By default, **runmd5sumlist** is an empty list. Populate it by running the Privilege Management for Unix & Linux utility program **pbsum -m <file names>**, which generates application and file MD5 checksum values.

Use MD5 checksum values to determine if the target files or applications have changed by establishing baseline checksum values and then comparing those baseline checksum values against a checksum that is generated during security policy file processing. If the checksum value that was generated during security policy file processing does not match any of the values in **runmd5sumlist**, then the file or application has changed since generation of the baseline checksum, and Privilege Management for Unix & Linux will refuse to run it.

Application MD5 checksum values can be used to determine if a virus has infected an application or if the file has been changed.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runmd5sumlist = list of checksum values;
```

Valid Values

A list of string that represents MD5 checksum values generated by **pbsum -m <file names>**. The default value is empty, which specifies no checksum checking.

Example

```
runmd5sumlist={"478cd2ea4b868c459d3fcd3132b00853",  
"38a0b33c1f5fa6a2ababf0ce386a2494"};
```

See Also

```
pbsum, runmd5sum
```

runenvironmentfile

- **Version 5.2 and earlier:** **runenvironmentfile** not available
- **Version 6.0 and later:** **runenvironmentfile** available

Data Type

String

Description

The **runenvironmentfile** variable enables you to specify the absolute path and file name of an environment file. Privilege Management for Unix & Linux can incorporate the environment variables that are specified in the environment file into the run environment. These environment variables are applied on the run host after the Accept event has been logged.

The **runenvironmentfile** variable overrides the **environmentfile** setting in the **pb.settings** file on the run host.

There is no read-only version of this variable.

The environment file must consist of the following:

- Comment lines, which have a # character in the first non-whitespace position.
- Blank lines
- Bourne shell compatible environment variable setting lines with the form NAME=VALUE

Each line in the file must contain less than 1024 characters. Line continuation is not supported. This file must not contain any shell commands or constructs other than the setting of environment variables. Comments must not appear on the same line as an environment variable.

Syntax

```
runenvironmentfile = string;
```

Valid Values

A string that contains the absolute path and file name of an environment file. The default value is empty.

Example

```
runenvironmentfile = "/etc/environment";
```

runpamssessionservice

Data Type

String, modifiable

Description

The **runpamssessionservice** variable stores the name of the PAM service which will be used to perform account management and session setup and teardown to manage task requests on a run host. It overrides **pamssessionservice** in **pb.settings** of the run host.

There is no read-only version of this variable.

Syntax

```
runpamssessionservice = pam_password_service;
```

Valid Values

A string that contains a name of a valid PAM session service that is present on the run host. There is no default value. If this variable is not defined, the run host's **pb.setting pamssessionservice** (if set) will be used.

Example

```
runpamssessionservice = "pbul_pam_stack";
```

See Also

```
runhost
```



For more information about **pamssessionservice**, please see the [Privilege Management for Unix & Linux System Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

runpamsetcred

Data Type

Integer, modifiable

Description

The **runpamsetcred** variable enables the **pam_setcred()** function, which is used to establish possible additional credentials of a user. It overrides **pamsetcred** in **pb.settings** of the run host.

There is no read-only version of this variable.

Syntax

```
runpamsessionsservice = pam_password_service;
```

Valid Values

1 or true	Enable pam_setcred()
0 or false	Do not enable pam_setcred()

Example

```
runpamsetcred = 1;
```

See Also

```
runhost
```



For more information about **pamsetcred**, please see the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

runpid

Data Type

Number, read-only

Description

The **runpid** variable contains the PID of the Privilege Management for Unix & Linux module processing the secured task. In the case of optimized run mode, this PID (for **pbrun**) is the same as the **submitpid**. Otherwise, this contains the PID of **pblocald**.

This read-only variable is not available during the processing of the policy, because it is created after the policy performs an accept. This variable is available in the event log.

There is no run version of this variable.

Valid Values

A number that contains a pid.

This is a read-only variable.

See Also

```
logpid, pid, submitpid, taskpid
```

runptyflags

- **Version 3.5 and earlier:** `runptyflags` not available
- **Version 4.0 and later:** `runptyflags` available

Data Type

Internal

Description

Flags that are used internally for pty settings; reserved for internal use.



Note: This run variable does not apply to `pbssh`. If it is present in the policy, it will not have any effect on `pbssh` and will be ignored.

runsecurecommand

- **Version 3.5 and earlier:** `runsecurecommand` variable not available
- **Version 4.0 and later:** `runsecurecommand` variable available

Data Type

Boolean

Description

The `runsecurecommand` variable enables you to perform an extra check on the security of the requested command. This check helps ensure that someone other than root or the runuser (for example, `sys` or `oracle`) could not have compromised the command.

When set to `true`, the run command and all directories above it are checked to see if anyone other than root or the run user has write permission. If the command file or any of the directories above it are writable by anyone other than root or the runuser, then the run host refuses to run the command. The `runsecurecommand` setting can be set to `yes` on the run host for the same effect.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runsecurecommand = boolean;
```

Valid Values

true	Non-zero. Check that the runcommand is writable only by root or the runuser.
false	Zero. No check is performed. The default is false .

Example

```
runsecurecommand = true;
```

runtimeLimit

- **Version 3.5 and earlier:** **runtimeLimit** variable not available
- **Version 4.0 and later:** **runtimeLimit** variable available

Data Type

Integer, modifiable

Description

The **runtimeLimit** variable specifies a time limit for a task request. If the job does not finish within the specified number of seconds, then it is terminated. This is similar to **runtimeout**, but is based on total time rather than idle time.



Note: The **runtimeLimit** variable is not honored in local mode.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runtimeLimit = number;
```

Valid Values

positive number	Enable time limit checking
0 or negative number	Disable time limit checking. This setting is the default.

Example

```
runtime-limit = 3600;
```

See Also

```
runtimeout, submittimeout, runtimewarn(), and runtimewarnlog
```

runtimeout

Data Type

Integer, modifiable

Description

The **runtimeout** variable specifies the amount of idle time, in seconds, that the submitting user is allowed before the run host terminates the current request. To change the idle time specification, set **runtimeout**.

There is no read-only version of this variable.



Note: The **runtimeout** variable is not honored in local mode.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runtimeout = number;
```

Valid Values

positive number	Enable idle checking
0 or negative number	Disable idle checking. This setting is the default.

Example

```
runtimeout = 600;
```

See Also

```
runtimelimit, submittimeout
```



For more information about **runtimeout** and **runtimeoutoverride**, please see the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

runutmpuser

Data Type

String, modifiable

Description

The **runutmpuser** variable contains the User Id that appears in the **utmp** logs on the run host. By default, **runutmpuser** is set to the value of the **user** variable. To change the name of the user that appears in **utmp**, set **runutmpuser**. If user does not exist on the run host, then **runutmpuser** is set to the value of the **runuser** variable.

There is no read-only version of this variable.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Syntax

```
runutmpuser = string;
```

Valid Values

A string that contains the **utmp** User Id. The default value is the value of the **user** variable.

Examples

```
runutmpuser = "root";
```

```
runutmpuser = "runuser";
```

See Also

```
requestuser, runuser, user
```

shellallowedcommands

- **Version 3.5 and earlier:** `shellallowedcommands` variable not available
- **Version 4.0 and later:** `shellallowedcommands` variable available

Data Type

List

Description

This variable contains a list of strings that contain commands that may be run without any further authorization. Each element of the list can contain either a command basename or absolute path. Shell template characters can be used at any point. This variable is used by `pbsh` and `pbksh` at startup time.

Syntax

```
shellallowedcommands = list;
```

Valid Values

A list of strings containing commands

Example

```
if (pbclientmode == "shell start")
shellallowedcommands = {"date", "/bin/df", "/usr/local/bin/*"};
```

See Also

```
pbclientmode, shellcheckbuiltins, shellcheckredirections, shellforbiddencommands,
shellloginincludefiles, shellreadonly
```

shellcheckbuiltins

- **Version 3.5 and earlier:** `shellcheckbuiltins` variable not available
- **Version 4.0 and later:** `shellcheckbuiltins` variable available

Data Type

Boolean

Description

When set to **true**, this variable directs the shell to check shell built-in commands as if they were standard commands. This variable is used by **pbsh** and **pbksh** at startup time.

Syntax

```
shellcheckbuiltins = boolean;
```

Valid Values

true	Privilege Management for Unix & Linux shells authorize and log shell built-in commands.
false	Privilege Management for Unix & Linux shells do not authorize or log shell built-in commands.

Example

```
shellcheckbuiltins = true;
```

See Also

```
pbclientmode, shellallowedcommands, shellcheckredirections, shellforbiddencommands,  
shellloginincludefiles, shellreadonly
```

shellcheckredirections

- **Version 3.5 and earlier:** **shellcheckredirections** variable not available
- **Version 4.0 and later:** **shellcheckredirections** variable available

Data Type

Boolean

Description

When set to **true**, this variable directs the shell to authorize I/O redirections (for example, <, >, >>). When this variable is set to **false**, I/O redirection is always allowed. **pbsh** and **pbksh** use this variable at startup time.

Syntax

```
shellcheckredirections = boolean;
```

Valid Values

true	Privilege Management for Unix & Linux shells authorize and log shell I/O redirection requests.
false	Always allows I/O redirection

Example

```
shellcheckredirections = true;
```

See Also

```
pbclientmode, shellallowedcommands, shellcheckbuiltins, shellforbiddencommands,  
shellloginincludefiles, shellreadonly
```

shellforbiddencommands

- **Version 3.5 and earlier:** `shellforbiddencommands` variable not available
- **Version 4.0 and later:** `shellforbiddencommands` variable available

Data Type

List

Description

This variable contains a list of strings that specify commands that will be rejected by **pbksh** and **pbsh** without consulting a Privilege Management for Unix & Linux Policy Server daemon. Each element of the list can contain either a command basename or absolute path. Shell template characters can be used at any point. This variable is used by **pbsh** and **pbksh** at startup time.

Syntax

```
shellforbiddencommands = list;
```

Valid Values

A list of strings as described above

Example

```
if (pbclientmode == "shell start")  
shellforbiddencommands = {"/etc/*", "/usr/sbin/*",  
"format", "/sbin/umount"};
```

See Also

```
pbclientmode, shellallowedcommands, shellcheckbuiltins, shellcheckredirections,  
shellloginincludefiles, shellreadonly
```

shellloginincludefiles

- **Version 3.5 and earlier:** **shellloginincludefiles** variable not available
- **Version 4.0 and later:** **shellloginincludefiles** variable available

Data Type

Boolean

Description

This variable controls whether the contents of included (sourced) shell scripts should be recorded in the I/O logs.

This is effective only if I/O logging for the shell is enabled. This variable is used by **pbsh** and **pbksh** at startup time.

Syntax

```
shellloginincludefiles = boolean;
```

Valid Values

true	Privilege Management for Unix & Linux shells authorize and log files that shell scripts and profiles include (source).
false	Contents of included shell scripts are not recorded in I/O logs

Example

```
if (pbclientmode == "shell start") shellloginincludefiles = true;
```

See Also

```
pbclientmode, shellallowedcommands, shellcheckbuiltins, shellcheckredirections,  
shellforbiddencmds, shellreadonly
```

shellreadonly

- **Version 3.5 and earlier:** **shellreadonly** variable not available
- **Version 4.0 and later:** **shellreadonly** variable available

Data Type

List

Description

The variable **shellreadonly** contains a list of environment variables that **pbsh** and **pbksh** will set to read-only at startup time. If the variable does not exist at start up time, then its entry is ignored. **pbsh** and **pbksh** use this variable at startup time.

Syntax

```
shellreadonly = list;
```

Valid Values

A list of environment variables

Example

```
if (pbclientmode == "shell start")
shellreadonly = {"PATH", "IFS", "SHELL", "ENV"};
```

See Also

pbclientmode, shellallowedcommands, shellcheckbuiltins, shellcheckredirections,
shellforbiddencommands, shellloginincludefiles

shellrestricted

- **Version 3.5 and earlier:** **shellrestricted** variable not available
- **Version 4.0 and later:** **shellrestricted** variable available

Data Type

Boolean

Description

Controls whether Privilege Management for Unix & Linux shells run in restricted mode. Restricted mode has the following limitations:

- The **cd** command is disabled
- The environment variables **SHELL**, **ENV**, and **PATH** are read-only
- Command names cannot use absolute or relative paths
- The **-p** option of the built-in command is disabled
- I/O redirections (**>**, **>|**, **>>**, and **<>**) that create files are disabled

Syntax

```
shellrestricted = boolean;
```

Valid Values

true	Runs Privilege Management for Unix & Linux shells in restricted mode
false	Disables restricted mode. The default is false .

Example

```
shellrestricted = true;
```

See Also

```
shellallowedcommands, shellcheckbuiltins, shellcheckredirections, shellforbiddencommands,  
shellloginincludefiles, shellreadonly
```

solarisproject

- **Version 6.0 and earlier:** **solarisproject** not available
- **Version 6.1 and later:** **solarisproject** available

Run Version

runsolarisproject



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

String, **solarisproject** is read-only. **Runsolarisproject** is modifiable.

Description

The **solarisproject** and **runsolarisproject** variables specify a Solaris project that the secured task should be associated with on a Solaris 9 or higher runhost. These variables initially contain the project specified on the **pbrun** commandline, or the empty string "" if not specified on the **pbrun** commandline. If the project has not been specified (**runsolarisproject** equals ""), the default project (as defined by Solaris) will be associated with the secured task. If set to a non-valid project name for the runuser, or specified for a non-Solaris runhost, the secured task will not be executed.

Valid Values

A string containing a valid Solaris project on a Solaris runhost.

Examples

```
runsolarisproject group.acctng  
runsolarisproject user.database
```

Backwards Compatibility

Earlier versions **pbmastertd** will not set the **solarisproject** and **runsolarisproject** variables; however, the policy can set the **runsolarisproject** variable.

submithost

Data Type

String, read-only

Description

The **submithost** variable contains the name of the machine from which the current task request was submitted (that is, the submit host). **submithost** is what the Policy Server considers the client name to be (based on the current **submithost** network interface).

The **submithost** and **host** and **runhost** variables are closely related. By default, the host and runhost variables are set to **submithost**, unless the user requests a specific run host by using the **-h** argument of the **pbrun** command.

There is no run version of this variable.

Valid Values

A string that contains the fully qualified name of the submit host machine. This is a read-only variable.

See Also

```
host, runhost, ipaddress, masterhost, pbrun, pid, subprocuser, submithostip, timezone submithostip
```

submithostip

Data Type

String, read-only

Description

The **submithostip** variable contains the IP address of the machine from which the current task request was submitted (that is, the submit host).

There is no run version of this variable.

Valid Values

A string that contains a valid IP address. This is a read-only variable.

See Also

```
host, ipaddress, masterhost, pbrun, pid, runhost, submithost, subprocuser, timezone
```

submitpid

Data Type

Number, read-only

Description

The **submitpid** variable contains the PID of the client (**pbrun**, **pbsh**, **pbksh**) submitting the task request.

This read-only variable is available during the processing of the policy, and in the event log.

There is no run version of this variable.

Valid Values

A number that contains a PID.

This is a read-only variable.

See Also

```
logpid, pid, runpid, taskpid
```

taskpid

Data Type

Number, read-only

Description

The **taskpid** variable contains the PID of the secured task launched by **pbrun**, or the session associated with **pbksh/pbsh** if **iologging** is on.

This variable is populated when the secured task is executed, and has no value until a session starts and therefore cannot be used in the policy. This variable is shown in the Finish event of the **eventlog** only when a **logserver** is used. It can also be used in the new 7.0 syslog formatting settings, **syslogsession_start_format** and **syslogsession_finish_format**.

For **pbksh** and **pbsh**, this variable is only populated if **iologging** is turned on.

Valid Values

A number that contains a PID. This is a read-only variable.

Example pb.settings:

```
syslogsession_finished_format "Privilege Management for Unix & Linux finished %command%  
pid:%taskpid% on %date% at %hour%:%minute%."
```

taskttyname

Data Type

String, read-only

Description

The **taskttyname** variable contains the name of the TTY device (that is, the terminal) associated to the secured task launched by **pbrun**, or the session associated with **pbksh/pbsh** if **iologging** is on.

This variable is populated when the secured task is executed, and has no value until a session starts and therefore cannot be used in the policy. This variable is shown in the Finish event of the **eventlog** only when a **logserver** is used. It can also be used in the new 7.0 syslog formatting settings, **syslogsession_start_format** and **syslogsession_finish_format**.

For **pbksh** and **pbsh**, this variable is only populated if **iologging** is turned on.

Valid Values

A string that contains a TTY name. This is a read-only variable.

timezone

Data Type

String, read-only

Description

The **timezone** variable contains a standard representation of the time zone on the machine from which the current task request was submitted (that is, the submit host). The **timezone** variable is relevant for users working in a cross-platform environment in which that submit host is a Sun machine that has its time zone set to a geographic region rather than the usual **timezone** file. Note that this variable applies to Solaris installations. The format of the **timezone** variable is dependent upon the operating system configuration parameters.

There is no run version of this variable.

Valid Values

A string that contains the standard representation of the time zone. The format of the **timezone** variable is dependent upon operating system configuration parameters. This is a read-only variable.

See Also

```
submithost
```

ttyname

Data Type

String, read-only

Description

The **ttyname** variable contains the name of the TTY device (that is, the terminal) from which the current task request was submitted on the submit host. If the client is running in pipe mode, then the value is **null**.

There is no run version of this variable.

Valid Values

A string that contains a TTY name. This is a read-only variable.

umask

Run Version

runumask



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Data Type

Number. **umask** is read-only. **runumask** is modifiable.

Description

The **umask** and **runumask** variables contain **umask** values for the submitting user. The **umask** value determines the default file permissions mask (read, write, execute) for newly created files. To change the **umask** values for the secured task, set **runumask**.



For more information on **umask**, refer to the Unix/Linux manual page for **umask**.

Syntax

```
runumask = number;
```

Valid Values

A string value containing valid **umask** values for the submitting user. These variables have no default values. The **pbrun** command environment initializes these variables.

Example

```
runumask = 022;
```

user

Run Version

runuser

 **IMPORTANT!**

*This run variable does not apply to **pbssh**. If it is present in the policy, it could produce undesirable results.*

Data Type

String. **user** is read-only. **runuser** is modifiable.

Description

The **user** and **runuser** variables specify the user name that is associated with the login name of the user that submitted the current task request (that is, the submitting user). By default, the current task runs under this user ID.

To change the user ID the current task runs under, set the **runuser** variable.

Syntax

```
runuser = string;
```

Valid Values

A string that contains a valid user name on the run host. **user** is a read-only variable and therefore has no default value. The default value of **runuser** is empty.

Example

```
runuser = "root";
```

See Also

```
requestuser, runeffectivegroup, runutmpuser
```

Command Line Parsing Variables

These variables support the **getopt()**, **getopt_long()**, and **getopt_long_only()** policy language functions. These functions examine the read-only task information variable **env**. The following table summarizes the command line parsing variables.

Table 20. Command Line Parsing Variables

Variable	Description
optarg	Contains the parameter for the last argument or an empty string if none was found Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
opterr	Determines whether to print errors from the getopt() , getopt_long () , and getopt_long_only() functions Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
optind	Contains the current argument list index Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
optopt	Contains the letter of the last option that had a problem Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
optreset	Set this to true to restart the getopt functions from the start. The next time a getopt function is called, optind is set to 1 . Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
optstrictparameters	The getopt_long() function provides strict interpretation of argument parameters. In particular, arguments with optional parameters are accepted only in the form --argument=parameter . Some non-compliant programs allow --argument parameter . To make getopt_long() recognize the latter form, set optstrictparameters to false . Version 3.5 and earlier: variable not available Version 4.0 and later: variable available

optarg

- **Version 3.5 and earlier :** **optarg** variable not available
- **Version 4.0 and later:** **optarg** variable available

Data Type

Integer, read-only

Description

Used with **getopt** functions. Contains the parameter for the last argument or an empty string if none was found.

Valid Values

A positive integer

Example

```
if (option == "f") filename = optarg;
```

See Also

```
getopt(), getopt_long(), getopt_long_only(), opterr, optind, optopt, optreset
```

opterr

- **Version 3.5 and earlier:** **opterr** variable not available
- **Version 4.0 and later:** **opterr** variable available

Data Type

Boolean

Description

Used with the **getopt** functions. Determines whether to display errors from these functions.

Valid Values

true	getopt function errors are displayed
false	getopt function errors are not displayed

Example

```
if (opterr == false) accept;
```

See Also

```
getopt(), getopt_long(), getopt_long_only(), optarg, optind, optopt, optreset
```

optind

- **Version 3.5 and earlier** : **optind** variable not available
- **Version 4.0 and later**: **optind** variable available

Data Type

Integer

Description

Used with **getopt** functions. Contains the current argument list index.

Syntax

```
optind = integer;
```

Valid Values

An integer between **0** and **argc**

Example

```
if (optind < argc) accept;
```

See Also

```
getopt(), getopt_long(), getopt_long_only(), optarg, opterr, optopt, optreset
```

optopt

- **Version 3.5 and earlier**: **optopt** variable not available
- **Version 4.0 and later** : **optopt** variable available

Data Type

String, read-only

Description

Used with **getopt** functions. Contains the letter of the last option that had a problem.

Valid Values

A string

Example

```
if (error) print ("Bad option", optopt);
```

See Also

```
getopt(), getopt_long(), getopt_long_only(), optarg, opterr, optind, optreset
```

optreset

- **Version 3.5 and earlier:** `optreset` variable not available
- **Version 4.0 and later:** `optreset` variable available

Data Type

Boolean

Description

Used with `getopt` functions. Set this to `true` to restart the `getopt` functions from the start. The next time a `getopt` function is called, `optind` is set to 1.

Syntax

```
optreset = boolean;
```

Valid Values

true	Sets <code>optind</code> to 1; the next call to <code>getopt()</code> , <code>getopt_long()</code> , or <code>getopt_long_only()</code> will start from the beginning of the <code>argv</code> list.
false	<code>getopt</code> functions are not restarted from the beginning of the <code>argv</code> list

Example

```
optreset = true;
```

See Also

```
getopt(), getopt_long(), getopt_long_only(), optarg, opterr, optind, optopt
```

optstrictparameters

- **Version 3.5 and earlier:** `optstrictparameters` variable not available
- **Version 4.0 and later:** `optstrictparameters` variable available

Data Type

Boolean

Description

The `getopt_long()` function provides strict interpretation of argument parameters. In particular, arguments with optional parameters are accepted only in the form `--argument=parameter`. Some non-compliant programs allow `--argument parameter`. To make `getopt_long()` recognize the latter form, set `optstrictparameters` to `false`.

Syntax

```
optstrictparameters = boolean;
```

Valid Values

true	Allow <code>getopt_long()</code> 's strict interpretation of argument parameters. The default is <code>true</code> .
false	Make <code>getopt_long()</code> recognize <code>--argument</code> parameter specifications

Example

```
optstrictparameters = false;
```

See Also

```
getopt(), getopt_long(), getopt_long_only(), optarg, opterr, optind, optopt, optreset
```

Logging Variables

Privilege Management for Unix & Linux uses logging variables to store both system and task-specific information. Using the security policy scripting language, the Security Administrator can query this information and use it to make security-related decisions about the current task request.

The following table summarizes the logging variables.

Variable	Description
event	Specifies the type of Privilege Management for Unix & Linux event that is currently logged. This is a global variable.
eventlog	Contains the absolute path specification for the current Privilege Management for Unix & Linux event log
exitdate	Contains the completion date for the current task request
exitstatus	Contains the task completion code, also called the return code, for the current task request
exittime	Contains the time, in HH:MM:SS format, of completion for the current task request
forbidkeyaction	Obsolete. Defines the action taken when a forbidden key sequence is entered during the execution of the current request
forbidkeypatterns	Obsolete. Defines the forbidden keystroke sequences, patterns, or both. An element in the forbidkeypatterns list represents each forbidden keystroke pattern or sequence.
i18n_exitdate	Contains the UTF-8 encoded completion date for the current task request.
i18n_exittime	Contains the UTF-8 encoded completion time for the current task request
iolog	Contains that absolute path specification for the current I/O log file
logmaximumfailures	Controls the maximum number of log failures for a job
lognopassword	Determines whether non-echoed input, such as passwords, is written to the I/O log file when I/O logging is active
logomit	Specifies which Privilege Management for Unix & Linux variables to omit from the event log. Use this user-defined variable to reduce the disk space that is used by the event log.
logstderr	Specifies whether error output from the current task request is recorded in the I/O log
logstderrlimit	Places a limit on the number of bytes from the standard error stream that Privilege Management for Unix & Linux writes to the I/O log at a time
logstdin	Specifies whether input from the current task request is logged to the I/O log
logstdinlimit	Places a limit on the number of bytes from the standard input stream that Privilege Management for Unix & Linux writes to an I/O log at a time.
logstdout	Specifies whether normal output from the current task request is logged to the I/O log.
logstdoutlimit	Places a limit on the number of bytes from the standard output stream that Privilege Management for Unix & Linux writes to the I/O log at a time.

passwordloggingprompts	Specifies the password prompts to be recognized when the lognpassword variable is set.
-------------------------------	---

event

Data Type

String

Description

The event variable specifies the type of Privilege Management for Unix & Linux event that is currently logged. This is a global variable.

Valid Values

accept	The current task request has passed security policy file validation criteria.
finish	The task has completed execution.
keystroke	The current task was terminated because of a forbidden keystroke pattern.
reject	The current task request did not pass security policy file validation criteria and was not executed.

This variable appears only in the event log.

See Also

accept, eventlog, reject

eventlog

Data Type

String

Description

The **eventlog** variable contains the absolute path specification for the current Event Log. The default value comes from the settings file or depends on the operating system, but this policy variable will always supercede those other definitions. Any parent directory in the path will be automatically created.

Beginning in version 10.3.0, new event log formats, such as SQLite DB and ODBC, were introduced. However, the filename specified by the **eventlog** variable in the policy will always be created in the original proprietary flat file format.

Syntax

```
eventlog = <absolute filename >
```

Valid Values

A string that contains the absolute path specification for the Event Log for the current secured task

Example

In the following example, the path defined by the **eventlog** policy variable will override the default value in the settings file.

```
eventlog = '/var/log/pmul/hr001/pb.eventlog';
```



For more information, please see the sections for the **eventdestinations** and **eventlog** settings keywords in the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

exitdate

Data Type

String, read-only

Description

The **exitdate** variable contains the completion date from the Policy Server for the current task request. The date is in YYYY/MM/DD format.

Valid Values

A string that contains the task completion date, in YYYY/MM/DD format, for the current task request. This is a read-only variable and appears only in the event log.

See Also

```
exitstatus, exittime, i18n_exitdate, i18n_exittime
```

exitstatus

Data Type

String, read-only

Description

The **exitstatus** variable contains the task completion code, also called the return code, for the current task request.

Valid Values

"The command exited with a Where **x** is the status code that is returned by the current task request.

status of x"	
"Command caught signal ## (XXXX)"	A signal that terminated the current task request.
"Idle Timeout Reached"	The current task request terminated because it exceeded the maximum idle time. The runtimeout variable sets the maximum idle time.
"Exec failed"	The command that is associated with the current task request was not found.
undefined	Privilege Management for Unix & Linux was unable to execute the command that is associated with the current task request. In this case, the exitstatus variable is undefined (that is, it has a string length of 0). This status indicates that the task may still be running, or aborted due to a network or other crash.

This variable appears only in the event log.

See Also

`exitdate`, `exittime`, `runtimeout`

exittime

Data Type

String, read-only

Description

The **exittime** variable contains the completion time (that is, the time of day that the task completed), for the current task request, from the Policy Server in HH:MM:SS format.

Valid Values

A string that contains the completion time for the current task request, in HH:MM:SS format. This is a read-only variable and appears only in the event log.

See Also

`exitdate`, `exitstatus`, `i18n_exitdate`, `i18n_exittime`

forbidkeyaction

Data Type

String

Description

Obsolete. The **forbidkeyaction** variable defines the action to take if a forbidden key sequence is entered during the execution of the current request.

Syntax

```
forbidkeyaction = action;
```

Valid Values

reject	Immediately terminate the current task request
ignore	Take no action; continue with task processing
Alert or any other string	Log the event in the event log with the specified string and continue with task processing

The default value is empty and no action is taken.

Examples

```
forbidkeyaction = "reject";
```

```
forbidkeyaction = "alert";
```

See Also

```
forbidkeypatterns, setkeystrokeaction
```

forbidkeypatterns

Data Type

List

Description

Obsolete. The **forbidkeypatterns** variable defines the forbidden keystroke sequences, patterns, or both. An element in the **forbidkeypatterns** list represents each forbidden keystroke pattern or sequence.

Wildcard search characters, along with other special characters, can be used to create a keystroke sequence or pattern.



For more information on using wildcard search characters, please see "Wildcard Search Characters" on page 1.

The Privilege Management for Unix & Linux security policy scripting language supports the standard set of shell-style, wildcard search characters. These are used for searches by the in operator and for forbidden and warning keystroke patterns.



For a summary of the available wildcard search characters, please see the table "Wildcard Search Characters" on page 1.

Syntax

```
forbidkeypatterns = {"pattern1", "pattern2", "pattern3", ...};
```

Valid Values

A list in which each element represents a forbidden keystroke sequence or pattern. This variable has no default value.

Example

```
forbidkeypatterns = {"*/bin/rm*", "*rm *", "*xterm*"};
```

See Also

```
forbidkeyaction, setkeystrokeaction
```

i18n_exitdate

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_exitdate** variable contains the completion date from the Policy Server for the current task request. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains the task completion date for the current task request. This read-only variable appears only in the event log.

See Also

```
exitstatus, exittime, i18n_exittime
```

i18n_exittime

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_exittime** variable contains the completion time (that is, the time of day that the task completed), for the current task request. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains the completion time for the current task request. This read-only variable appears only in the event log.

See Also

```
exitdate, exitstatus, i18n_exitdate
```

iolog

Data Type

String

Description

The **iolog** variable contains the absolute path specification for the current I/O log file. The default value for this variable is undefined, which does no I/O logging. The **iolog** file can log standard input, standard output, and standard error information that is associated with the current task request. Any parent directory in the path will be automatically created.

Syntax

```
iolog = string;
```

Valid Values

A string that contains the absolute path specification for the current **iolog** file. The default value is undefined.

Example

```
iolog = "/var/log/sample.log";
```

See Also

```
logmktemp(), mktemp()
```

logmaximumfailures

Data Type

Integer

Description

Controls the maximum number of log failures for a job. When the maximum number of failures is exceeded, the secured task terminates.

The default is **25**. If **logmaximumfailures** is set to **0**, Privilege Management for Unix & Linux will keep trying to log data no matter how many failures occur.

Syntax

```
logmaximumfailures = non-negative-integer;
```

Valid Values

0 to **max_int**

Example

```
logmaximumfailures = 20;
```

See Also

```
eventlog, iolog, logservers
```

lognopassword

Data Type

Boolean

Description

The **lognopassword** variable determines whether non-echoed input, such as passwords, is written to the I/O log file when I/O logging is active.

Starting with version 7.0.0, all input and output is logged until a password prompt is recognized on **stdout**. Password prompts to recognize must be listed in the policy language list variable **passwordloggingprompts** which defaults to **{"Password:", "password:", "Passwd:", "passwd:"}** for v7.0.0 to v7.5.0, and to **{"Password", "password", "Passwd", "passwd"}** for v7.5.1 and later.

After a password prompt is recognized, non-echoed **stdin** is not logged until a newline is received, or until input exceeds 80 characters.

Syntax

```
lognopasswd = boolean;
```

Valid Values

true	Do not log passwords (or other non-echoed input).
false	Log all input keystrokes. This setting is the default.

The initial **lognpassword** value comes from the settings file. If **passwordlogging** is set to **never**, **lognpassword** is set to **true** and becomes read-only.

Example

```
lognpassword = true;
```

See Also

```
passwordloggingprompts
```

logomit

Data Type

List

Description

The **logomit** variable specifies which Privilege Management for Unix & Linux user-defined variables to omit from the event log. Use this variable to reduce the disk space that is used by the event log. Metacharacter patterns can be used. By default, this variable is undefined, which means that all Privilege Management for Unix & Linux variables are written to the event log. Beginning with Privilege Management for Unix & Linux 4.0, **logomit** can accept templates.

Syntax

```
logomit = list;
```

Valid Values

A list in which each element names a Privilege Management for Unix & Linux user-defined variable to omit from the event log. The default value is undefined.

Example

```
logomit = {"a", "b"};
```

See Also

```
env, runenv
```

logstderr

Data Type

Boolean

Description

The **logstderr** variable specifies whether error output from the current task request is logged to the I/O log. The default value is **true**.

Syntax

```
logstderr = boolean;
```

Valid Values

true	Log task error information from stderr . This value is the default
false	Do not log task error information from stderr

Example

```
logstderr = true;
```

See Also

```
logstderrlimit
```

logstderrlimit

Data Type

Integer

Description

The **logstderrlimit** variable places a limit on the number of bytes from the standard error stream that Privilege Management for Unix & Linux writes, at a time, to the I/O log. When data appears on any of the other channels, this variable is reset to zero. A value of 0 results in no limit to the amount of **stderr** data sent to the I/O log. To turn off the logging of task standard error data, set the **logstderr** variable to **false**.

Syntax

```
logstderrlimit = number;
```

Valid Values

integer	An integer specifying the maximum number of bytes.
0	No limit on the number of bytes. This setting is the default.

Example

```
logstderrlimit = 4096;
```

See Also

```
logstderr
```

logstdin

Data Type

Boolean

Description

The **logstdin** variable specifies whether input from the current task request is logged to the I/O log. The default value is **true**.

Syntax

```
logstdin = boolean;
```

Valid Values

true	Log task input information from stdin . This value is the default.
false	Do not log task input information from stdin .

Example

```
logstdin = false;
```

See Also

```
logstdinlimit
```

logstdinlimit

Data Type

Integer

Description

The **logstdinlimit** variable places a limit on the number of bytes from the standard input stream that Privilege Management for Unix & Linux writes, at a time, to the I/O log. When data appears on any of the other channels, the this variable is reset to zero. A value of **0** has the effect of placing no limit on the amount of **stdin** data sent to the I/O log. To turn off the logging of standard input data to the I/O log, set the **logstdin** variable to **false**.

Syntax

```
logstdinlimit = number;
```

Valid Values

positive integer	An integer specifying the maximum number of bytes.
0	No limit on the number of bytes. This value is the default.

Example

```
logstdinlimit = 512;
```

See Also

```
logstdin
```

logstdout

Data Type

Boolean

Description

The **logstdout** variable specifies whether output from the current task request is logged to the I/O log. The default value is **true**.

Syntax

```
logstdout = boolean;
```

Valid Values

true	Log task output information from stdout . This value is the default.
false	Do not log task output information from stdout .

Example

```
logstdout = 1;
```

See Also

```
logstdoutlimit
```

logstdoutlimit

Data Type

Integer

Description

The **logstdoutlimit** variable places a limit on the number of bytes from the standard output stream that Privilege Management for Unix & Linux writes to the I/O log at a time. When data appears on any of the other channels, this variable is reset to zero. A value of 0 has the effect of placing no limit on the amount of **stdout** data sent to the I/O log. Set the **logstdout** variable to **false** to turn off the logging of standard output data to the I/O log.

Syntax

```
logstdoutlimit = number;
```

Valid Values

positive integer	An integer specifying the maximum number of bytes.
0	No limit on the number of bytes. This value is the default.

Example

```
logstdoutlimit = 200;
```

See Also

```
logstdout
```

passwordloggingprompts

- **Version 6.2 and earlier:** `passwordloggingprompts` variable not available
- **Version 7.0 and later:** `passwordloggingprompts` variable available

Data Type

List

Description

The `passwordloggingprompts` variable controls the `lognopassword` feature. When passwords should not be logged, all input and output will be logged until a password prompt is recognized on `stdout`. Password prompts to recognize must be listed in the `passwordloggingprompts` variable. When a password prompt is recognized, non-echoed `stdin` is not logged until a newline is received, or until input exceeds 80 characters.

Syntax

```
passwordloggingprompts = list;
```

Valid Values

A list of character values.

The default list for v7.0.0 to v7.5.0 is `{"Password:", "password:", "Passwd:", "passwd:"}`.

The default list for v7.5.1 and later is `{"Password", "password", "Passwd", "passwd"}`.

Examples

This example sets the list to a single prompt to recognize.

```
passwordloggingprompts = {"Enter ANY string:"};
```

This example sets the list to three prompts to recognize.

```
passwordloggingprompts={"Enter ANY string:", "password:", "passwd:"};
```

This example appends the prompt `"Enter key:"` to the list.

```
passwordloggingprompts={passwordloggingprompts,"Enter key:"};
```

See Also

```
lognopassword
```

System Variables

Privilege Management for Unix & Linux System Variables contain information that pertains to all Privilege Management for Unix & Linux task requests. The following table summarizes the system variables.

Table 22. System Variables

Variable	Description
date	Contains the current date, taken from Policy Server host, in YYYY/MM/DD format
day	Contains the current date, taken from Policy Server host, in DD format
dayname	Contains the current day of the week, as a three-character abbreviation for the day of the week, taken from Policy Server host
false	A read-only variable with a predefined value of 0 . May be used in place of a 0 value when evaluating a conditional expression or initializing a variable
hour	Contains the current hour, taken from Policy Server host, in HH format
i18n_date	Contains the UTF-8 encoded current date, taken from Policy Server host
i18n_day	Contains the UTF-8 encoded current day, taken from Policy Server host
i18n_dayname	Contains the UTF-8 encoded current day of the week, taken from Policy Server host
i18n_hour	Contains the UTF-8 encoded current hour, taken from Policy Server host
i18n_minute	Contains the UTF-8 encoded minute portion of the current time, taken from Policy Server host.
i18n_month	Contains the UTF-8 encoded current month, taken from the Policy Server host
i18n_time	Contains the UTF-8 encoded current time, taken from the Policy Server host
i18n_year	Contains the UTF-8 encoded current year taken from the Policy Server host
lineinfile	Contains the file name of the security policy file that triggered the accept or reject condition for the current task request.
linenum	Identifies the specific line number, within a security policy file, that triggered the accept or reject event for the current task request
logignoreconnect	The logignoreconnect variable controls how Privilege Management for Unix & Linux optimizes network traffic between pblogd and pblocald . This optimization involves reconnecting pblocald directly to pblogd , thus bypassing pbmasterd for log related I/O streams.
masterhost	Contains the fully qualified name of the Policy Server host machine (that is, the machine running pbmasterd)
minute	Contains the minute portion of the current time, taken from Policy Server host, in MM format.
month	The month variable contains the current month, taken from the Policy Server host machine, in MM format.
noreconnect	Controls how Privilege Management for Unix & Linux optimizes network traffic between pbrun and pblocald . This optimization involves reconnecting pbrun directly to pblocald , thus bypassing

	pbmasterd for I/O streams processing.
optimizedrunmode	Indicates whether the Policy Server has optimized pblocald out of the connection. Version 4.0 and earlier: variable not available Version 5.0 and later: variable available
outputredirect	Determines if Privilege Management for Unix & Linux prompt output is written to the standard error stream (stderr) or the standard output stream (stdout)
pbclientcertificateissuer	Contains the certificate issuer line from the client program
pbclientcertificatesubject	Contains the certificate subject line from the client program
pbclientkerberosuser	Contains the name of the client user's principal when Kerberos is used
pbclientmode	Specifies the specific mode for a request Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbclientname	Contains the name of the Privilege Management for Unix & Linux component from which the current task request originated
pblogdreconnection	Affects the formation of the reconnection between pblogd and pblocald
pbrunreconnection	Affects the formation of the reconnection between pbrun and pblocald
pbversion	Contains the version of Privilege Management for Unix & Linux that is being run
pid	An integer that represents the pbmasterd process ID
ptyflags	Reserved for internal use
status	Contains the return code from the last system command that was run by the policy
submittimeout	Specifies the amount of idle time that the submitting user is allowed before the submit host terminates the current request
subprocuser	The subprocuser variable contains the user name under which all Policy Server host (that is, pbmasterd) sub-processes run (for example, commands that are run using the system() function).
time	Contains the current time, taken from the Policy Server host, in HH:MM:DD format (for example, 08:24:52)
true	A read-only variable that has a predefined value of 1 . May be used in place of a numeric value 1 when evaluating a conditional expression or initializing a variable
uniqueid	Contains a 12-character or longer string that is guaranteed to be unique across the entire Privilege Management for Unix & Linux system (that is, Policy Server host, submit host, run host and log host). Use this value to guarantee a unique identification in the Event Log files and to generate unique filenames.
year	Contains the current year taken from the Policy Server host, in YYYY format.

date

Data Type

String, read-only

Description

The **date** variable contains the current date, taken from the Policy Server host, in **YYYY/MM/DD** format.

Valid Values

A string that contains a date, in **YYYY/MM/DD** format, from the Policy Server host.

See Also

```
day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

day

Data Type

Integer, read-only

Description

The **day** variable contains the current date, taken from the Policy Server host, in **DD** format.

Valid Values

An integer that contains a value from 1 - 31 (inclusive) from the Policy Server host. This is a read-only variable and therefore has no default value.

See Also

```
date, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

dayname

Data Type

String, read-only

Description

The **dayname** variable contains the current day of the week, as a three-character abbreviation, taken from the Policy Server host.

Valid Values

A character string from the Policy Server host that contains one of the following values: **Mon, Tue, Wed, Thu, Fri, Sat, or Sun.**

See Also

```
day, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute,
i18n_month, i18n_time, i18n_year
```

false

Data Type

Boolean, read-only

Description

The **false** variable is a read-only variable with a predefined value of **0**.

Many program statements rely upon conditional tests to determine what program statement should be executed next. The if statement is an example of this. Conditional tests evaluate to either a **true** value or a **false** value.

In the security policy scripting language, a **true** value is represented by any positive, non-zero integer, but is usually represented by the integer value **1**. A **0** represents **false**.

Because **true** and **false** values are used so frequently within security policy files, the variable **true** may be used in place of a numeric value **1** and the variable **false** may be used in place of a **0** value when evaluating a conditional expression or initializing a variable.

Valid Values

0. Constant, cannot be changed

See Also

```
true
```

hour

Data Type

Integer, read-only

Description

The **hour** variable contains the current hour, taken from the Policy Server host, in **HH** format.

Valid Values

An integer ranging from **0** to **23** (inclusive) from the Policy Server host

See Also

```
date, day, dayname, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_date

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_date** variable contains the current date, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a date.

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_day

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_day** variable contains the current date, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a day value.

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_dayname

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_dayname** variable contains the current day of the week, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a value for the day of the week

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_hour

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_hour** variable contains the current hour, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains an hour value

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_minute, i18n_month, i18n_time, i18n_year
```

i18n_minute

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_minute** variable contains the minute portion of the current time, taken from the Policy Server host. It is formatted according to the operating system's locale settings. The month, day, date, and year variables can be used together to determine the current date, per the Policy Server host. The **hour** and **minute** variables can be used together to determine the current time, per the Policy Server host.

Valid Values

A UTF-8 encoded string that contains a minute value

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_month, i18n_time, i18n_year
```

i18n_month

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_month** variable contains the current month, taken from the Policy Server host. It is formatted according to the operating system's locale settings. The month, day, date, and year variables can be used together to determine the current date per the Policy Server host. The **hour** and **minute** variables can be used together to determine the current time per the Policy Server host.

Valid Values

A UTF-8 encoded string that contains the month value

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_time, i18n_year
```

i18n_time

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_time** variable contains the current time, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains the current time

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_year
```

i18n_year

Data Type

UTF-8 encoded string, read-only

Description

The **i18n_year** variable contains the current year, taken from the Policy Server host. It is formatted according to the operating system's locale settings.

Valid Values

A UTF-8 encoded string that contains a year value

See Also

```
date, day, dayname, hour, minute, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time
```

lineinfile

Data Type

String, read-only

Description

The **lineinfile** variable contains the file name of the security policy file that triggered the accept or reject condition for the current task request. Note that only the file name, rather than the entire path specification, is contained in this variable.

Valid Values

A character string that contains the name of the security policy file in which an accept or reject event was triggered for the current task request.

This variable appears only in the event log.

See Also

```
linenum
```

linenum

Data Type

Integer, read-only

Description

The **linenum** variable identifies the specific line number, within a security policy file, that triggered the accept or reject event for the current task request. This number is a line number within the security policy file identified by **lineinfile**.

Valid Values

An positive integer. This variable appears only in the event log.

See Also

```
lineinfile
```

logignoreconnect

Data Type

Boolean, modifiable

Description

The **logignoreconnect** variable controls how Privilege Management for Unix & Linux optimizes network traffic between **pblogd** and **pblocald**, and **pblocald** and **pbrun**. This optimization involves reconnecting **pblocald** directly to **pblogd** and **pbrun**, thus bypassing **pbmasterd** for log-related IO streams.

When set to **true**, all **pblocald** to **pblogd** communications are routed through **pbmasterd**, as is **pbrun** to **pblocald** communications.

In Optimized Run Mode, this has no affect.

Syntax

```
logignoreconnect = boolean;
```

Valid Values

true	Disable optimization
false	Enable optimization. This value is the default.

Example

```
lognoreconnect = false;
```

See Also

```
noreconnect
```

masterhost

Data Type

String, read-only

Description

The **masterhost** variable contains the fully qualified name of the Policy Server host machine (that is, the machine that is running **pbmasterd**).

Valid Values

A string that contains the fully qualified name of the Policy Server host

See Also

```
host, runhost, submithost, submithostip
```

minute

Data Type

Integer, read-only

Description

The **minute** variable contains the minute portion of the current time, taken from the Policy Server host, in **MM** format. The month, day, date, and year variables can be used together to determine the current date, per the Policy Server host. The **hour** and **minute** variables can be used together to determine the current time, per the Policy Server host.

Valid Values

An integer that ranges from 0 - 59 inclusive

See Also

```
date, day, dayname, hour, month, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

month

Data Type

Integer, read-only

Description

The **month** variable contains the current month, taken from the Policy Server host, in **MM** format. The **month**, **day**, **date**, and **year** variables can be used together to determine the current date per the Policy Server host. The **hour** and **minute** variables can be used together to determine the current time per the Policy Server host.

Valid Values

An integer ranging from 1 - 12, inclusive

See Also

```
date, day, dayname, hour, minute, time, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

noreconnect

Data Type

Boolean, modifiable

Description

The **noreconnect** variable controls how Privilege Management for Unix & Linux optimizes network traffic between **pbrun** and **pblocald**. This optimization involves reconnecting **pbrun** directly to **pblocald**, thus bypassing **pbmasterd** for I/O stream processing.

Syntax

```
noreconnect = boolean;
```

Valid Values

true	Disable optimization
false	Enable optimization. This value is the default.

Example

```
noreconnect = true;
```

See Also

```
logignoreconnect
```

outputredirect

Data Type

String, modifiable

Description

The **outputredirect** variable determines whether Privilege Management for Unix & Linux prompt output is written to the standard error stream (**stderr**) or to the standard output stream (**stdout**). The main use for this feature is to allow prompts to appear on the user's monitor even if it is running in a pipeline. When run in a pipeline, prompts would normally go to that pipeline. By setting **outputredirect**, you can force the output to the monitor.

Syntax

```
outputredirect = string;
```

Valid Values

stderr	Write Privilege Management for Unix & Linux prompt output to the standard error file
stdout	Write Privilege Management for Unix & Linux prompt output to the standard output file

The default value is empty.

Example

```
outputredirect = "stderr";
```

See Also

```
ilog, logstderr, logstderrlimit, logstdin, logstdout, logstdoutlimit
```

pbclientcertificateissuer

Data Type

String, read-only

Description

This variable contains the issuer line from the client program (**pbrun** or **pbguid**). This variable is available only while the policy is running.

Valid Values

A string that contains the certificate issuer line from the client program

See Also

```
plocaldcertificateissuer, pblogdcertificateissuer, pbmasterdcertificateissuer  
pbclientcertificatesubject
```

pbclientcertificatesubject

Data Type

String, read-only

Description

pbclientcertificatesubject contains the subject line from the client program (**pbrun** or **pbguid**). This variable is available only when the policy is running.

Valid Values

A string that contains the certificate subject line from the client program

See Also

```
plocaldcertificatesubject, pblogdcertificatesubject, pbmasterdcertificatesubject
```

pbclientkerberosuser

Data Type

String, read-only

Description

pbclientkerberosuser contains the name of the client (**pbrun** or **pbguid**) user's principal when Kerberos is used.

Valid Values

A string that contains the name of the client user's principal

pbclientmode

- **Version 3.5 and earlier:** `pbclientmode` variable not available
- **Version 4.0 and later:** `pbclientmode` variable available

Data Type

String, read only

Description

`pbclientmode` specifies the specific mode for a request. It is set as shown in the following table.

Table 23. pbclientmode Invoke Events and Values

How Invoked	pbclientmode Value
<code>pbrun</code>	<code>run</code>
<code>pbssh</code>	<code>pbssh</code>
<code>pbguid</code>	<code>pbguid</code>
<code>pbksh</code> or <code>pbsh</code> startup	<code>shell start</code>
Shell built-in from <code>pbksh</code> or <code>pbsh</code>	<code>shell builtin</code>
Command from shell command line or argument	<code>shell command</code>
Redirection in a shell command (<code><</code> , <code>></code> , or <code>>></code>)	<code>shell redirect</code>

Valid Values

A string as described above

Example

```
if (pbclientmode == "shell start") shellcheckbuiltins = true;
else if (pbclientmode == "shell redirect" && argv[1] == "/dev/null")
reject;
```

See Also

```
shellallowedcommands, shellcheckbuiltins, shellcheckredirections, shellforbiddencommands,
shellreadonly, shellloginincludefiles
```

pbclientname

Data Type

String, read-only

Description

The **pbclientname** variable contains the name of the Privilege Management for Unix & Linux component from which the current task request originated.

Valid Values

pbrun	The current task request originated from pbrun
pbguid	The current task request originated from the Privilege Management for Unix & Linux Web user interface
pbsh	The current task request originated from the pbsh Privilege Management for Unix & Linux shell
pbksh	The current task request originated from the pbksh Privilege Management for Unix & Linux shell

pblogdreconnection

Data Type

Boolean, modifiable

Description

This variable affects the formation of the reconnection between **pblogd** and **pblogald**. If the value is missing or false, then **pblogd** listens for connections that are initiated by **pblogald** under the control of **pblogmasterd**. If **pblogdreconnection** is set to true, then **pblogald** listens for connections that are initiated by **pblogd** under the control of **pblogmasterd**.

There is no read-only version of this variable.

Syntax

```
pblogdreconnection = boolean;
```

Valid Values

true	pblogald listens for connections that are initiated by pblogd under the control of pblogmasterd .
false	pblogd listens for connections that are initiated by pblogald under the control of pblogmasterd . This value is the default.

Example

```
pblogdreconnection = true;
```

See Also

`pbrunreconnection`, `runeffectivegroup`, `runeffectiveuser`

pbrunreconnection

Data Type

Boolean, modifiable

Description

This variable affects the formation of the reconnection between **pbrun** and **pblocald**. If the value is missing or false, then **pbrun** listens for connections that are initiated by **pblocald** under the control of **pbmasterd**. If **pbrunreconnection** is set to **true**, **pblocald** listens for connections that are initiated by **pbrun** under the control of **pbmasterd**.

There is no read-only version of this variable.

Syntax

```
pbrunreconnection = boolean;
```

Valid Values

true	pblocald listens for connections that are initiated by pbrun under the control of pbmasterd .
false	pbrun listens for connections that are initiated by pblocald under the control of pbmasterd . This value is the default.

Example

```
pbrunreconnection = true;
```

See Also

```
pblogdreconnection, runeffectivegroup, runeffectiveuser
```

pbversion

Data Type

String, read-only

Description

The **pbversion** variable contains the version number of Privilege Management for Unix & Linux that is being run.

Valid Values

A string that contains the Privilege Management for Unix & Linux version number

pid

Data Type

Integer, read-only

Description

The **pid** variable contains the Unix or Linux process ID number for **pbmasterd** on the Policy Server host.

Valid Values

An integer that represents the **pbmasterd** process ID

See Also

```
masterhost
```

ptyflags

Data Type

Internal, read-only

Description

Reserved for internal use

status

Data Type

Integer, read-only

Description

The **status** variable contains the return code from the last **system()** command that was run by the policy.

Valid Values

An integer that contains the return code from a call to the **system()** function. The value before the first **system ()** call is undefined.

See Also

```
system ()
```

submittimeout

Data Type

Integer

Description

This variable specifies the idle time, in seconds, that is allotted to the submitting user before the submit host terminates the current request.



Note: The *submittimeout* variable is not honored in local mode.

Syntax

```
submittimeout = number;
```

Valid Values

positive integer

Enables idle checking; specifies the idle time in seconds

0 or negative integer

Disables idle checking. This value is the default.

Example

In the following example, the submitting user is allotted 300 seconds before the request is terminated.

```
submittimeout = 300;
```

See Also

runtimeout

subprocuser

Data Type

String, modifiable

Description

The **subprocuser** variable contains the user name under which all Policy Server host (that is, **pbmasterd**) **subprocesses** run (for example, commands that are run using the **system()** function). By default, all Policy Server host sub-processes run as root.

Syntax

```
subprocuser = string;
```

Valid Values

A string that specifies a user name. The default value is **root**.

Example

```
subprocuser = "user";
```

See Also

```
system()
```

time

Data Type

String, read-only

Description

The **time** variable contains the current time, taken from the Policy Server host in **HH:MM:DD** format (for example, **08:24:52**).

Valid Values

A string containing the current time in **HH:MM:SS** format

See Also

```
date, day, dayname, hour, minute, month, year, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

true

Data Type

Boolean, read-only

Description

The **true** variable is a read-only variable with a predefined value of **1**.

Many program statements rely upon conditional tests to determine what program statement should be executed next. The if statement is an example of this. Conditional tests generally evaluate to either a **true** or **false** value. In the security policy scripting language, any positive, non-zero integer can represent a **true** value, but **1** is normally used. A **0** represents a **false** value.

Because **true** and **false** values are frequently used when creating security policy files, the variable `true` may be used in place of a numeric value **1** and the variable `false` may be used in place of a **0** value when evaluating a conditional expression or initializing a variable.

Valid Values

1. Constant, cannot be changed

See Also

```
false
```

uniqueid

Data Type

String, read-only

Description

The **uniqueid** variable contains a 12-character or longer string that is guaranteed to be unique across the entire Privilege Management for Unix & Linux system (Policy Server host, submit host, run host and log host). This value is used to guarantee a unique identification in the event log files and can be used to generate unique file names. For example:

```
iolog="usr/adm/pblog" + uniqueid;
```

Valid Values

A 12-character or longer string value that is unique across the entire Privilege Management for Unix & Linux system

See Also

```
ipaddress, masterhost
```

year

Data Type

Integer, read-only

Description

The **year** variable contains the current year, taken from the Policy Server host, in **YYYY** format.

Valid Values

An integer that contains a year in **YYYY** format

See Also

```
date, day, dayname, hour, minute, month, time, i18n_date, i18n_day, i18n_dayname, i18n_hour, i18n_minute, i18n_month, i18n_time, i18n_year
```

Host Identification Variables

The host identification variables identify the characteristics of the various Privilege Management for Unix & Linux machines.

The following table summarizes these variables.

Table 24. Host Identification Variables

Variable	Description
masterlocale	The locale setting on the Policy Server host. Version 6.0.1 and earlier: variable not available Version 6.1 and later: variable available
runlocale	The locale setting on the run host. Version 6.0.1 and earlier: variable not available Version 6.1 and later: variable available <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">  Note: This run variable does not apply to pbssh. If it is present in the policy, it will not have any effect on pbssh and will be ignored. </div>
submitlocale	The locale setting on the submit host. Version 6.0.1 and earlier: variable not available Version 6.1 and later: variable available
pbguidmachine	The machine type ID from uname on the GUI host Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbguidnodename	The nodename from uname on the GUI host Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbguidrelease	The operating system release from uname on the GUI host Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbguidsysname	The system name from uname on the GUI host Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbguidversion	The operating system version from uname on the GUI host Version 3.5 and earlier: variable not available Version 4.0 and later: variable available

pbkshmachine	The machine type ID from uname on the pbksh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbkshnodename	The nodename from uname on the pbksh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbkshrelease	The operating system release from uname on the pbksh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbkshsysname	The system name from uname on the pbksh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbkshversion	The operating system version from uname on the pbksh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pblocaldcertificateissuer	The issuer string from the pblocald certificate
pblocaldcertificatesubject	The subject string from the pblocald certificate
pblocaldmachine	The machine type ID from uname on the run host
pblocaldnodename	nodename from uname on the run host
pblocaldrelease	The operating system release from uname on the run host
pblocaldsysname	The system name from uname on the run host
pblocaldversion	The operating system version from uname on the run host
pblogdcertificateissuer	The issuer string from the pblogd certificate
pblogdcertificatesubject	The subject string from the pblogd certificate
pblogdmachine	The machine type ID from uname on the log host
pblogdnodename	The nodename from uname on the log host
pblogdrelease	The operating system release from uname on the log host
pblogdsysname	The system name from uname on the log host
pblogdversion	The operating system version from uname on the log host
pbmasterdcertificateissuer	The issuer string from the pbmasterd certificate
pbmasterdcertificatesubject	The subject string from the pbmasterd certificate

pbmasterdmachine	The machine type ID from uname on the Policy Server host
pbmasterdnodename	The nodename from uname on the Policy Server host
pbmasterdrelease	The operating system from uname on the Policy Server host
pbmasterdsysname	The system name from uname on the Policy Server host
pbmasterdversion	The operating system from uname on the Policy Server host
pbrunmachine	The machine type ID from uname on the submit host
pbrunnodename	The nodename from uname on the submit host
pbrunrelease	The operating system release from uname on the submit host
pbrunsysname	The system name from uname on the submit host
pbrunversion	The operating system version from uname on the submit host
pbshmachine	The machine type ID from uname on the pbsh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbshnodename	The nodename from uname on the pbsh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbshrelease	The operating system release from uname on the pbsh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbshsysname	The system name from uname on the pbsh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available
pbshversion	The operating system version from uname on the pbsh machine Version 3.5 and earlier: variable not available Version 4.0 and later: variable available

masterlocale

- **Version 6.0.1 and earlier:** **masterlocale** variable not available
- **Version 6.1 and later:** **masterlocale** variable available

Data Type

String, read-only

Description

The locale setting on the Policy Server host

Valid Values

A string that contains the locale setting (such as **zh_CN.utf8**) on the Policy Server host.

runlocale

- **Version 6.0.1 and earlier:** **runlocale** variable not available
- **Version 6.1 and later:** **runlocale** variable available

Data Type

String, read-only

Description

The locale setting on the run host.



Note: This run variable does not apply to **pbssh**. If it is present in the policy, it will not have any effect on **pbssh** and will be ignored.

Valid Values

A string that contains the locale setting (such as **zh_CN.utf8**) on the run host.

submitlocale

- **Version 6.0.1 and earlier:** **submitlocale** variable not available
- **Version 6.1 and later:** **submitlocale** variable available

Data Type

String, read-only

Description

The locale setting on the submit host

Valid Values

A string that contains the locale setting (such as **zh_CN.utf8**) on the submit host.

pbguidmachine

- **Version 3.5 and earlier:** **pbguidmachine** variable not available
- **Version 4.0 and later:** **pbguidmachine** variable available

Data Type

String, read-only

Description

The machine type ID from **uname** on the GUI host

Valid Values

A string that contains the machine GUI host hardware from the **uname** command.

pbguidnodename

- **Version 3.5 and earlier:** **pbguidnodename** variable not available
- **Version 4.0 and later:** **pbguidnodename** variable available

Data Type

String, read-only

Description

The nodename from **uname** on the GUI host

Valid Values

A string that contains the GUI host name from the **uname** command.

pbguidrelease

- **Version 3.5 and earlier:** **pbguidrelease** variable not available
- **Version 4.0 and later:** **pbguidrelease** variable available

Data Type

String, read-only

Description

The operating release from **uname** on the GUI host

Valid Values

A string that contains the GUI host operating system version from the **uname** command

pbguidsysname

- **Version 3.5 and earlier:** **pbguidsysname** variable not available
- **Version 4.0 and later:** **pbguidsysname** variable available

Data Type

String, read-only

Description

The system name from **uname** on the GUI host

Valid Values

A string that contains the GUI host operating system implementation string from the **uname** command.

pbguidversion

- **Version 3.5 and earlier:** **pbguidversion** variable not available
- **Version 4.0 and later:** **pbguidversion** variable available

Data Type

String, read-only

Description

The operating system version from **uname** on the GUI host

Valid Values

A string that contains the GUI host operating system version string from the **uname** command

pbkshmachine

- **Version 3.5 and earlier:** **pbkshmachine** variable not available
- **Version 4.0 and later:** **pbkshmachine** variable available

Data Type

String, read-only

Description

The machine type ID from **uname** on the **pbksh** machine

Valid Values

A string that contains the machine hardware ID from the **uname** command.

pbkshnodename

- **Version 3.5 and earlier:** **pbkshnodename** variable not available
- **Version 4.0 and later:** **pbkshnodename** variable available

Data Type

String, read-only

Description

The nodename from **uname** on the **pbksh** machine

Valid Values

A string that contains the nodename from the **uname** command.

pbkshrelease

- **Version 3.5 and earlier:** **pbkshrelease** variable not available
- **Version 4.0 and later:** **pbkshrelease** variable available

Data Type

String, read-only

Description

The operating system release from **uname** on the **pbksh** machine

Valid Values

A string that contains the operating system version from the **uname** command

pbkshsysname

- **Version 3.5 and earlier:** **pbkshsysname** variable not available
- **Version 4.0 and later:** **pbkshsysname** variable available

Data Type

String, read-only

Description

The system name from **uname** on the **pbksh** machine

Valid Values

A string that contains the operating system implementation string from the **uname** command.

pbkshversion

- **Version 3.5 and earlier:** **pbkshversion** variable not available
- **Version 4.0 and later:** **pbkshversion** variable available

Data Type

String, read-only

Description

The operating system version from **uname** on the **pbksh** machine

Valid Values

A string that contains the operating system version from the **uname** command

pblocaldcertificateissuer**Data Type**

String, read-only

Description

The issuer string from **pblocald**'s certificate. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains **pblocald**'s certificate issuer line

See Also

```
pbclientcertificateissuer, pblogdcertificateissuer, pbmasterdcertificateissuer
```

pblocaldcertificatesubject**Data Type**

String, read-only

Description

The subject string from the **pblocald** certificate. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the **pblocald** certificate subject line

See Also

```
pbclientcertificatesubject, pblogdcertificatesubject, pbmasterdcertificatesubject
```

pblocaldmachine

Data Type

String, read-only

Description

The machine type ID from **uname** on the run host. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the run host machine hardware from the **uname** command

pblocalnodename

Data Type

String, read-only

Description

The nodename from **uname** on the run host. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the run host node name from the **uname** command

pblocaldrelease

Data Type

String, read-only

Description

The operating system release from **uname** on the run host. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the run host operating system version from the **uname** command

pblocalsysname

Data Type

String, read-only

Description

The system name from **uname** on the run host. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the run host operating system implementation string from the **uname** command

pblocaldversion

Data Type

String, read-only

Description

The operating system version from **uname** on the run host. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the run host operating system version string from the **uname** command.

pblogdcertificateissuer

Data Type

String, read-only

Description

The issuer string from **pblogd**'s certificate. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the **pblogd** certificate issuer line

See Also

```
pbclientcertificateissuer, pblocaldcertificateissuer, pbmasterdcertificateissuer
```

pblogdcertificatesubject

Data Type

String, read-only

Description

The subject string from **pblogd**'s certificate. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the **pblogd** certificate subject line

See Also

```
pbclientcertificatesubject, pblocaldcertificatesubject, pbmasterdcertificatesubject
```

pblogdmachine

Data Type

String, read-only

Description

The machine type ID from **uname** on the log server. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the log host machine hardware from the **uname** command.

pblogdnodename

Data Type

String, read-only

Description

The nodename from **uname** on the log server. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the log host node name from the **uname** command.

pblogdrelease

Data Type

String, read-only

Description

The operating system release from **uname** on the log server. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the log host operating system version from the **uname** command.

pblogsysname

Data Type

String, read-only

Description

The system name from **uname** on the log server. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the log host operating system implementation string from the **uname** command.

pblogdversion

Data Type

String, read-only

Description

The operating system version from **uname** on the log server. This value is stored in the event log, but is not available during policy execution.

Valid Values

A string that contains the log host operating system version string level string from the **uname** command.

pbmasterdcertificateissuer

Data Type

String, read-only

Description

The issuer string from the **pbmasterd** certificate.

Valid Values

A string that contains the **pbmasterd** certificate issuer line

See Also

`pbclientcertificateissuer, pblocaldcertificateissuer, pblogdcertificateissuer`

pbmasterdcertificatesubject

Data Type

String, read-only

Description

The subject string from the **pbmasterd** certificate.

Valid Values

A string that contains the **pbmasterd** certificate subject line

See Also

```
pbclientcertificatesubject, pblocaldcertificatesubject, pblogdcertificatesubject
```

pbmasterdmachine

Data Type

String, read-only

Description

The machine type ID from **uname** on the Policy Server host.

Valid Values

A string that contains the Policy Server host machine hardware from the **uname** command.

pbmasterdnodename

Data Type

String, read-only

Description

The node name from **uname** on the Policy Server host.

Valid Values

A string that contains the Policy Server host node name from the **uname** command.

pbmasterdrelease

Data Type

String, read-only

Description

The operating system release from **uname** on the Policy Server host.

Valid Values

A string that contains the Policy Server host operating system version from the **uname** command.

pbmasterdsysname

Data Type

String, read-only

Description

The system name from **uname** on the policy server host.

Valid Values

A string that contains the Policy Server host operating system implementation string from the **uname** command.

pbmasterdversion

Data Type

String, read-only

Description

The operating system from **uname** on the policy server host.

Valid Values

A string that contains the Policy Server host operating system version string level string from the **uname** command.

pbrunmachine

Data Type

String, read-only

Description

The machine type ID from **uname** on the submit host.

Valid Values

A string that contains the submit host machine hardware ID from the **uname** command.

pbrunnodename**Data Type**

String, read-only

Description

The node name from **uname** on the submit host

Valid Values

A string that contains the submit host node name from the **uname** command.

pbrunrelease**Data Type**

String, read-only

Description

The operating system release from **uname** on the submit host.

Valid Values

A string that contains the submit host operating system version from the **uname** command.

pbrunsysname**Data Type**

String, read-only

Description

The system name from **uname** on the submit host.

Valid Values

A string that contains the submit host operating system implementation string from the **uname** command.

pbrunversion**Data Type**

String, read-only

Description

The operating system version from **uname** on the submit host.

Valid Values

A string that contains the submit host operating system version string from the **uname** command.

pbshmachine

- **Version 3.5 and earlier:** **pbshmachine** variable not available
- **Version 4.0 and later:** **pbshmachine** variable available

Data Type

String, read-only

Description

The machine type ID from **uname** on the **pbsh** machine

Valid Values

A string that contains the **pbsh** host machine hardware ID from the **uname** command.

See Also

```
pbshnodename, pbshrelease, pbshsysname, pbshversion
```

pbshnodename

- **Version 3.5 and earlier:** **pbshnodename** variable not available
- **Version 4.0 and later:** **pbshnodename** variable available

Data Type

String, read-only

Description

The nodename from **uname** on the **pbsh** machine

Valid Values

A string that contains the **pbsh** host node name from the **uname** command.

pbshrelease

- **Version 3.5 and earlier:** **pbshrelease** variable not available
- **Version 4.0 and later:** **pbshrelease** variable available

Data Type

String, read-only

Description

The operating system release from **uname** on the **pbsh** machine

Valid Values

A string that contains the **pbsh** host operating system version from the **uname** command.

pbshsysname

- **Version 3.5 and earlier:** **pbshsysname** variable not available
- **Version 4.0 and later:** **pbshsysname** variable available

Data Type

String, read-only

Description

The system name from **uname** on the **pbsh** machine

Valid Values

A string that contains the **pbsh** host operating system implementation string from the **uname** command.

pbshversion

- **Version 3.5 and earlier:** **pbshversion** variable not available
- **Version 4.0 and later:** **pbshversion** variable available

Data Type

String, read-only

Description

The operating system version from **uname** on the **pbsh** machine

Valid Values

A string that contains the **pbsh** host operating system version string from the **uname** command.

X11 Session Capture Variables

The X11 variables are used to capture X Windows sessions.

xwincookie

Data Type

String, read-only

Description

The **xwincookie** variable contains the X Windows Authentication cookie from the client and is available for logging.

There is no run version of this variable.

Valid Values

A string

See Also

```
xwindisplay, xwinproto, xwinforward, xwinreconnect
```

xwinproto

Data Type

String, read-only

Description

The **xwinproto** variable contains the X Windows Authentication protocol from the client and is available for logging.

There is no run version of this variable.

Valid Values

A string

See Also

```
xwncookie, xwindisplay, xwinforward, xwinreconnect
```

xwindisplay

Data Type

String, read-only

Description

The **xwindisplay** variable contains the X Windows Authentication DISPLAY string from the client and is available for logging.

There is no run version of this variable.

Valid Values

A string

See Also

```
xwncookie, xwinproto, xwinforward, xwinreconnect
```

xwinforward

Data Type

Boolean, modifiable

Description

The **xwinforward** variable controls whether Privilege Management for Unix & Linux will forward X Windows applications through to the client X Server.

Syntax

```
xwinforward = Boolean;
```

Valid Values

true	Enable X Windows forwarding. This value is the default.
false	Disable X Windows forwarding.

See Also

```
xwncookie, xwindisplay, xwinproto, xwinreconnect
```

xwinreconnect

Data Type

Boolean, modifiable

Description

The **xwinreconnect** variable contains how Privilege Management for Unix & Linux optimizes X Windows network traffic between **pbrun** and **pblocald**. This optimization involves reconnecting **pblocald** directly to **pbrun** for X Windows forwarding, thus bypassing **pbmasterd** for I/O streams.

Syntax

```
xwinreconnect = Boolean;
```

Valid Values

true	Enable reconnection between pbrun and pblocald . This value is the default.
false	Disable reconnection between pbrun and pblocald .

See Also

```
xwncookie, xwindisplay, xwinproto, xwinforward
```

Built-in Functions and Procedures

The security policy scripting language provides built-in functions and procedures to help simplify security policy implementation. Built-in functions and procedures are stand-alone subroutines that perform specific tasks. The difference between a function and a procedure is that a function returns a value while a procedure does not.

Taking advantage of Privilege Management for Unix & Linux built-in functions and procedures can dramatically speed the implementation time of a company's security policy implementation.

Privilege Management for Unix & Linux built-in functions are divided into the following groups:

- Date and time functions
- File and path functions
- Format and conversion functions
- Input and output functions and procedures
- LDAP functions
- List functions
- Miscellaneous functions
- NIS functions
- Policy environment functions and procedures
- String functions
- Task control procedures
- Task environment functions and procedures
- User and password functions
- PAM policy functions
- Advanced Control and Audit (ACA) procedure

Advanced Control and Audit

Advanced Control and Audit (ACA) provides the ability to control and audit file system activity. The ACA language targets specific actions, such as **open/read/write/exec**, defines whether each action can or cannot be performed on a file, and can also specify the auditing level. The files for each rule are specified using shell-style file patterns to match files.

ACA auditing requires iologging to be enabled for the session. If ACA statements are included in the policy and iologging is not enabled, for versions prior to 10.3, the request proceeds with ACA controls, but without auditing. Beginning in 10.3, if all ACA statements have a log level of 0 (zero), the task continues without logging as before. If any ACA statement contains a loglevel greater than zero, the requested task is rejected with the error: *"1008.02 ACA audit logging requires an iolog to be specified."* ACA only affects the targeted process and child processes and poses no threat to the operating system as a whole. It can also be configured to not apply to specific child processes to ensure that services can be restarted without ACA being applied.

Each specified action is intercepted and processed to determine if the action is allowed and if auditing is required. Where an audit level is specified, the relevant data is sent back to the originating client to be written to an iolog. When ACA is enabled, the iolog contains both iologging and auditing information. The **pbreplay -A (--audit)** command line option is used to display the audit records from an iolog.

When the allowed action is an execute action, the ACA policy is passed-on to the new child task to enable ACA policy to continue to be enforced. This enables complete logging and control over a shell session. For example, Privilege Management for Unix & Linux can be configured to control a bash shell and allow execution of **'vi'** while allowing the user to shell escape to another bash shell or to any other allowed program while still enforcing the ACA policy defined for **'vi'** and all subsequent executions.

ACA should not be used to audit daemons as this results in very large sets of audit data and network traffic and adds little-to-no security to the non-interactive daemon. ACA rules can be specified to disable ACA for daemon launching mechanisms. In the case that a daemon needs to be executed within an ACA controlled shell session and that session is subsequently terminated, the controlling **pbrun** or **pblocald** forks a new process (owned by init) to continue processing ACA auditing.

ACA should also not be used on programs that manipulate logical volumes.

When processing symbolic links, each link in a link chain is evaluated against the ACA policy. If the requested permission is blocked in any part of the chain, the requested permission is denied.

ACA errors such as the inability to read the ACA policy, inability to audit, or out of memory are logged to **syslog** and **stderr**. ACA also uses **pbrestcall** to send any error messages to a policy or log server using the REST interface. On the log server running **pbconfigd**, the keyword **eventdestinations** must be used to send ACA **errlog** data to syslog or to a database.

Example

```
eventdestinations          errlog=syslog chgmt=db
```

To disable central logging, in the policy, set the variable **pbulacacentrallogging** to **0**.

Example

```
pbulacacentrallogging=0;
```

Important Considerations

The ACA is currently enabled for file-specific operations like stat, access, open, read, write, truncate, link, unlink, rename, chmod, and chown. Socket and memory operations are not supported. Furthermore, the ACA does not restrict access to critical operating system files, directories, and devices that are required for normal user activity.

For instance, **read** access to the following locations is protected: **/proc**, **/dev/null**, **/dev/zero**, **/dev/tty**, **/dev/urandom**, terminal, and time zone data.

By default, ACA denies all actions. All allowed actions must be specified explicitly.

Example

If you only have the following ACA rule in the policy:

```
aca("file", "/etc/resolv.conf", "read");
```

Since there is no rule for any other actions, only read actions on **/etc/resolv.conf** will be allowed, all other actions on all other files will be disallowed. With the above rule in the policy:

```
pbrun cat /etc/resolv.conf
```

works; however, the following actions fail even as root:

```
pbrun ls /var
pbrun cat /home/myfile
```



Note: Many simple commands may operate correctly because they perform operations the ACA does not intercept. Commands such as **id**, **date**, **pwd**, and **echo** may not call any file-related functions such as **open()**, thus those commands will work even though it appears ACA should deny all access. Caching daemons may also affect whether the file-related function calls are used. For example, **nscd** may cache user data from **/etc/passwd**, so **'id'** may function without read access to **/etc/passwd**.

ACA allows for the provisioning of a rule to cover other actions **not** specifically matched by the file specifications in subsequent ACA calls. It must be the first ACA rule in the policy. To define this rule you use **unmatched** as the **filespec**, this matches all files not matched by other ACA commands.

Example:

```
aca("file", "unmatched", "all", "DEFAULT Rule");
aca("file", "/etc/*", "!all", "Protect /etc");
```

The first rule provides a default for the **filesystem**, allowing all access to all file actions and for all non-matched files, as long as the **runuser** has the correct file permissions required. The second rule disallows all access, including read, write, rename, chmod, truncate, and open on files in **/etc**.

Other Considerations

- ACA does not apply to **pbksh** and **pbsh**
- ACA has no control over **stdin**, **stdout**, or **stderr**, because they are opened before ACA begins processing.
- Creating links requires ACA read permissions for the existing file, and ACA link permissions for the new link.
- ACA will recognize Privilege Management for Unix & Linux binaries to ensure that we do not get a permissions loop, which is when a process running ACA tries to launch a process with ACA.

- The system fails to work properly if you add the ACA shared libraries to the system `/ect/ld.so.preload` or equivalent file. The ACA shared libraries require policy data read from a file descriptor provided by the parent pbrun or plocald. The system cannot provide that file descriptor (or the PMUL ACA policy), so every binary executed fails.

When ACA is specified and an older client on versions 8.5 and below performs an Optimized Run Mode (ORM) request, the policy server rejects requests.

aca

Description

Trap file system related library calls, such as **open/read/write/exec**, allow, disallow, and audit the calls and specify actions that can or cannot be performed on a file using shell style file patterns to match files. It also specifies an auditing level.

Syntax

```
aca( control_type, filespec, action permissions and auditing [, tag]);
```

Arguments

control_type	Currently always set to file filespec. Shell style file specification which matches one or more files.
filespec	<p>The shell style specification includes wildcards <code>*</code> and <code>?</code>, character classes where <code>[</code> and <code>]</code> delineate a class, and <code>!</code> being the first character in the class negates the other characters in the class, ranges in a class where <code>-</code> between two characters define the range. A <code>-</code> at the beginning or end of a class matches the <code>-</code>, and a <code>]</code> at the beginning of a class matches a <code>]</code>. (See 'man 7 glob' on Linux.) Wildcards, ranges, and classes may appear within any path or file name portion of the filespec, however it must start with a <code>/</code>. For example, <code>*/whoami</code> will not work.</p> <p>Filespecs that begin with a slash <code>/</code> will match all slashes only with a slash (for example, will not match with wildcard expression such as <code>*</code>, <code>?</code>, or <code>[...]</code>). Fully specifying all the slashes in a path protects against, for example, <code>/usr*/bin/date</code> from matching <code>/usr/local/directory1/evil/date</code>.</p> <p>Filespecs that begin with <code>*</code> will allow wildcards to match any slash in the path. This allows for example, <code>*/reboot</code> to match <code>/usr/bin/reboot</code>, <code>/usr/sbin/reboot</code>, <code>/bin/reboot</code>, <code>/sbin/reboot</code>, and <code>/usr/local/bin/reboot</code>.</p> <p>The special filespec unmatched is used to match all files not matched by other filespecs that have been defined.</p> <p>Prior to version 10.3.0, default was used with filespec in the policy. In version 10.3.0 and later, unmatched is used in place of default. For backward compatibility, default will continue to work.</p> <div style="border: 1px solid #007bff; padding: 5px; margin: 5px 0;">  Note: <code>/tmp/banned/*</code> matches files and sub-directories within <code>/tmp/banned</code>, However, access to the directory itself still works. <code>/tmp/banned/</code> disables the whole directory and all contents. </div> <p>Other than "unmatched", the ACA filespec definitions are processed in the order they were defined, and the first match is used; subsequent matches are ignored.</p> <div style="border: 1px solid #007bff; padding: 5px; margin: 5px 0;">  For more information, please see "Important Considerations" on page 212. </div>

action permissions and auditing	<ul style="list-style-type: none"> • One or more of the following action names, separated by the pipe symbol. • Spaces are not allowed in permissions. • The appearance of an action name enables that action. • Preceding the action name with a ! is used to disallow the action. • Each action name may be followed by an optional loglevel, specified as :log=[0-9] before the pipe. • The final log=level applies to action names that do not have individual loglevels. This allows different loglevels for each action name for a given filespec.
Tag	An optional text string used to arbitrarily group, organize, or identify output in the ACA reports.

Action	Description
all	Allow all permissions. The all permission must precede any other permissions.
read	<p>For a normal file, this allows read(). For a directory with read and execute bits set for the runuser, this allows chdir() and opendir(). Note that this affects the ability to open a file or directory with read permissions, however read()s are not intercepted nor audited.</p> <div style="border: 1px solid #1a3d4d; background-color: #e6f2ff; padding: 5px; margin-top: 10px;">  <p>Note: Prior to version 9.4, stat() calls were trapped and audited as part of the "read" permissions. Starting in 9.4, stat() calls are no longer trapped nor audited.</p> </div>
write	For a normal file, this allows open() with create or update, and write() . For a directory, this allows mknod() . Note that this affects the ability to open a file with write permissions, however write() s are not intercepted nor audited.
unlink	For a normal file, this allows unlink() . For a directory, this allows rmdir() .
mknod	This allows mknod()
exec	Allows execution of non-setuid programs that use shared libraries.
execsetuid	Allows execution of setuid binaries on platforms that support LD_PRELOAD with setuid binaries.
execstatic	Allows execution of statically linked binaries (disables ACA for that process and any children)
disable	Disables ACA, upon an exec , for the specified file pattern; and any children of that process. The disable permission should not be used with the unmatched filespec.
chmod	Allows changing of rwx permissions and the sticky bit.
chmodpriv	Allows changing of setuid and setgid permissions
chown	Allows changing of setuid and setgid permissions
link	Allows creation of hard and soft links using link()
owner	Allows above operation only if runuser is the file owner
log=level	Audits access at the specified level (0-9)

- Loglevel zero , or no log=level specified, specifies that no auditing (logging) of the call is performed.
- Loglevel 1 performs the minimal auditing, recording only the call, permission, and path.
- LogLevel 2 indicates that **exec** calls will additionally log the **argv**, and open calls for read, write, or both will additionally log the **device/inode/mode/uid/gid** of the file.
- LogLevel 3 indicates that **exec** calls will additionally log the environment supplied.

ACA can derive a shell's command history by logging additional information. This is enabled with the procedure **enablesessionhistory()**.

Interactions of **exec**, **execstatic**, **execsetuid**:

- **exec** means execution of a dynamically linked **non-setuid** not **setgid** binary is allowed.
- **execstatic** means execution of a statically linked **non-setuid** not **setgid** binary is allowed.
- **execsetuid** means execution of a dynamically linked **setuid/setgid** binary is allowed but not a **nonsetuid/setgid** binary.
- **execstatic|execsetuid** means any **setuid** binary or any static binary including a setuid static binary, a setuid dynamic binary, or a static binary.

In other words, this allows execution of any non-dynamic binary.

- **exec|execstatic|execsetuid** allows any execution.

Return Values

None

Examples

<code>aca("file", "unmatched", "all log=1");</code>	Allows all access to all files not matched by other AC rules, auditing every action at level 1.
<code>aca("file", "/bin/*", "!all");</code>	Disables access of files and subdirectories within /bin, however access to /bin for ls, etc, is still allowed
<code>aca("file", "/bin/", "!all");</code>	Disables all access of /bin and its files and subdirectories. ls, etc, are also not allowed. Auditing is not enabled.
<code>aca("file", "/bin/*", "!all exec:log=2");</code>	Allows exec for all files in /bin. Disallows all other actions for those files. Audits the execs at level 2
<code>aca("file", "/bin/umount", "!all log=9");</code>	Ignored due to above /bin/* pattern
<code>aca('file','unmatched','all:log=1 exec:log=2 execstatic:log=2 execsetuid:log=2','DEFAULT');</code>	
<code>aca('file','/sbin/*','all: log=1 write:log=2 exec:log=2 execstatic:log=2 execsetuid:log=2','Protect sbin files');</code>	
<code>aca("file", "/sbin/lvm", "all disable log=2");</code>	Disable ACA for Linux lvm (note there are more to disable)
<code>aca("file", "/sbin/service", "all disable log=2");</code>	Disable ACA for Linux daemon mechanism

<code>aca("file", "/etc/init.d/*", "all disable log=2"); ;</code>	Disable ACA for Linux daemon mechanisms
<code>aca("file", "...", "...log=2");</code>	When an audit log is requested but not set in the rule, a message is displayed that an iolog must be set in the rule.

Enablesessionhistory

Description

The **enablesessionhistory()** procedure is used to set the internal read-only variable **pbulacasesessionhistory**. This is used for iologged, ACA controlled shell sessions (for example, bash). The **enablesessionhistory()** procedure takes a Boolean argument. Values of **1** or **true** will enable session history. Values of **0** or **false** will disable session history.

When enabled, the ACA preload library will audit additional information for the secured task (presumably a shell), giving **pbreplay** the ability to interpret the shell "history", within certain limitations.

Note that **iolog** must be set, and ACA must be enabled with at least one `aca(...)` statement.

ACA normally exits when it encounters certain errors. When ACA is used only for session history, and no files or operations are blocked, an optional parameter can be used to cause ACA to continue when those errors are encountered. This results in the task being allowed to continue, however the session history recorded will be incomplete.

The relevant portion of the policy should be similar to:

```
aca("file", "default", "all");
enablesessionhistory( true, true);
iolog=<file>;
```

Known limitations

This mechanism cannot capture or reproduce:

- Shell internals, such as if/then/else, while, math, variable setting or testing
- Which builtin was used
- 2>&1 redirection and ordering
- Complex redirection
- Exact quoting of **argv**
- (complex) | (pipelines)
- Exact shell history numbering

This feature adds the new **--history** option to **pbreplay**, to replay the shell's "history" from the `aca iolog`. The **--history** option cannot be used in conjunction with the **-A** option).

Syntax

```
enablesessionhistory( enable_history [, continue_on_error] );
```

Arguments

<code>enable_history</code>	Required Boolean true or 1 to enable or false or 0 to disable.
<code>continue_on_error</code>	Optional true or 1 to enable or false or 0 to disable. Defaults to false .

Example

```
enable_session_history( true );  
enable_session_history( true, true );
```

See also

```
aca ( )
```



For more information about **pbreplay**, please see the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

Date and Time Functions

These functions perform operations and comparisons on dates and times. The following table summarizes the date and time functions.

Table 25. Date and Time Function Summary

Function	Description
<code>datecmp()</code>	Compares two dates and returns the results of the comparison
<code>strftime()</code>	Formats the current date and time, as defined on the Policy Server host, per the supplied format string
<code>timebetween()</code>	Determines if the current time, as defined on Policy Server host, is between time1 and time2 , inclusive

datecmp

Description

The `datecmp()` function compares two dates and returns the results of the comparison.

The two input parameters, **date1** and **date2**, contain the date strings to compare. These fields should have the format **YYYY/MM/DD**, where:

YYYY	A year numeric character string such as 2001 . If the specified year is only two digits, then that value is automatically concatenated with 19 to form a year between 1900 and 1999, inclusive. For example, if the value 01 is supplied for year, the actual year value is processed as 1901.
MM	Month between 1 and 12 inclusive
DD	Day between 1 and 31 inclusive.

Use the forward slash character (`/`) as a field separator. Zeros or spaces can be used as leading pad characters for the year, month, or day.

Syntax

```
result = datecmp (date1, date2);
```

Arguments

date1	Required. Character string containing a date formatted as YYYY/MM/DD
date2	Required. Character string containing a date formatted as YYYY/MM/DD

Return Values

Negative Integer	A negative integer is returned if date1 is less than date2 (date1 < date2).
0	Zero is returned if date1 is equal to date2 (date1 == date2).

Positive Integer

A positive integer is returned if **date1** is greater than **date2** (**date1** < **date2**).

Example

In the example,

```
date1 = "2001/01/21";  
result = datecmp (date1, "2002/01/21");
```

datecmp compares the value in **date1** against the date January 21, 2002. The result is returned in **result**. Because **date1** contains the date **2001/01/21**, the result of **datecmp** is a negative integer because **date1** is less than **date2**.

strftime**Description**

The **strftime()** function formats the current date and time, as defined on Policy Server host, per the supplied format string.



For more information on how to create a format string, please see "[Time Format Commands](#)" on page 78.

Note that different operating systems may provide different options for their own native **strftime()** function. Consult your operating system's **strftime()** manual page for more information.

Syntax

```
result = strftime (formatstring);
```

Arguments**formatstring**

Required. Character string that contains the format command characters that specify how the current date should be formatted

Return Values

strftime() returns a formatted character string containing the current date and time from the Policy Server host.

See Also

```
date, day, dayname, hour, minute, month, time, year
```

timebetween**Description**

The **timebetween()** function determines whether the current time, as defined on the Policy Server host, is between **time1** and **time2**, inclusive.

The **time1** and **time2** parameters contain integer time values. These time values should be specified in military time (**HHMM**) format, where:

HH	A number from 0 to 23 , inclusive, that represents the hour
MM	A number between 0 to 59 , inclusive, that represents the minutes

If **time2 < time1**, the comparison crosses the midnight boundary.

Syntax

```
result = timebetween (time1, time2);
```

Arguments

time1	Required. An integer containing a time value formatted as HHMM
time2	Required. An integer containing a time value formatted as HHMM

Return Values

true	The current time is between time1 and time2 or the current time is equal to either time1 or time2 .
false	The current time is either less time1 or greater than time2 .

Example

In the example,

```
result = timebetween (1100, 1500);
```

the following times set result as follows:

- **08:00** result set to **false**
- **11:00** result set to **true**
- **12:30** result set to **true**
- **15:00** result set to **true**
- **15:01** result set to **false**

File and Path Functions

File and path functions are used to verify, return, and generate information about directories, file paths, names, and file names. The following table summarizes the file and path functions.

Table 26. File and Path Function Summary

Function	Description
<code>access()</code>	Verifies the existence of a path and/or file
<code>basename()</code>	Returns the file name portion of a path
<code>dirname()</code>	Returns the directory portion of a path
<code>logmktemp()</code>	Generates a unique file name on the log host
<code>mktemp()</code>	Generates a unique file name on the Policy Server host
<code>stat()</code>	Returns information about a directory or file

access

Description

The **access()** function verifies the existence of a path and/or file on the Policy Server host. path should contain a fully qualified name, starting with a forward slash character (/).

Syntax

```
result = access (path);
```

Arguments

path Required. String that contains the name of the path and/or file to verify.

Return Values

true	The directory or file exists on the Policy Server host.
false	The directory or file does not exist on the Policy Server host.

Example

In the example,

```
result = access ("/tmp/user.txt");
```

result contains true if **/tmp/user.txt** exists on Policy Server host. result contains false if **/tmp/user.txt** does not exist on the Policy Server host and is not accessible to the superuser.

See Also

```
logmktemp(), mktemp(), stat()
```

basename

Description

The **basename()** function returns the file name portion from the provided path. **basename** actually works by searching the provided string for the rightmost token. A forward slash character (/) delimits tokens. **basename** ignores any number of trailing slash characters.

For example, given the string **/one/two/three**, **basename** returns the rightmost token, which in this case is **three**.

Given the string **/one/two/**, **basename** would ignore the trailing slash and return **two**.

Syntax

```
result = basename (path);
```

Arguments

path Required. Character string containing a file path and file name

Return Values

result contains the rightmost token (that is, the file name) of the supplied character string (that is, the path name). An empty character string ("") is returned if no token is found.

Example

In this example,

```
result = basename ("/var/adm/pblog.txt");
```

result contains the file name **pblog.txt**.

See Also

```
dirname()
```

dirname

Description

The **dirname()** function returns the path component of path. **dirname()** searches the provided string for the rightmost token and returns everything but the rightmost token. Tokens are delimited with the forward slash character (/). **dirname** ignores all trailing slashes.

For example, given the string `/one/two/three`, **dirname** returns everything but the rightmost token. In this example, result contains `/one/two/`.

Given the string `/one/two/three/`, **dirname** ignores the trailing slash and result contains `/one/two`.

Syntax

```
result = dirname (path);
```

Arguments

path	Required. Character string that contains a path and file name
-------------	---

Return Values

result contains the contents of path, minus the rightmost token (that is, the file name). If a token is not found, a `.` is returned.

Example

In the example,

```
result = dirname ("/var/adm/pblog.txt");
```

result contains the directory `/var/adm/`.

See Also

```
basename ()
```

logmktemp

Description

The **logmktemp()** function returns a file name that is guaranteed to be unique on the log host.

This function requires a full path template. Do not save lologs to temp directories.

Syntax

```
result = logmktemp (template);
```

Arguments

template	Required. Character string that contains a file name template. Within template, characters forming a unique identifier replace six trailing X characters. Many, but not all, user systems require precisely six X characters, which must be the trailing characters. Five X character ss, or X character ss in the middle of a template, might work on some systems, but this behavior is not guaranteed.
-----------------	--

Return Values

result contains the generated file name. If a unique file name cannot be generated from template, then **result** contains a blank character string ("").

Example

In this example,

```
result = logmktemp ("/var/adm/iolog.XXXXXX");
```

result contains the file name `/var/adm/iolog.XXXXXX`, where `XXXXXX` is replaced by a unique identifier that is generated by the operating system.

See Also

```
mktemp(), stat()
```

mktemp

Description

The **mktemp()** function returns a file name that is guaranteed to be unique on the Policy Server host.

Syntax

```
result = mktemp (template);
```

Arguments

template

Required. Character string that contains a file name template. Within template, characters forming a unique identifier replace six trailing **X** characters. Many, but not all, user systems require precisely six X characters, which must be the trailing characters. Five X character ss, or X character ss in the middle of a template, might work on some systems, but this behavior is not guaranteed

Return Values

result contains the generated file name. If a unique file name cannot be generated from **template**, **result** contains a blank character string ("").

Example

In the example,

```
result = mktemp ("/var/adm/iologXXXXXX");
```

result contains the file name `/var/adm/iolog.XXXXXX`, where `XXXXXX` is replaced by a unique identifier that is generated by the operating system.



Note: In order to have an I/O log created in this manner, the **iolog** variable must be set to the result of **logmktemp()**. For example:

```
iolog = logmktemp("/var/adm/iolog.XXXXXX");
```

See Also

```
logmktemp(), stat()
```

stat

Description

The **stat()** function returns general information, from the operating system, about the requested file or directory on the Policy Server host. **result** contains an empty list (that is, with length equal to **0**) if the specified file or directory was not found. The **length()** function can be used to determine whether **result** is empty.

Syntax

```
result = stat (path);
```

Arguments

path Required. Character string containing a path and/or file name.

Return Values

result is a list that contains file and/or directory information. Each element in the list contains a different piece of information, as shown below. Each list element is a character string. An empty list is returned (that is, with list length equal to **0**) if the specified file or directory does not exist. If **result** is empty, then the specified path or file was not found.

result elements:

- **result [0]** = file size
- **result [1]** = file owner
- **result [2]** = file group
- **result [3]** = file permissions
- **result [4]** = file access time
- **result [5]** = file creation time
- **result [6]** = file modification time
- **result [7]** = file access date
- **result [8]** = file creation date
- **result [9]** = file modification date
- **result [10]** = file access time in seconds
- **result [11]** = file creation time in seconds

- **result [12]** = file modification time in seconds
- **result [13]** = inode number
- **result [14]** = device number

Example

In the example,

```
result = stat ("/etc");
```

result might contain the following elements:

- `result [0] = 7144`
- `result [1] = bin`
- `result [2] = bin`
- `result [3] = 755`
- `result [4] = 101`
- `result [5] = 101`
- `result [6] = 101`
- `result [7] = 1970/01/01`
- `result [8] = 1970/01/01`
- `result [9] = 1970/01/01`
- `result [10] = 1`
- `result [11] = 1`
- `result [12] = 1`
- `result [13] = 20`
- `result [14] = 2`

See Also

```
access(), length()
```

Format and Conversion Functions

The following table summarizes the format and conversion functions.

Table 27. Format and Conversion Functions

Function	Description
<code>atoi()</code>	Converts a character string to an integer value
<code>sprintf()</code>	Formats the supplied arguments and returns them as a single character string

atoi

Description

The `atoi()` function converts a character string to an integer value.

Syntax

```
result = atoi (string);
```

Arguments

string	Required. Character string that contains the numeric character string to convert to an integer value.
---------------	---

Return Values

result contains the converted integer value.

Example

In this example,

```
result = atoi ("123");
```

result contains the integer value **123**.

sprintf

Description

The `sprintf()` function creates a character string by formatting the supplied arguments according to the formatting commands in a format control string. The resulting character string is returned in **result**.

The format control string controls the formation of the character string that is returned in **result**. It consists of two types of information: actual content and format command characters. The format command characters are used to insert and format the supplied arguments. The number of format command characters in the format control string must match the number of supplied arguments. In

other words, if there are three formatting commands in the format control string, then three function arguments must be supplied. Otherwise, an error is generated.



For more information on format command characters, please see "[Format Commands](#)" on page 77.

Syntax

```
result = sprintf (controlstring [,expression1, ...]);
```

Arguments

controlstring	Required. Character string that contains the format control string that is used to generate the formatted string.
expression1 -	Optional. Character string and integer values to substitute into the format control string.

Return Values

result contains the formatted character string.

Example

In this example,

```
result = sprintf ("System administrator Ids: %s %s %s", "Adm1", "Adm2", "Adm3");
```

the character string **System administrator Ids: Adm1 Adm2 Adm3** is assigned to **result**.

See Also

```
fprintf, print(), printf, syslog
```

Input/Output Functions and Procedures

The following table summarizes Privilege Management for Unix & Linux's input/output functions and procedures.

Table 28. Input/Output Functions and Procedures

Function/ Procedure	Description
<code>fprintf()</code>	Formats and appends a character string to a file
<code>input()</code>	Prompts the user for a single line of input
<code>inputnoecho()</code>	Similar to the <code>input()</code> function, <code>inputnoecho()</code> prompts the user for a single line of input, but does not display the input on the screen as it is entered
<code>print()</code>	Displays a single line of information on the user's screen. The line terminates with the newline character.
<code>printf()</code>	Displays a formatted character string on the user's screen
<code>printnnl()</code>	Similar to the <code>print</code> procedures, <code>printnnl</code> displays a single line of information on the user's screen, but the line is not terminated with the newline character
<code>printvars()</code>	Prints all Privilege Management for Unix & Linux variables to the user's terminal
<code>readfile()</code>	Returns the entire contents of a file in a character string
<code>syslog()</code>	Writes a formatted message to the syslog facility

fprintf

Description

The `fprintf` procedure is similar to the `print` procedure, except that the created formatted character string is appended to a file, rather than being displayed at the user's terminal.

See the discussion on `printf` for a more detailed discussion on how to create use format command characters within the format control string.

Syntax

```
fprintf (filename, controlstring [,expression1, ...]);
```

Arguments

filename	Required. Character string that contains the name of a file. A fully qualified path name, starting with a forward slash character (/).
controlstring	Required. The character string, including format command characters, that is written to filename .
expression1...	Optional. Values to substitute into controlstring , based on the specified format command characters.

Return Values

Because **fprintf** is a procedure, no return value is set.

Example

In this example,

```
fprintf ("/var/adm/pblog.txt", "System administrator Ids: %s %s %s", "Adm1", "Adm2", "Adm3");
```

the character string **System administrator Ids: Adm1 Adm2 Adm3** is appended to the file **/var/adm/pblog.txt**.

See Also

```
print, printf, sprintf(), syslog
```

input

Description

The **input()** function prompts the user for a single line of input. There is no default prompt. If the user attempts to enter more than a single line of input, then the excess input is ignored.

Syntax

```
result = input (prompt);
```

Arguments

prompt Required. Character string that contains the prompt displayed to the user.

Return Values

result is a character string that contains the single line of input that is typed by the user.

Example

In this example,

```
result = input ("Please enter your first and last name:");
```

displays the prompt **Please enter your first and last name:** to the user. The resulting input is stored in **result**.

See Also

```
inputnoecho(), outputredirect, readfile()
```

inputnoecho

Description

The **inputnoecho()** function prompts the user for a single line of input. There is no default prompt. It ignores excess input if the user supplies more than one line of input.

The **inputnoecho()** function works like the **input()** function, except that the input that is typed by the user is not shown on the terminal. This function is useful when prompting the user for a password or other types of confidential information.

Syntax

```
result = inputnoecho (prompt);
```

Arguments

prompt Required. Character string containing the prompt displayed to the user.

Return Values

result is a character string that contains the single line of input that is typed by the user.

Example

In this example,

```
result = inputnoecho ("Please enter your first and last name:");
```

displays the prompt **Please enter your first and last name:** to the user. The resulting input is stored in **result**.

See Also

```
input ()
```

print

Description

The **print** procedure writes one or more expressions to the user's terminal as a single line. The line terminates with a newline character. A comma separates each argument. If an integer is supplied as an argument, then its value is automatically converted to a character string. If a list is supplied, then it prints as a series of quoted strings with the entire series between braces.

The **print** and **printnl** procedures work in the same manner. The only difference is that **print** terminates the generated character string with a newline character, whereas **printnl** does not.

Syntax

```
print (expression1 [, expression2, ...]);
```

Arguments

expression1	Required. A value that is displayed to the user.
expression2, ...	Optional. Additional values that are displayed to the user.

Return Values

Because **print** is a procedure, no return value is set.

Example

In the first example,

```
print ("Your task request has been accepted.", "Thank you.");
```

writes the following to the user's terminal:

```
Your task request has been accepted. Thank you.
```

This line terminates with a newline character.

The second example,

```
TrustedUsers = {"JWhite", "TBrown", "SBlack"};  
print ("The trusted users are:", TrustedUsers);
```

writes the following on the user's terminal:

```
The trusted users are: {"JWhite", "TBrown", "SBlack"}
```

This line terminates with a newline character.

See Also

```
fprintf, outputredirect, printf, printnnl, sprintf(), syslog
```

printf

Description

The **printf** procedure creates a character string by formatting the supplied arguments according to the formatting commands in a format control string. The resulting character string is written to the user's terminal.

The format control string controls the generation of the character string that is written to the user's terminal. It consists of two types of information: actual content and format command characters. The format command characters are used to insert and format the supplied arguments. The number of format command characters in the format control string must match the number of supplied arguments. In other words, if there are three formatting commands in the format control string, then three function arguments are needed. Otherwise, an error is generated.



For more information on format command characters, please see "[Format Commands](#)" on page 77.

Syntax

```
printf (controlstring [,arugment1, ...]);
```

Arguments

controlstring	Required. Character string that contains the format control string that is used to generate the formatted string that is returned in result
argument1 ...	Optional. Character strings and/or integer values to substitute into the formatted string

Return Values

Because **printf** is a procedure, no return value is set.

Example

In this example,

```
printf ("System administrator Ids: %s %s %s\n", "JWhite", "TWhitman", "EPipes");
```

the following string is printed:

```
System administrator Ids: JWhite TWhitman EPipes
```

See Also

`fprintf`, `outputredirect`, `print`, `sprint()`, `syslog`

printfnl

Description

The **printfnl** procedure writes one or more expressions to the user's terminal as a single line. The line does not terminate with a new line character. A space separates each argument.

The **print** and **printfnl** procedures work in the same manner. The only difference being that **print** terminates the generated character string with a newline character, whereas **printfnl** does not.

Syntax

```
printfnl (expression1 [, expression2, ...]);
```

Arguments

expression1	Required. An expression that contains the information to display to the user
expression2 ...	Optional. Additional expressions to display to the user.

Return Values

Because **printnln** is a procedure, no return value is set.

Example

In the example below,

```
printnln ("Your task request has been accepted."); print ("Thank you.");
```

writes the following to the user's terminal:

```
Your task request has been accepted. Thank you.
```

The text that is printed by **printnln** is not terminated with a newline character, so the text that is printed with **print** appears on the same line.

See Also

```
fprintf, outpuredirect, print, printf, sprintf(), syslog
```

printvars

Description

The **printvars** procedure prints all user and Privilege Management for Unix & Linux variables to the user's terminal. This function is often useful when debugging security policy files.

Syntax

```
printvars();
```

Arguments

There are no arguments.

Return Values

Because **printvars** is a procedure, no return value is set.

Example

```
printvars();
```

readfile

Description

The **readfile()** function returns the contents of a file in a character string. Any file type can be processed. The entire file is placed in a single character string. The **length()** function can be used to determine the length of the returned character string.

Additionally, **readfile** will check if the file passed as argument is in the configuration database (**/etc/pb.db**), and if it is, reads the file from the database. If the file is not in the database, **readfile** reverts to check if the file is in the filesystem.

Syntax

```
readfile (filename);
```

Arguments

filename Required. Character string that contains the complete path and file name of the file to read

Return Values

Character string that contains the contents of the specified file

Example

```
result = readfile ("/var/adm/pblog.txt");
```

If the **/path/file** is imported in the config database, then **readfile** gets the file from the config database:

```
# pbadmin -cfg -i /path/file
```

See Also

```
length(), split()
```

syslog

Description

The **syslog** procedure enables you to send diagnostic messages to the **syslog** facility. It creates a character string by formatting the supplied arguments according to the formatting commands in a format control string. The resulting character string is written to the system's **syslog**.

The format control string controls the formation of the character string that is written to the system's **syslog** facility. It consists of two types of information: actual content and format command characters. The format command characters are used to insert and format the supplied arguments. The number of format command characters in the format control string must match the number of supplied arguments. In other words, if there are three formatting commands in the format control string, then three function arguments are required. Otherwise, an error is generated.

Starting with version 7.0.0, as an alternate to the use of **syslog()** function in the policy, you can use the settings **syslog_accept_format**, **syslog_reject_format**, **syslogsession_start_format**, **syslogsession_start_fail_format**, and **syslogsession_finished_format** in the **pb.settings** file. These settings format **syslog** messages for Accept and Reject events, and the session events Start, Finish, and Start_Fail.

i For more information about these settings, please see *Customized Syslog Formatting* in the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

i For more information on format command characters, please see "Format Commands" on page 77.

Syntax

```
syslog (controlstring [,expression1, ...]);
```

Arguments

controlstring	Required. Character string that contains the control string that is used to generate the formatted string that is passed to the syslog facility
expression1 ...	Optional. Expressions to substitute into the formatted string

Return Values

Because **syslog** is a procedure, no return value is set.

Example

In this example,

```
syslog ("System administrator Ids: %s %s %s", "Adm1", "Adm2", "Adm3");
```

the message

```
System Administrator Ids: Adm1 Adm2 Adm3
```

is written to **syslog** (the **syslog** daemon, typically **syslogd**, and Privilege Management for Unix & Linux must be configured for this to work).

See Also

```
fprintf, print, printf, sprintf(), PowerBroker syslog setting
```

LDAP Functions

Privilege Management for Unix & Linux LDAP support is based on the LDAP version 2 API, as defined in RFC 1823. Specific parts of the LDAP API are mapped to a series of Privilege Management for Unix & Linux functions.

The following table summarizes the Privilege Management for Unix & Linux LDAP functions.

Table 29. LDAP Function Summary

Function	Description
<code>ldap_attributes()</code>	Returns the attributes that are associated with an LDAP entry
<code>ldap_bind()</code>	Binds an open LDAP connection to a user
<code>ldap_dn2ufn()</code>	Converts a DN to a user-friendly naming format
<code>ldap_entry_count()</code>	Returns the number of entries that are returned by an LDAP search
<code>ldap_explodedn()</code>	Returns the components of a DN in a list
<code>ldap_firstentry()</code>	Returns the first entry that is returned by a search
<code>ldap_getdn()</code>	Returns the DN of an LDAP entry
<code>ldap_getvalues()</code>	Returns values that are associated with an LDAP entry
<code>ldap_init()</code>	Connects to an LDAP server Version 3.5 and earlier: function available Version 4.0 and later: function deprecated
<code>ldap_nextentry()</code>	Returns the next entry that is returned by a search
<code>ldap_open()</code>	Opens a connection to an LDAP server
<code>ldap_search()</code>	Opens a connection to an LDAP server
<code>ldap_search()</code>	Searches an LDAP tree
<code>ldap_unbind()</code>	Unbinds and disconnects a connection from an LDAP directory

Perform an LDAP Search

The general process for performing an LDAP search is outlined below.



For more information on using LDAP, refer to your LDAP documentation.

1. Use the `ldap_open()` function to establish an LDAP server connection.
2. Bind the LDAP server connection to the user by using the `ldap_bind()` function.
3. Use the function `ldap_search()` to search an LDAP directory.
4. Use the `ldap_entry_count()` function to determine the number of entries that were found by the query.
5. Loop through the entries that were found by the query by using the `ldap_firstentry()` and `ldap_nextentry()` functions.

6. Use the function `ldap_attributes()` to obtain a list of attributes that are available for an entry.
7. Use the `ldap_getvalues()` function to retrieve the actual attribute values that are associated with an entry.
8. Process the next entry. Repeat steps 5 through 7 until all entries are processed.
9. Use the function `ldap_unbind()` to unbind and close the LDAP Server connection.

ldap_attributes

Description

The `ldap_attributes()` function returns a list that contains all of the attributes that are associated with the specified LDAP entry. Each element in result contains an attribute name.

Syntax

```
result = ldap_attributes (LDAPEntry);
```

Arguments

LDAPEntry	Required. A unique LDAP entry that is generated by <code>ldap_firstentry()</code> , <code>ldap_nextentry()</code> , or <code>ldap_search()</code> .
------------------	---

Return Values

A list in which each element contains an attribute name. On error, it returns an empty list.

Example

In the example

```
result = ldap_attributes (LDAPEntry);
```

result might look like {"firstname", "lastname", "department", "jobcode"}.

See Also

```
ldap_firstentry(), ldap_nextentry(), ldap_search()
```

ldap_bind

Description

The `ldap_bind()` function binds an existing LDAP server connection using the specified DN and password. If the DN is not specified, an anonymous bind is attempted.

Syntax

```
result = ldap_bind (ConnectionId, dn [,Password]);
```

Arguments

ConnectionId	Required. LDAP server connection that is generated by the <code>ldap_open()</code> function
dn	Required. User's DN. May be an empty string
Password	Optional. String that contains the password for <code>dn</code> .

Return Values

0	Bind operation successful
1	Bind operation failed

Example

In this example,

```
result = ldap_bind (ldapConnection, "");
```

an anonymous bind is performed using the LDAP server connection that is specified in `ldapConnection`.

See Also

```
ldap_open(), ldap_unbind()
```

ldap_dn2ufn

Description

The `ldap_dn2ufn()` function converts the supplied DN into a more user-friendly form by stripping off the type names. The resulting character string is returned in `result`.

Syntax

```
result = ldap_dn2ufn (dn);
```

Arguments

dn	Required. A string that contains a DN (Distinguished Name)
-----------	--

Return Values

string	A character string that contains a DN name with type names removed
Empty string	Error

Example

In this example,

```
result = ldap_dn2ufn (dn);
```

result contains the specified DN name without type names.

See Also

```
ldap_explodedn()
```

ldap_entry_count

Description

The **ldap_entry_count()** function returns the number of entries that exist in a specific LDAP message. The **ldap_search()** function generates **LDAPEntry**.

Syntax

```
result = ldap_entry_count (LDAPEntry);
```

Arguments

LDAPEntry	Required. LDAP message that is generated by ldap_search() .
------------------	--

Return Values

integer	The number of entries that are contained in the specified LDAP message
0	If zero entries or on error

Example

In this example,

```
result = ldap_entry_count (LDAPEntry);
```

result contains the number of entries in the LDAP message that is identified by **LDAPEntry**.

See Also

```
ldap_search()
```

ldap_explodedn

Description

The **ldap_explodedn()** function splits the supplied DN into its separate sub-components. Each sub-component is called a relative distinguished name (RDN).

The **notypes** argument specifies whether the RDNs are returned with only values or both values and attributes. Setting **notypes** to **false** returns both values and attributes. Setting **notypes** to **true** will return only values.

The RDNs are returned in a list. If only values were requested, then each list element will contain one value. If both values and attributes have been requested, each result list element will have the format "**attribute=value**".

Syntax

```
result = ldap_explodedn (dn, notypes);
```

Arguments

dn	Required. A string that contains a Distinguished Name (DN).
notypes	Required. An integer that represents a true or false value.

Return Values

result is a list containing the DN sub-components (that is, the RDNs). If only values were requested, then the list has the following format:

```
{"value", "value", ...}
```

If both values and attributes have been requested, then the list will have the following format:

```
{"attribute=value", "attribute=value", ...}.
```

Example

In this example,

```
result = ldap_explodedn (dn, false);
```

result would be a list containing DN sub-components. Both values and attributes are returned in this case.

See Also

```
ldap_dn2ufn ()
```

ldap_firstentry

Description

The **ldap_firstentry()** function returns the first entry in the specified LDAP message that is returned from **ldap_search()**.

The first entry message is needed to retrieve successive entries from the specified LDAP message by using the **ldap_nextentry()** function.

The **ldap_firstentry()** function does not retrieve values. It returns a unique entry. The result can be used in a function such as **ldap_getvalues()** to actually retrieve attribute values.

Syntax

```
result = ldap_firstentry (LDAPEntry);
```

Arguments

LDAPEntry	Required. LDAP message. ldap_search() generates LDAP messages.
------------------	---

Return Values

LDAPEntry	An LDAP entry
------------------	---------------

Empty String	Error
---------------------	-------

Example

```
result = ldap_firstentry (LDM);
```

See Also

```
ldap_nextentry(), ldap_search()
```

ldap_getdn

Description

The **ldap_getdn()** function returns the DN for the specified LDAP entry.

Syntax

```
result = ldap_getdn (LDAPEntry);
```

Arguments

LDAPEntry	Required. An LDAP entry. <code>ldap_firstentry()</code> , <code>ldap_nextentry()</code> , and <code>ldap_search()</code> generate LDAP entries.
------------------	---

Return Values

string	A DN.
Empty string	Error condition

Example

```
result = ldap_getdn (LDAPEntry);
```

See Also

```
ldap_firstentry(), ldap_nextentry(), ldap_search()
```

ldap_getvalues

Description

The `ldap_getvalues()` function returns the values that are associated with the specified attribute. The values are returned in a list where each list element represents a value. The `length()` function can be used to determine the number of elements that are returned in result. If `ldap_getvalues()` is successful, result has the format {"value", "value", ...}.

The `ldap_getvalues()` function is typically used after a call to `ldap_search()`, `ldap_firstentry()`, or `ldap_nextentry()` to retrieve attribute values for the entry that is currently being processed.

Syntax

```
result = ldap_getvalues (LDAPEntry, attributeName);
```

Arguments

LDAPEntry	Required. An LDAP entry that is created by <code>ldap_firstentry()</code> , <code>ldap_nextentry()</code> , or <code>ldap_search()</code> .
attributeName	Required. String that identifies the attribute for which a value should be returned.

Return Values

list	If successful, then a list of character strings is returned. Each element in the list will contain a value.
empty list	Error condition, list length is set to zero.

Example

```
result = ldap_getvalues (LDAPEntry, "uid");
```

See Also

```
ldap_firstentry(), ldap_getattributes, ldap_nextentry(), ldap_search()
```

ldap_init

- **Version 3.5 and earlier:** `ldap_init()` function available
- **Version 4.0 and later:** `ldap_init()` function deprecated

Description

Initializes a connection to an LDAP database. This function supersedes `ldap_open()` and `ldap_init()`.

Syntax

```
ldap_initialize (ldap_url [, 2 | 3])
```

Arguments

ldap_url	Required, string. An LDAP URL pointing to the desired LDAP database.
version	Optional, number. The LDAP database version. Either a 2 or 3. If the version is not included, then a version 2 connection is created.

Return Values

On success, an LDAP Connection is returned. On failure, null is returned.

Example

```
connection = ldap_initialize("ldap://ldaphost");
```

See Also

```
ldap_init(), ldap_open()
```

ldap_nextentry

Description

The `ldap_nextentry()` function returns the next LDAP entry in the specified LDAP message.

The `ldap_nextentry()` function does not retrieve values. It returns a unique entry. The result can be used in a function like `ldap_getvalues()` to actually retrieve attribute values.

Syntax

```
result = ldap_nextentry (LDAPEntry);
```

Arguments

LDAPEntry	Required. An LDAP entry that is returned by the previous <code>ldap_firstentry()</code> or <code>ldap_nextentry()</code> .
------------------	--

Return Values

LDAP_Entry	An LDAP entry.
empty string	Error condition.

Example

```
result = ldap_nextentry (LDAPEntry);
```

See Also

```
ldap_firstentry(), ldap_search()
```

ldap_open

Description

The `ldap_open()` function establishes a connection to the LDAP server that is specified in `ServerName`. The connection is made through the port number in `port` (if specified).

Syntax

```
result = ldap_open (ServerName [,port]);
```

Arguments

ServerName	Required. Character string that contains the host name of an LDAP server.
port	Optional. Integer that contains a port number. The default port number is 389.

Return Values

LDAP_Connection	If the open operation is successful, an LDAP server connection is returned in result .
------------------------	---

Example

In the example

```
result = ldap_open ("mycompany.ldap.server1", 200);
```

if the open operation is successful, **result** contains an LDAP server connection ID for **mycompany.ldap.server1** on port **200**. If the connection is not successful, **result** contains a null string.

See Also

```
ldap_bind(), ldap_initialize(), ldap_unbind()
```

ldap_search

Description

The **ldap_search()** function searches the LDAP directory below the baseDN, using the search criteria that are specified in the search filter. The scope argument defines the scope, or boundaries, of the search.

Syntax

```
result = ldap_search (ConnectionId, baseDN, scope, searchfilter, attributeList, attributeFlag);
```

Arguments

ConnectionId	Required. LDAP Server Connection
baseDN	Required. String that contains the base DN for the search
scope	Required. String that contains a search scope value. Value entries are subtree (search the baseDN and the entire directory below), onelevel (search the baseDN and one level below), and base (search the baseDN only).
searchfilter	Required. String that contains search criteria
attributeList	Required. List that identifies the attributes that should be returned. Each list element must be an attribute name. An empty list defaults to all attributes.
attributeFlag	Required. Integer that represents either true or false . If set to true , only attribute types are returned. If set to false , both attribute types and values are returned.

Return Values

LDAP message	The search operation was successful.
empty string	Unsuccessful search

Example

```
result = ldap_search (ConnectionId, "dc=beyondtrust, "dc=com", subtree", "jobcode=mgr", {}, 0);
```

See Also

```
ldap_attributes(), ldap_entry_count(), ldap_firstentry(), ldap_getvalues(), ldap_nextentry()
```

ldap_unbind

Description

The **ldap_unbind()** function unbinds and closes an existing LDAP server connection.

Syntax

```
result = ldap_unbind (LDAP_Connection);
```

Arguments

LDAP_Connection	Required. An LDAP Server Connection that was created by ldap_open() .
-----------------	--

Return Values

0	Unbind operation successful.
-1	Unbind operation failed

Example

In this example,

```
result = ldap_unbind (ldapConnection);
```

an unbind and close are performed on the LDAP server connection ID specified in **ldapConnection**.

See Also

```
ldap_bind(), ldap_open()
```

List Functions

The following table summarizes the available Privilege Management for Unix & Linux list functions.

Table 30. List Functions

Function	Description
<code>append()</code>	Creates a new list by appending one or more strings or lists to the end of another list
<code>insert()</code>	Creates a new list by inserting additional strings or lists into a specific position (indicated by an integer index) in the original list
<code>join()</code>	Creates a new string by concatenating each element of a specified list separated by a delimiter character. This is the opposite of the <code>split()</code> function.
<code>length()</code>	Returns the number of elements in a list
<code>range()</code>	Creates a new list from a specific range of elements from an existing list
<code>replace()</code>	Creates a new list by deleting a specific range of elements from an existing list. Replacement elements can be inserted into the new list in positions where original elements were deleted.
<code>search()</code>	Searches a list for a specific pattern
<code>split()</code>	Creates a new list by splitting the contents of a string into individual list elements. This is the opposite of the <code>join()</code> function.

append

Description

The `append()` function creates a new list by concatenating the supplied arguments to the end of `list1` in sequential order.

Syntax

```
result = append (list1, list-or-string1 [,list-or-string2, ...]);
```

Arguments

<code>list1</code>	Required. Contains the list to which the specified arguments are appended.
<code>list-or-string1</code>	Required. Contains either a character string or a list. This argument is appended to <code>list1</code> .
<code>list-or-string2 ...</code>	Optional. Contains additional character strings and/or lists. These additional arguments are appended to <code>list1</code> .

Return Values

The newly created list

Example

In this example,

```
TrustedUsers = {"JWhite", "TBrown", "SBlack"};
NewList = append (TrustedUsers, "RRoads");
```

result contains the following list {"JWhite", "TBrown", "SBlack", "RRoads"}.

In the second example,

```
List1 = {"JWhite", "TBrown"};
List2 = {"SBlack", "RRoads"};
NewList = append (List1, "RGreen", List2);
```

result contains:

```
{"JWhite", "TBrown", "RGreen", "SBlack", "RRoads"}
```

See Also

```
insert(), join()
```

insert

Description

Returns a list constructed by inserting the strings or lists into a specific position (indicated by an integer index) in the specified list. Note that **0** is the start of the list, **1** is between the first and second elements in the list, and so on.

If you specify an index number that is larger than the specified list, then the strings are placed at the end of the list.

Syntax

```
result = insert (list, index, list-or-string1 [, list-or-string2, ...])
```

Arguments

list	Required. The original list
index	Required. The integer index
list-or-string1	Required. The list or string to insert
list-or-string2	Optional. The subsequent list(s) or string(s) to insert

Return Values

A list

Example

Using

```
trustedusers={"jamie", "cory", "tom"};
a=insert(trustedusers, 1, "leslie");
```

sets to the list:

```
{"jamie", "leslie", "cory", "tom"}
```

See Also

```
append(), join(), replace()
```

join

Description

The **join()** function creates a string by concatenating all of the elements in a list. The specified delimiter character separates each element in the generated string. If a delimiter character is not specified, then a blank is used as the delimiter.

Syntax

```
result = join (list [,delimiter]);
```

Arguments

list	Required. The list whose elements are to be concatenated into a new character string.
delimiter	Optional. If specified, the delimiter character is used as a separator character between list elements as they are concatenated together.

Return Values

result Contains the new character string

Example

In this example,

```
TrustedUsers = {"Fred", "John", "George"};
NewString = join (Trustedusers, ",");
```

NewString contains the character string: **Fred, John, George.**

See Also

```
split()
```

length

Description

The **length()** function returns the number of elements in the specified list. The index number for the first element in a list is always **0**. The index number for the last list element is always the list length - **1**.

Syntax

```
result = length (list1);
```

Arguments

list1 Required. The list for which the number of elements is determined

Return Values

result Contains the number of elements in **list1**

Example

In this example,

```
list1 = {"Fred", "George", "Sally"};
result = length (list1);
```

result would contain the integer value **3**.

See Also

```
append(), insert(), join(), range(), split()
```

range

Description

The **range()** function generates a new list from the elements in a list, starting at the element number that is specified by **index1** and ending with the element number that is specified by **index2**.

The first element in a list always has an index value of **0**. An index number that is larger than the last index in the list is treated as the last element. In the case where **index1** is larger than the last index in the list, an empty list is returned (that is, with a list length equal to **0**).

Syntax

```
result = range (list1, index1, index2);
```

Arguments

list1	Required. The list from which a new list is extracted
index1	Required. The element number, in list1 , at which the extraction should begin
index2	Required. The element number in list1 at which the extraction should end, inclusive

Return Values

result Contains the new list that was extracted from **list1**

Example

In this example,

```
list1 = {"JWhite", "SBrown", "RRoads"};  
result = range (list1, 1, 2);
```

result contains the following list:

```
{"SBrown", "RRoads"}
```

See Also

```
append(), insert(), join(), length(), replace(), split()
```

replace

Description

The **replace()** function replaces elements in a list, thereby creating a new list. The list elements in the specified range are deleted and those that are specified by the string arguments are inserted in their place. If replacement arguments are not supplied, then the appropriate elements are deleted without being replaced.

Syntax

```
result = replace (list1, index1, index2 [,string1...]);
```

Arguments

list1	Required. The list from which list elements are removed, and optionally, replaced by new elements
--------------	---

index1	Required. The first element in the range of elements to delete or replace
index2	Required. The last element in the range of elements to delete or replace
string1...	Optional. The character string(s) that will replace the list elements that are being deleted

Return Values

result	Contains the new list that is created by deleting or replacing elements from the original list
---------------	--

Example

In this example,

```
list1 = {"Adm1", "Adm2", "Adm3", "Adm4"};
result = replace (list1, 2, 3, "SysAdm1", "SysAdm2");
```

result contains the following list:

```
{"Adm1", "Adm2", "SysAdm1", "SysAdm2"}
```

See Also

```
append(), insert(), join(), length(), range(), split()
```

search

Description

The **search()** function searches a list for the first element that is found to match a specific pattern. The search is case sensitive and wildcard characters can be used within the pattern.

 For more information on using wildcard characters, please see "[Wildcard Search Characters](#)" on page 82 and "[quote](#)" on page 266.

Syntax

```
result = search (list1, pattern);
```

Arguments

list1	Required. The list to search.
pattern	Required. The pattern to search for.

Return Values

An integer value is returned. If a match is found, then **result** contains the element number of the first pattern match in the list. If no match is found, result is set to **-1**.

Example

In this example,

```
list1 = {"ADM1", "ADM2", "ADM3", "SYSADM1", "SYSADM2", "USER1", "USER2"};
result = search (list1, "SYS*");
```

result is set to **3** as **list1[3]** is the first element in the list to match the search pattern.

See Also

```
append(), insert(), join(), length(), range(), replace()
```

split

Description

The **split()** function creates a list from a string. The string is broken up into separate list elements based on the characters in the specified delimiter string. If a delimiter string is not specified, then a string containing space, tab (**\t**), and newline (**\n**) is used. If none of the delimiter characters are encountered, then a list that contains one element (that is, the entire string) is returned.

Syntax

```
result = split (string1[,delimiter[,omit_empty_elements]]);
```

Arguments

string1	Required. The string to separate into list elements
delimiter	Optional. The delimiter string that is used to break the string into separate elements. If delimiter is not specified, then \t\n is used as the delimiter string
omit_empty_elements	Optional. Boolean value that determines whether empty elements of the resulting list are omitted (true) or included (false). If omit_empty_elements is not specified, it defaults to true

Return Values

result contains the new list.

Example

In this example,

```
UserList = "user1,user2,user3,,user4";  
result = split (UserList,",");
```

result contains the following list:

```
{"user1", "user2", "user3", "user4"}
```

In this example,

```
UserList = "user1,user2,user3,,user4";  
result = split (UserList,",",false);
```

result contains the following list:

```
{"user1", "user2", "user3", "", "user4"}
```

See Also

```
append(), insert(), join(), length(), range(), replace()
```

Miscellaneous Functions and Procedures

Miscellaneous functions and procedures (refer to the following table) do not fit into any other category.

Table 31. Miscellaneous Functions and Procedures

Function/ Procedure	Description
egrep()	Runs the policy server host's egrep() command using the provided arguments and files, and returns the result as a string. Version 4.0 and earlier: function not available Version 5.0 and later: function available
fgrep()	Runs the policy server host's fgrep command using the provided arguments and files, and returns the result as a string. Version 4.0 and earlier: function not available Version 5.0 and later: function available
glob()	Matches a string to a pattern.
grep()	Runs the policy server host's grep command using the provided arguments and files, and returns the result as a string. Version 4.0 and earlier: function not available Version 5.0 and later: function available
iologcloseaction runhost()	Executes a specified program on the runhost when the session is ended and the iolog is closed. Version 9.3 and earlier: procedure not available Version 9.4 and later: procedure available
ipaddress()	Returns a machine's IP address
isset()	Checks a variable to see if it has a value
quote()	Encloses a string in quotation marks
remotesystem()	Runs a command on a specified Privilege Management for Unix & Linux runhost
runtimewarn()	Warns the user on stderr that the session has exceeded the time limit. Version 9.3 and earlier: procedure not available Version 9.4 and later: procedure available
runtimewarnlog()	Records to logserver's syslog that a user's session has exceeded the time limit Version 9.3 and earlier: procedure not available Version 9.4 and later: procedure available
system()	Runs a command
unset	Removes temporary variables from the event and I/O log files

egrep

- **Version 4.0 and earlier:** **egrep()** function not available
- **Version 5.0 and later:** **egrep()** function available

Description

The **egrep()** function runs the policy server host's **egrep()** command using the provided arguments and files, and returns the result as a string.

Syntax

```
egrep ([egrep-arguments, ] search-pattern, filename-or-template [, filename-or-template ...]);
```

Arguments

egrep-arguments	Optional. Switch arguments to the policy server host's egrep command. Refer to the policy server host's grep documentation for specifics.
search-pattern	Required. The regular expression to search for.
filename-or-template	Required. A file name, possibly with wildcards, to search for the search-pattern .

Return Values

A string that contains the output of **egrep()**.

Example

```
result = egrep ("-w", "word", "filename");  
result = egrep ("pattern", "manynames*");
```

See Also

```
fgrep (), grep ()
```

fgrep

- **Version 4.0 and earlier:** **fgrep()** function not available
- **Version 5.0 and later:** **fgrep()** function available

Description

The **fgrep()** function runs the policy server host's **fgrep** command using the provided arguments and files, and returns the result as a string.

Syntax

```
fgrep ([fgrep-arguments, ] search-pattern, filename-or-template [, filename-or-template ...]);
```

Arguments

fgrep-arguments	Optional. Switch arguments to the policy server host's fgrep command. Refer to the policy server host's fgrep documentation for specifics.
search-pattern	Required. The regular expression to search for.
filename-or-template	Required. A file name, possibly with wildcards to search for the search-pattern .

Return Values

A string that contains the output of **fgrep**.

Example

```
result = fgrep ("-w", "word", "filename");
result = fgrep ("pattern", "manynames*");
```

See Also

```
egrep (), grep ()
```

glob

Description

The **glob()** function searches a character string for a specific shell-style pattern. **glob()** is often used to match patterns to file names because the patterns that are used are the same patterns that are used by the Unix/Linux shell file name matching algorithms.



For more information on creating search patterns, please see "[Wildcard Search Characters](#)" on page 82 and "[quote](#)" on page 266.

Syntax

```
result = glob (pattern, string);
```

Arguments

pattern	Required. The search pattern
string	Required. The string to search

Return Values

true	A pattern match was found
false	A pattern match was not found

Example

```
result = glob (pattern, logfilename);
```

grep

- **Version 4.0 and earlier:** **grep()** function not available
- **Version 5.0 and later:** **grep()** function available

Description

The **grep()** function runs the policy server host's **grep** command using the provided arguments and files, and returns the result as a string.

Syntax

```
grep ([grep-arguments, ] search-pattern, filename-or-template [, filename-or-template ...]);
```

Arguments

grep-arguments	Optional. Switch arguments to the policy server host's grep command. Refer to the policy server host's grep documentation for specifics.
search-pattern	Required. The regular expression to search for.
filename-or-template	Required. A file name, possibly with wildcards, to search for the search-pattern .

Return Values

A string containing the output of **grep**.

Example

```
result = grep ("-w", "word", "filename");  
result = grep ("pattern", "manynames*");
```

See Also

```
egrep(), fgrep()
```

iologcloseaction

Description

iologcloseaction() is used to specify a program to be executed on the Logserver (or policy server, if no logserver) when an iolog is closed.

This can be used for example to execute scripts that can send IOlog or ACA data to splunk or other systems. When Privilege Management for Unix & Linux is installed, an example Perl script called **closeactionsplunk.pl**, that sends ACA data from the IOlog to Splunk is installed in **/opt/pbul/scripts**.

Note that unlike the **iologcloseactionrunhost()** procedure, this does not include the ability to specify **runuser**, **runcwd**, **environment**, **timeout**, or command line arguments.

IOLogs with a **closeaction** specified, or when Solr is used, are placed in a queue, rather than acted upon immediately.

pbconfigd monitors the queue and launches **pbreplay** to handle both Solr and **iologcloseaction** activity.

Syntax

```
iologcloseaction( command );
```

Arguments

command	<p>Required string specifying the /full/path/to/external/program</p> <p>The syntax for the script or program must be /path/to/external/program/path/to/iolog.log</p> <p>The program should exit 0 if successful, should exit 255 (or -1) to have Privilege Management for Unix & Linux log that the script failed, and should exit 254 (-2) to have Privilege Management for Unix & Linux re-queue the item and have the queue mechanism pause. This can be used for example, to indicate that a destination host is not reachable, and additional closeaction activity should not take place immediately.</p>
----------------	--

Example

```
iologcloseaction("/opt/pbul/scripts/closeactionsplunk.pl");
```

See Also

iolog variable, Iologcloseactionrunhost procedure



For more information, please see the following sections in the [Privilege Management for Unix & Linux System Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>:

- **iologactionqueuetimelimit**
- **pseudo_iologcloseaction**



- `iologactionmaxprocs` keywords
- `pbdbutil --iologidx`

iologcloseactionrunhost

Description

`iologcloseactionrunhost()` is used to specify a `/path/filename` to be executed on the runhost when the iolog is closed. The specified `/path/filename` can be a shell script or binary. The user to run the program as, environment, arguments, and working directory are specified in the function call. `Stdin`, `stdout`, `stderr` are redirected to `/dev/null`. The timeout (specified in seconds) is mandatory. A timeout value of zero indicates no timeout. Note that a timeout value greater than zero will cause the end user's invocation of `pbrun` to pause while the close action takes place or until the timeout expires. Any runtime errors such as invalid user, `cwd`, or command are logged via `syslog`, and to the appropriate Privilege Management for Unix & Linux log (for example, `pbrunlog`, `pblocaldlog`) if specified in `pb.settings`.

Syntax

```
iologcloseactionrunhost( user, environment, timeout, cwd "/path/command and arguments");
```

Arguments

User	The user to run the command. This user must exist on the runhost.
Environment	ENV settings to execute the command with. If an empty list is specified, <code>su -</code> is used to create a login environment.
Timeout	Required integer. When set to <code>0</code> , no timeout is used, and the specified command could potentially run forever. When set to <code>> 0</code> , specified the number of seconds for a timeout. If the timeout is reached, the command is terminated using <code>SIGTERM</code> , and if needed, by a <code>SIGKILL</code> .
Cwd	Required string to specify the working directory. Note that with an empty environment list, this directory may be changed via the login shell.
command	Required string specifying the fully qualified command, and its arguments. This is passed to <code>su</code> using <code>su's -c</code> option.

Examples

```
iologcloseactionrunhost( "jsmith", {"PATH=/bin", "TMPDIR=/tmp/", "PBUL=PBULTEST"}, 20, "/tmp",
"/usr/local/bin/closeaction -a -b" );
```

```
iologcloseactionrunhost( "root", {}, 0, "/tmp", "/usr/local/bin/closeaction -a -b" );
```

See also

iolog variable

ipaddress

Description

The **ipaddress()** function returns the IP address of the machine that is specified by `hostname`. `hostname` should be a fully qualified machine name.

Syntax

```
result = ipaddress (hostname);
```

Arguments

hostname Required. A fully qualified host name

Return Values

result contains the IP address of the specified machine. If the IP address cannot be determined, a blank string is returned (that is, **length = 0**).

Example

In this example,

```
result = ipaddress (hostname);
```

result would contain the IP address of the machine specified in **hostname**.

isset

Description

The **isset()** function determines whether a variable has been set. A variable with a blank or zero value returns **true**, because blank and zero are considered values.

Syntax

```
result = isset (string);
```

Arguments

string Required. A string that contains a variable name

Return Values

true	Integer. The specified variable has a value
false	Integer. The specified variable does not have a value

Example

In this example:

```
runhost = "beyondtrust1";
result = isset ("runhost");
```

result contains an integer value of 1 (**true**) because the **runhost** variable has a value of **beyondtrust1**.

See Also

unset

policytimeout

Description

The new Privilege Management for Unix & Linux 8.0.2 **policytimeout()** procedure adds an overall policy timeout mechanism so that **pbmasterd** can abort the request when the policy processing takes an inordinate amount of time.

For example, when **submitconfirmer()** is used, but the submitting user (or process) does not enter a password.

This will prevent **pbmasterd** processes that appear to be nonresponsive when the policy is waiting for user input which may never arrive. When the policy timeout is encountered, the request is rejected, with the **exitstatus** set to:

```
policy timeout (<seconds> seconds) reached for <submitting user> on host <submithost> for command
<command and args>
```

That message will also be logged to **pbmasterd.log**.

This timeout mechanism terminates **pbmasterd** any time that the policy processing takes longer than the timeout value specified.

This includes any user input functions, infinite loops, long running external programs run with **system()** and **remotesystem()**, DNS and NFS hangs, and lengthy policies.

When the **policytimeout()** procedure is called at the beginning of the policy it will apply to the entire policy. If called later, it will apply to the rest of the policy.

If the function is not called, or called with a value of **0**, there will be no timeout and **pbmasterd** will process the entire policy (including waiting for user input) before terminating.

The **policytimeout()** procedure can be called many times, each time overriding the value previously set.

This timeout is canceled when an accept or reject is encountered (for example, the policy is completed). Note that this timeout does not affect the **runconfirmer** mechanism, which is processed after an accept. This timeout does not affect the secured task once accepted. For example, this cannot protect against a user not providing username/password input for `pbrun telnet <host>`.

pbmasterd informs Privilege Management for Unix & Linux 8.0.2 clients (**pbrun**, **pbksh**, **pbsh**, **pbssh**) of the timeout, and those clients will also timeout. Note that the exact timing of **pbmasterd** timing out and the client timing out is not exact.

pbmasterd and the client process the timeout independently, and either may terminate before the other. Older clients cannot process such a timeout, and may appear nonresponsive when **pbmasterd** terminates during expected user input. **pbmasterd** does not have a mechanism to interrupt an older client that is expecting input.

When **remotesystem()** is used with the **submithost**, the policy timeout is independent of the timeout specified in the **remotesystem** function call. The first of those timeouts to be encountered will be the one processed.

When **remotesystem()** is used with a host other than the **submithost**, only the timeout specified in the **remotesystem** function call is used. If that is **0** (meaning no timeout), and the policy server encounters the policy timeout, the remote host may have a "hung" **pblocald** process.

Syntax

```
policytimeout ( <timeout_value_in_seconds> );
```

Arguments

timeout_value_in_seconds Required. Specifies the policy timeout value in seconds.

Return Values

Not applicable

Example 1

```
policytimeout (25);  
submitconfirmuser (user);  
accept;
```

Example 2

```
tmout=2;  
policytimeout (tmout);  
submitconfirmuser (user);  
accept;
```

Example 3

```
policytimeout (25);  
...  
policytimeout (40);  
...  
policytimeout (0);  
...
```

See Also

```
remotesystem()
```

quote

Description

The **quote()** function encloses a string in the specified character. It also inserts a backslash character (\) in front of any special characters that are contained in the string, to indicate that these characters should be taken literally (that is, treated as special characters). The **quote()** function is useful when parsing arguments into commands that are shell scripts.



For more information on special characters, please see "[Special Characters](#)" on page 82.

Syntax

```
result = quote (string1, quotechar);
```

Arguments

string1	Required. The string to enclose in the specified quotechar
quotechar	Required. The character to use as the enclosing character

Return Values

result contains the quoted string.

Example

In the example:

```
result = quote ("Hello, Hello, Hello", "*");
```

result is assigned:

```
"*Hello, Hello, Hello*"
```

remotesystem

Description

Introduced in Privilege Management for Unix & Linux 7.1, **remotesystem()** is used to run commands on a host other than the policy server host (any Privilege Management for Unix & Linux runhost) as part of the policy. This can be called as a procedure (command output is shown on **pbrun**'s terminal) or as a function (command output is captured into a policy variable). This is similar to the

system() function/procedure, however the command is run on a different host. The Privilege Management for Unix & Linux variable status is set to the return code of the command upon exit. Input to the command comes from the user's keyboard or from the **inputstring** argument if it is present. Output goes to the user's screen or to the result string variable, if present.

If the specified host is the same as the **submithost**, the requesting program (**pbrun**, **pbksh**, **pbsh**) will execute the command. If the specified host is not the **submithost**, **pblocald** will be used to execute the command.

This is primarily intended to be used as a function, without interactive keyboard or screen I/O. Limited I/O is allowed, however programs such as vi are not supported.

This policy function requires Privilege Management for Unix & Linux 7.1 clients (**pbrun**, **pbsh**, **pbksh**, **pbssh**, **pblocald**).



Note: Do not use **remotesystem()** as a procedure (without the **result** variable) in a policy that is processing **pbguid** requests.

Syntax

```
[result =] remotesystem( hostname, user, environment, timeout, cwd, "command and arguments"
[,inputstring]);
```

Arguments

hostname	Required. The host on which to run the command. This can be short name, FQDN, or IP address.
user	Required. The user to execute the command as.
environment	Required. A list specifying the environment variables to execute the command with.
timeout	Required. The maximum time in seconds that the remote command is allowed to take. A timeout of zero indicates no timeout.
cwd	Required. Directory from which to execute the command.
command	Required. The command (possibly including path) and arguments to run.
inputstring	Optional. Command input, formatted into a single character string

Return Values

If the result variable is specified, **remotesystem()** acts as a function returning the output of the command. If the result variable is not specified, the output from the command that is executed by the **remotesystem()** procedure will appear on **stderr** of the requesting program (**pbrun**, **pbsh**, **pbksh**, **pbssh**).

The Privilege Management for Unix & Linux variable status is set to the return code. In general, a return code of **0** means the command completed successfully. For a description of non-zero return codes, see the documentation for the command that is being run. A status of **-15** indicates a timeout.

Examples

In the example:

```
processlist = remotesystem( submithost, "root", {"PATH=/bin","TMPDIR=/tmp/"}, 20, "/tmp", "ps -ef", "" );
```

The **processlist** variable is assigned the output from the **ps** command executed on the **submithost**. Note that the optional input argument was empty quotes, meaning that the command will not be given any input. In the example:

```
processlist = remotesystem( submithost, "root", {"PATH=/bin","TMPDIR=/tmp/"}, 20, "/tmp", "bash -c 'ps -ef | grep ^" +user+"'");
```

Again, the **processlist** variable is assigned the output from the **ps** command on executed on the **submithost**. Note that the optional input argument was not provided, meaning that the submituser's keyboard is connected through to the command. Note that **bash -c** was used to allow for a shell to process the multiple commands (**ps** and **grep**).

See Also

```
system(), status
```

runtimewarn

Description

After the specified number of minutes, a message is written to the user's **stderr**. If the optional message argument is not specified, the default message is: *WARNING: You have exceeded the maximum allowed session time.*

Internally, this feature makes use of the new read-only policy variables **runtimewarn** and **runtimewarnmsg** to communicate the details from the policy server to the run host.

This feature might typically be used to warn a user of an upcoming timeout specified by the **runtime-limit** variable. Note that the **runtimewarn** time limit is specified in minutes (within a procedure), while **runtimeout** is specified in seconds (as a variable).

This feature may also be used with the new **runtimewarnlog()** procedure described below.

Syntax

```
runtimewarn( minutes [, message] );
```

Arguments

Minutes	Required positive integer specifying the timeout in minutes.
Message	Optional string specifying a message to issue to the user on stderr .

Examples

```
runtimewarn(20);
runtimewarn(20, "Warning, your session will expire soon!");
```

See also

```
runtimelimit, runtimewarnlog()
```

runtimewarnlog

Description

This feature requires an I/O log. After the specified number of minutes, a message is written to the logserver's **syslog**. This message allows variable substitution using the `%variable%` syntax. Any variable recorded in the Accept event can be incorporated into the message. When the finish event is logged, the new **timelimitexceeded** variable is set to **1**. If the time limit is not exceeded, the **timelimitexceeded** variable is not recorded in the finish event. If the optional message argument is not specified, the default message is: `user:%user% exceeded time limit as %runuser%@%runhost% for %runargv%`

Internally, this feature makes use of the new read-only policy variables **runtimewarnlog** and **runtimewarnlogmsg** to communicate the details from the policy server to the run host.

This feature might typically be used to create log entries of the longer sessions, possibly after warning a user using **runtimewarn()** of an upcoming timeout specified by the **runtimelimit** variable. Note that the **runtimewarnlog** time limit is specified in minutes (within a procedure), while **runtimeout** is specified in seconds (as a variable).

Syntax

```
runtimewarnlog( minutes [, message] );
```

Arguments

Minutes	Required positive integer specifying the timeout in minutes.
Message	Optional string specifying a message to syslog on the logserver.

Examples

```
runtimewarnlog(20);  
runtimewarnlog(20, "user:%user% exceeded session time limit");
```

See also

```
runtimelimit, runtimewarn()
```

system

Description

The **system()** function is used to run commands on the policy server host as part of the policy. The Privilege Management for Unix & Linux variable status is set to the return code of the command upon exit. By default, commands that are run by the **system()** function

are run as root. However, commands can be run as different users by setting the Privilege Management for Unix & Linux variable **subprocuser**.

Input to the command comes from the user's keyboard or from the **inputstring** if it is present. Output goes to the user's screen or to the result string variable, if present.



Note: Do not use **system()** without the **result** variable in a policy that is processing **pbguid** requests.

Syntax

```
[result =] system (command [,inputstring]);
```

Arguments

command	Required. The command to run
inputstring	Optional. Command input arguments, formatted into a single character string

Return Values

result contains the output of the command. If the result variable is not specified, the output from the command that is executed by the **system()** function will appear on **stderr** of the requesting program (**pbrun**, **pbsh**, **pbksh**).

The Privilege Management for Unix & Linux variable status is set to the return code. In general, a return code of **0** means the command completed successfully. For a description of non-zero return codes, see the documentation for the command that is being run.

Example

In the example

```
result = system ("echo date");
```

result is assigned **date\n** because the echo command outputs the string **date** with a newline character.

See Also

```
policygetenv(), policysetenv, policyunsetenv, status, subprocuser
```

unset

Description

The **unset** procedure is used to remove temporary variables from the event and I/O log files when the variables are no longer needed. Variables that are required for the functioning of a Privilege Management for Unix & Linux daemon may not be unset.

Syntax

```
unset (variable);
```

Arguments

variable Required. The temporary variable to remove

Return Values

Not applicable

Example

In the example,

```
unset ("xyz");
```

removes the temporary variable **xyz** from the log files.

See Also

```
isset(), logomit
```

NIS Functions

NIS functions are used to access the network information system. They are summarized in the following table.

Table 32. NIS Function Summary

Function	Description
<code>innetworkgroup()</code>	Determines if a machine is a member of a specific netgroup
<code>inusernetgroup()</code>	Determines if a user is a member of a specific netgroup

innetworkgroup

Description

The `innetworkgroup()` function determines if a specific machine is a member of a netgroup.

Syntax

```
result = innetworkgroup (netgroup, host [, user [, domain]])
```

Arguments

netgroup	Required. Name of the netgroup to query
host	Required. The name of the machine in question
user	Optional. The user name
domain	Optional. The user name

Return Values

true	The specified machine is a member of the specified netgroup
false	The specified machine is not a member of the specified netgroup

Example

In this example,

```
result = innetworkgroup ("myhosts", "machine1");
```

result contains an integer value of **1 (true)** if **machine1** is a member of the netgroup **myhosts**. **result** contains an integer value of **0 (false)** if **machine1** is not a member of the netgroup **myhosts**.

See Also

```
inusernetgroup()
```

inusernetgroup

Description

The **inusernetgroup()** function determines if a user is a member of a specific netgroup.

Syntax

```
result = inusernetgroup (netgroupname, username);
```

Arguments

netgroupname	Required. Name of the netgroup to query
username	Required. Name of the user in question

Return Values

true	The specified user is a member of the specified netgroup
false	The specified user is not a member of the specified netgroup

Example

In this example,

```
currentuser = "sysadm1";  
result = inusernetgroup ("myhosts", currentuser);
```

result contains an integer value of **1 (true)** if **sysadm1** is a member of the netgroup **myhosts** or **0 (false)** if **sysadm1** is not a member of the netgroup.

See Also

```
innetwork()
```

Policy Environment Functions and Procedures

Policy environment functions and procedures are used to get, set, and unset the values of environment variables on the policy server host during the run of a policy. The following table summarizes these functions and procedures.

Table 33. Policy Environment Functions and Procedures

Function/ Procedure	Description
<code>getlistsetting()</code>	Returns the value of a list setting in the current policy server host settings file. Version 4.0 and earlier: function not available Version 5.0 and later: function available
<code>getnumericsetting()</code>	Returns the value of a numeric setting in the current policy server host settings file. Version 4.0 and earlier: function not available Version 5.0 and later: function available
<code>getstringsetting()</code>	Returns the value of a string setting in the current policy server host settings file. Version 4.0 and earlier: function not available Version 5.0 and later: function available
<code>getyesnosetting()</code>	Returns the value of a yes/no setting in the current policy server host settings file. Version 4.0 and earlier: function not available Version 5.0 and later: function available
<code>policygetenv()</code>	Sets the value of a local variable to that of an environment variable on the policy server host
<code>policysetenv</code>	Enables the user to locally set an environment variable on the policy server host
<code>policyunsetenv</code>	Used to locally unset the value of an environment variable on the policy server host

getlistsetting

- **Version 4.0 and earlier:** `getlistsetting()` function not available
- **Version 5.0 and later:** `getlistsetting()` function available

Description

The `getlistsetting()` function returns the value of a list setting in the current policy server host settings file.

Syntax

```
getlistsetting (setting-name)
```

Arguments

setting-name	Required. The list setting to retrieve.
---------------------	---

Return Values

A list that contains the value of the specified setting.

Example

```
submitMasterList = getlistsetting("submitmasters");
```

See Also

```
getnumericsetting, getstringsetting, getyesnosetting
```

getnumericsetting

- **Version 4.0 and earlier:** `getnumericsetting()` function not available
- **Version 5.0 and later:** `getnumericsetting()` function available

Description

The `getnumericsetting()` function returns the value of a numeric setting in the current policy server host settings file.

Syntax

```
getnumericsetting (setting-name)
```

Arguments

setting-name	Required. The numeric setting to retrieve.
---------------------	--

Return Values

A number that contains the value of the specified setting.

Example

```
delayTime= getnumericsetting("masterdelay");
```

See Also

```
getlistsetting (), getstringsetting (), getyesnosetting ()
```

getstringsetting

- **Version 4.0 and earlier:** `getstringsetting()` function not available
- **Version 5.0 and later:** `getstringsetting()` function available

Description

The **getstringsetting()** function returns the value of a string setting in the current policy server host settings file.

Syntax

```
getstringsetting (setting-name)
```

Arguments

setting-name	Required. The string setting to retrieve.
---------------------	---

Return Values

A string that contains the value of the specified setting.

Example

```
policyDirectory = getstringsetting("policydir");
```

See Also

```
getlistsetting (), getnumericsetting (), getyesnosetting ()
```

getyesnosetting

- **Version 4.0 and earlier:** **getyesnosetting()** function not available
- **Version 5.0 and later:** **getyesnosetting()** function available

Description

The **getyesnosetting()** function returns the value of a yes/no setting in the current policy server host settings file.

Syntax

```
getyesnosetting (setting-name)
```

Arguments

setting-name	Required. The yes/no setting to retrieve.
---------------------	---

Return Values

A number containing the value of the specified setting.

- **0** False. A **no** value
- **1** True. A **yes** value

Example

```
useKerberos = getyesnosetting("kerberos");
```

See Also

```
getnumericsetting (), getstringsetting ()
```

policygetenv

Description

The **policygetenv()** function sets the value of a local variable to that of an environment variable on the policy server.

Syntax

```
result = policygetenv (variable);
```

Arguments

variable	
	Required. The environment variable on the policy server host that is used to set the value of the local variable.

Return Values

The value of the specified environment variable

Example

In this example,

```
termtype = policygetenv("TERM");
```

the local variable **termtype** is set equal to the **TERM** variable on the policy server.

See Also

```
policysetenv, policyunsetenv, system()
```

policysetenv

Description

The **policysetenv** procedure is used to locally set an environment variable on the policy server host.

Syntax

```
policysetenv(variable, value)
```

Arguments

variable	Required. The environment variable on the policy server host to set
value	Required. The value to set the variable to

Return Values

Not applicable

Example

In this example,

```
policysetenv("PATH", "/bin:/usr/bin:/usr/sbin");
```

the policy server host's PATH variable is set to **/bin:/usr/bin:/usr/sbin**.

See Also

```
policyunsetenv, system()
```

policyunsetenv

Description

The **policyunsetenv** procedure is used to locally unset an environment variable on the policy server

Syntax

```
policyunsetenv(variable)
```

Arguments

variable	Required. The environment variable to be unset on the policy server
-----------------	---

Return Values

The value of the environment variable

Example

In this example,

```
policyunsetenv("OLDPATH");
```

the environment variable **OLDPATH** is removed from the policy server's environment.

See Also

```
policysetenv
```

String Functions

String functions are used to manipulate and handle string variables. The following table summarizes the available string functions.

Table 34. String Function Summary

Function	Description
charlen()	Returns the number of single-byte or multiple-byte characters in a string Version 6.0.1 and earlier: function not available Version 6.1 and later: function available
gsub()	Replaces all occurrences of a pattern within a source string
length()	Returns the number of bytes in a string
pad()	Pads a string with a specified pad character
sub()	Replaces the first occurrence of a pattern within a source string
substr()	Extracts part of a string
tolower()	Returns a copy of a string, converted to all lowercase Version 4.0 and earlier: function not available Version 5.0 and later: function available
toupper()	Returns a copy of a string, converted to all uppercase Version 4.0 and earlier: function not available Version 5.0 and later: function available

charlen

Description

The **charlen()** function returns the number of characters (single-byte or multiple-byte) in the argument string.

By contrast, the **length()** function returns the number of bytes in a string, which equals the number of characters only for single-byte character encodings. Also in contrast to the **length()** function, the **charlen()** function does not accept a list as an argument.

Syntax

```
result = charlen (string)
```

Arguments

string	Required. A character string in single-byte or multiple-byte encoding.
---------------	--

Return Values

result Contains an integer that indicates the number of characters in **string**

Example

In this example,

```
string = "BeyondTrust Software";  
howLong = charlen(string);
```

The **howLong** variable contains the integer value **20**.

See Also

```
length()
```

gsub

Description

The **gsub()** function replaces all occurrences of the pattern within the source string.

Syntax

```
result = gsub (pattern, replacement, sourcestring);
```

Arguments

pattern	Required. The regular expression pattern to search for.
replacement	Required. The replacement string.
sourcestring	Required. The source string to search for all occurrences of pattern.

Return Values

The resulting string

Example

In this example,

```
newstring = gsub("abc", "xyz", startingstring)
```

xyz replaces all occurrences of **abc** in **startingstring**.

See Also

```
sub ()
```

length

Description

The **length()** function returns the length, in bytes, of the specified string. Note that for multiple-byte character sets, the number of bytes is not the same as the number of characters; use the **charlen()** function instead.

Syntax

```
result = length (string1);
```

Arguments

string1	Required. The string for which a length value is determined.
----------------	--

Return Values

result is set to the length (as an integer value) of **string1**.

Example

In this example,

```
currentuser = "John Stone";  
result = length (currentuser);
```

result would be an integer with a value of **10**.

pad

Description

The **pad()** function creates a new string from **string1** based on the specified length (**length**) and pad character (**padchar**). If **string1** is shorter than the specified length, then it is padded by adding the appropriate number of the specified pad character to the end of the string. If **string1** is longer than the specified length, then it is truncated and pad characters are not added. If the length of **string1** is equal to the specified length, no changes are made and the original contents of **string1** are returned in **result**.

The **pad()** function supports both single-byte and multiple-byte character sets.

Syntax

```
result = pad (string1, length, padchar);
```

Arguments

string1	Required. The string field to pad using the specified pad character
length	Required. The length (number of characters) of the new string
padchar	Required. The pad character that is used to pad string1 , if string1 is shorter than the value specified in length

Return Values

result contains the new string.

Examples

In this example,

```
string = "Jim White";  
result = pad (string1, 10, "123");
```

result contains **Jim White1**.

In this example,

```
string1 = "書策搜";  
result = pad (string1, 4, "文");
```

result contains the value 書策搜文.

sub

Description

The **sub()** function replaces the first occurrence of the pattern within the source string.

Syntax

```
result = sub (pattern, replacement, sourcestring);
```

Arguments

pattern	Required. The regular expression pattern to search for
replacement	Required. The replacement string
sourcestring	Required. The source string to search for the first occurrence of pattern

Return Values

The resulting string

Example

In this example,

```
newstring = sub("\n$", "", textstring)
```

the first occurrence of a trailing new line is replaced with nothing, effectively chopping it off.

See Also

```
gsub ()
```

substr

Description

The **substr()** function extracts a substring from the specified string variable (**string1**) based on the provided starting position (**start**) and optional length (**length**). The first character in **string1** is position **1**. If the optional length is not specified, then **substr()** returns all characters from the starting position through the end of the string.

An error is generated if a negative starting position is given or if the starting position is past the end of the string (for example, if **string1** is 10 characters long and the specified starting location is **12**).

The **substr()** function supports single-byte and multiple-byte character strings. In either case, the starting position and length are in units of characters, not bytes.

Syntax

```
result = substr (string1, start [, length]);
```

Arguments

string1	Required. The string from which a substring is extracted
start	Required. Specifies the substring starting position within string1 . The first character in string1 is position 1 .
length	Optional. Specifies the maximum length of the substring

Return Values

result contains the new substring.

Examples

In this example,

```
UserList = "User1, User2, User3";  
result1 = substr (UserList, 8, 5);  
result2 = substr (UserList, 8);
```

result1 contains the value **User2**, and **result2** contains **User2, User3**.

In this example,

```
UserList = "書策搜書策搜書策搜書策搜書策搜書策搜書策搜";  
result = substr (UserList, 8, 5);
```

result contains the value 策搜書策搜.

tolower

- **Version 4.0 and earlier:** **tolower()** function not available
- **Version 5.0 and later:** **tolower()** function available

Description

The **tolower()** function returns a copy of a string, converted to all lowercase.

The **tolower()** function supports both single-byte and multiple-byte character sets. If the character set for the locale does not distinguish uppercase and lowercase characters, the original string is returned unchanged.

Syntax

```
tolower (string)
```

Arguments

string	Required. The string to convert to lowercase.
---------------	---

Return Values

A string that contains a lowercase copy of the argument.

Example

```
result = tolower (variableName);  
result = tolower("String Constant");
```

See Also

```
toupper ()
```

toupper

- **Version 4.0 and earlier:** **toupper()** function not available
- **Version 5.0 and later:** **toupper()** function available

Description

The **toupper()** function returns a copy of a string, converted to all uppercase.

The **toupper()** function supports both single-byte and multiple-byte character sets. If the character set for the locale does not distinguish uppercase and lowercase characters, the original string is returned unchanged.

Syntax

```
toupper (string)
```

Arguments

string	Required. The string to convert to uppercase.
---------------	---

Return Values

A string that contains an uppercase copy of the argument.

Example

```
result = toupper (variableName);  
result = toupper ("String Constant");
```

See Also

```
tolower()
```

Task Control Procedures

The task control procedures are used to control the execution of the secured task. These functions are summarized in the following table.

Table 35. Task Control Procedures

Procedure	Description
setkeystrokeaction	Used in a policy to override forbidkeypatterns and forbidkeyaction , which will be discontinued at a future date

setkeystrokeaction

Description

The **setkeystrokeaction** procedure looks for a keystroke pattern in the input stream and performs the specified action. It extends the functionality of the **forbidkeypatterns** list and **forbiddenkeyaction** string. If used in a policy, **setkeystrokeaction** overrides **forbidkeypatterns** and **forbidkeyaction**, which will be discontinued at a future date.

Syntax

```
setkeystrokeaction(pattern, patterntype, action);
```

Arguments

pattern	Required. The pattern to match. This can be a shell-type template or regular expression.
patterntype	Required. The type of search, specified by the pattern argument. Valid values are shell for shell-style pattern matching or re for regular expression matching.
action	Required. The action to take if the pattern is found. If set to reject , the program aborts and the action is logged in the Privilege Management for Unix & Linux event log and syslog (if in use). A value of ignore results in no action being taken when the pattern is encountered. Any other value is used to tag the keystroke event in the event log.

Return Values

None

Examples

The first example,

```
setkeystrokeaction("*rm*", "shell", "reject");
```

has **setkeystrokeaction** set to terminate the current job if the pattern **rm** is found anywhere in the input stream. This would react to **rm**, **/bin/rm**, **disarm**, and **alarm**.

In the second example,

```
setkeystrokeaction("*rm*", "shell", "warn");
```

if **rm** is found anywhere in the input stream, **setkeystrokeaction** is configured to record the keystroke event with a **warn** tag in the event log.

In the third example,

```
setkeystrokeaction("rm", "re", "reject");
```

the job is terminated if the pattern **rm** is seen anywhere in the input.

In the last example,

```
setkeystrokeaction("[[:boundary:]]rm[[:boundary:]]", "re", "user ran rm");
```

the **setkeystrokeaction** procedure logs a keystroke event and tags it with **user ran rm** if **rm** is seen as an entire word. It ignores words that contain the letters **rm** (for example, **disarm** or **alarm**) but would react to **rm** and **/bin/rm**.

See Also

```
forbidkeyaction, forbidkeypatterns
```

Task Environment Functions and Procedures

Task environment functions are used to manage task environment variables. The task environment functions and procedures are summarized in the following table.

Table 36. Task Environment Functions and Procedures

Function/ Procedure	Description
keystrokeactionprofile	Provides advanced control over remote SSH and Telnet sessions
getenv()	Retrieves an environment variable from env
keepenv	Keep only the listed variables. Clear all others from runenv
setenv	Sets the value of an environment variable in runenv
unsetenv	Delete an environment variable from runenv

All task environment functions and procedures act upon the Privilege Management for Unix & Linux environment variables **env** and **runenv**.

env and **runenv** are list variables that contain all of the environment variables that are defined for the current request. **env** is a read-only variable that contains task information from the initial task request on the submit host. **runenv** is a modifiable variable that contains the task information that is actually used during task execution on the run host.

env and **runenv** have the following format:

```
{"variable-name=value", "variable-name=value", ...};
```



For more information on **env** and **runenv**, please see "Task Information Variables" on page 1.

keystrokeactionprofile

Description

The Advanced Keystroke Action component was introduced in Privilege Management for Unix & Linux version 9.4.2 and provides advanced control over remote SSH and Telnet sessions.

Syntax

```
keystrokeactionprofile="profile";
```

Arguments

profile

Required

A configured Advanced Keystroke Action profile

Return Values

None

Example

```
keystrokeactionprofile="demo";
```



For more information on Advanced Keystroke Action, please see *Advanced Keystroke Action* in the [Privilege Management for Unix & Linux Administration Guide](https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

getenv

Description

The **getenv()** function returns the value of the environment variable that is specified in the name parameter.

Values that are returned by **getenv** are unaffected by the **setenv**, **keepenv**, and **unsetenv** procedures, because **getenv** accesses the user's original, read-only task environment variable information that is stored in the **env** variable from the client on the submit host.

Syntax

```
result = getenv (name, value);
```

Arguments

name	Required. A string that contains the name of a task environment variable.
value	Optional. A string that contains the value to use if the environment variable name does not exist in env .

Return Values

If the specified task environment variable is found, then result contains its value.

If the specified task environment variable is not found, then the value returns as a string. If value is not specified, then an empty string is returned.

Example

In this example,

```
result = getenv ("TZ");
```

the value of the environment variable **TZ** is retrieved from env and stored in **result**. If **TZ** is not found, then **result** is empty.

See Also

```
setenv
```

keepenv

Description

The **runenv** variable is a list in which each element contains an environment variable. The format of a **runenv** element is **name=value**, where name is the name of an environment variable and value is the current value of that variable.

The **keepenv** procedure modifies the **runenv** variable so that it contains only the variables that are listed as input parameters. All other environment variables that are stored in the **runenv** variable are deleted.

keepenv is typically used to limit the set of environment variables that are available to the current task during execution.

Syntax

```
keepenv (name1, [,name2, ...]);
```

Arguments

name1	Required. String that contains the name of a task environment variable that should be stored in runenv .
name2	Optional. String that contains the name of a task environment variable that should be stored in runenv .

Return Values

Because **keepenv** is a procedure, no return value is set.

Example

In this example,

```
keepenv ("TERM", "CWD", "PS1");
```

runenv contains the environment variables **TERM**, **CWD**, and **PS1**. All other environment variables are deleted from **runenv**.

See Also

```
setenv, unkeepenv()
```

setenv

Description

The **setenv** procedure sets the value of an environment variable in **runenv**.

Syntax

```
setenv (name, value);
```

Arguments

name	Required. String that contains the name of the variable to set in runenv
value	Required. String that contains the value of the specified variable

Return Values

Because **setenv** is a procedure, no return value is set.

Example

In this example,

```
setenv ("SHELL", "/bin/sh");
```

the **SHELL** environment variable that is stored in **runenv** is set to **/bin/sh**.

See Also

```
keepenv, setenv
```

unsetenv

Description

The **unsetenv** procedure deletes environment variables from **runenv**.

Syntax

```
unsetenv (name1 [, name2,...]);
```

Arguments

name1	Required. A string or a list of character strings that contain the names of runenv environment variables to delete
name2	Optional. A string or a list of character strings that contain the names of runenv environment variables to delete

Return Values

Because **unsetenv** is a procedure, no return value is set.

Example

In this example,

```
unsetenv ("IFS", "USER");
```

the **runenv** environment variables **IFS** and **USER** are deleted.

See Also

```
keepenv
```

Command Line Parsing Functions

Privilege Management for Unix & Linux provides functions to facilitate the parsing of command arguments. The following table summarizes these functions.

Table 37. Command Line Parsing Functions

Function	Description
<code>getopt()</code>	Examines a list of arguments for short options. Version 3.5 and earlier: function not available Version 4.0 and later: function available
<code>getopt_long()</code>	Examines a list of arguments for any combination of short or long-style options. Version 3.5 and earlier: function not available Version 4.0 and later: function available
<code>getopt_long_only()</code>	Examines a list of arguments long-style options. Version 3.5 and earlier: function not available Version 4.0 and later: function available

getopt

- **Version 3.5 and earlier:** `getopt()` function not available
- **Version 4.0 and later:** `getopt()` function available

Description

Breaks up command lines for easy parsing and to check for legal options. This function examines a list of arguments for short options.

A short option consists of a dash followed by a single letter and possibly a parameter. For example, in the command `command -a -b name -c`, `-a` and `-c` are short options with no extra parameter, and `-b` is a short option with the parameter name.

On the first invocation, `getopt()` examines the first argument. On subsequent invocations, it picks up where it left off and examines the next argument.

Syntax

```
result = getopt (argc, argv, short-option-string)
```

Arguments

<code>argc</code>	Required. Number. The number of entries that are in the argument array list <code>argv</code> .
<code>argv</code>	Required. List. The argument array to process.
<code>short-option-string</code>	Required. A string that contains valid options. This list contains the letters for the short options. Each letter can be followed by a single colon (:) to indicate a required argument if

the option is found. Each letter can be followed by two colons (::) to indicate an optional argument to the option. The leading characters of the short option string can modify the search characteristics as follows: A leading **+** stops parsing as soon as the first non-option parameter is found that is not an option argument. All other parameters are treated as non-option strings. A leading **-** returns non-option parameters at the place where they are found.

Return Values

If a valid option is found, then the function returns that option. If an optional or required argument is associated with the option, then the policy variable **optarg** contains the value of that argument.

If no valid option is found or if a required argument is missing, then a question mark (?) is returned. The variable **optchar** is set to the letter of the problem option.

When the end of the argument list is found, an empty string, "", is returned.

The variable **optind** is set to the subscript of the next string in the **argv** list.

Example

The example

```
result = getopt(argc, argv, "ab:c");
```

examines the list of arguments in **argv** looking for **-a** or **-c** without a parameter, or **-b** with a parameter.

See Also

```
getopt_long(), getopt_long_only(), optarg, optchar, opterr, optind, optopt, optrese
```

getopt_long

- **Version 3.5 and earlier:** `getopt_long()` function not available
- **Version 4.0 and later:** `getopt_long()` function available

Description

Breaks up command lines for easy parsing and to check for legal options. This function examines a list of arguments for any combination of short-style or long-style options.

A short option consists of a dash followed by a single letter and possibly a parameter. For example, in the command `command -a -b name -c`, **-a** and **-c** are short options with no extra parameter, and **-b** is a short option with the parameter `name`.

A long option consists of two dashes followed by a name and possibly a parameter. For example, in the command `command --option1 --option2=2 --option3 parameter --option4`, **--option1** and **--option4** are long options with no parameters, and **--option2** and **--option3** are options with extra parameters.

On the first invocation, it examines the first argument. On subsequent invocations, it picks up from where it left off and examines the next argument.

Syntax

```
result = getopt_long(argc, argv, short-option-string, long-option-list)
```

Arguments

argc	Required. Number. The number of entries that are in the argument array list argv .
argv	Required. List. The argument array to process.
short-option-string	Required. A string that contains valid options. This list contains the letters for the short options. Each letter can be followed by a single colon (:) to indicate a required argument if the option is found. Each letter can be followed by two colons (::) to indicate an optional argument to the option. The leading characters of the short option string can modify the search characteristics as follows: A leading + stops parsing as soon as the first non-option parameter is found that is not an option argument. All other parameters are treated as non-option strings. A leading - returns non-option parameters at the place where they are found.
long-option-list	Required. List. A list of strings that contains the long options. Each parameter can be followed by a single colon (:) to indicate it has a required parameter, or two colons (::) to indicate that it may have an optional parameter

Return Values

If a valid option is found, then the function returns that option. If an optional or required argument is associated with the option, then the policy variable **optarg** contains the value of that argument.

If no valid option is found, or if a required argument is missing, then a question mark (?) is returned. The variable **optchar** is set to the letter of the problem option.

When the end of the argument list is found, an empty string, "", is returned.

The variable **optind** is set to the subscript of the next string in the **argv** list.

Example

The example

```
result = getopt_long(argc, argv, "ab:c", {"long1", "long2:"});
```

examines the list of arguments in **argv** looking for **-a** or **-c** without a parameter, **-b** with a parameter, **--long1** without a parameter, or **--long2** with a parameter.

See Also

```
getopt(), getopt_long_only(), optarg, optchar, opterr, optind, optopt, optreset, optstrictparameters
```

getopt_long_only

- **Version 3.5 and earlier:** `getopt_long_only()` function not available
- **Version 4.0 and later:** `getopt_long_only()` function available

Description

Breaks up command lines for easy parsing and to check for legal options. This function examines a list of arguments for long-style options only.

A long option usually consists of two dashes followed by a name and possibly a parameter. When using the long-only version of **getopt**, the function also recognizes a single dash at the front of an option. For example, in the command `command --option1 --option2=2 --option3 parameter --option4`, **--option1** and **--option4** are long options with no parameters, and **--option2** and **--option3** are options with extra parameters.

On the first invocation, it examines the first argument. On subsequent invocations, it picks up from where it left off and examines the next argument.

Syntax

```
result = getopt_long_only (argc, argv, short-option-string, long-option-list)
```

Arguments

argc	Required. Number. The number of entries in the argument array list argv .
argv	Required. List. The argument array to process.
short-option-string	Required. Although this function does not process short options, the entry is still available to specify the leading control modifiers. The leading characters of the short option string may modify the search characteristics as follows: A leading + stops parsing as soon as the first non-option parameter is found that is not an option argument. All other parameters are treated as non-option strings. A leading - returns non-option parameters at the place where they are found.
long-option-list	Required. List. A list of strings that contains the long options. Each parameter can be followed by a single colon (:), to indicate it has a required parameter, or two colons (::) to indicate that it may have an optional parameter

Return Values

If a valid option is found, then the function returns that option. If an optional or required argument is associated with the option, then the policy variable **optarg** contains the value of that argument.

If no valid option is found, or if a required argument is missing, then a question mark (?) is returned. The variable **optchar** is set to the letter of the problem option.

When the end of the argument list is found, an empty string, "", is returned.

The variable **optind** is set to the subscript of the next string in the **argv** list.

Example

```
result = getopt_long_only (...)
```

See Also

```
getopt(), getopt_long()
```

User and Password Functions

User and password functions are used to verify passwords and provide password control. The following table summarizes the user and password functions.

Table 38. User and Password Function and Variable Summary

Element	Description
<code>getfullname()</code> function	Returns the specified user's full name
<code>getgroup()</code> function	Returns the specified user's primary group
<code>getgrouppasswd()</code> function	Prompts for a user and the password of one of the members of the group specified as argument to the function
<code>getgroups()</code> function	Returns all groups the specified user is in
<code>gethome()</code> function	Returns the specified user's home directory
<code>getshell()</code> function	Returns the specified user's default login shell.
<code>getstringpasswd()</code> function	Prompts the user for a special password
<code>getuid()</code> function	Returns the user's uid
<code>getuserpasswd()</code> function	Prompts the user for the password belonging to the specified user
<code>ingroup()</code> function	Determines whether a user belongs to a specific group
<code>submitconfirmuser()</code> function	Controls if a user must enter a password before the current task request can be accepted
<code>runconfirmuser</code> variable	Controls whether a user must enter a password before the current task request can be executed.
<code>runconfirmmessage</code> variable	Contains the prompt that is displayed when the submitting user is required to provide a password.

getfullname

Description

The `getfullname()` function retrieves the full name of the specified user. This information is taken from the `gecos` field of `/etc/passwd` on the policy server host or the password map in NIS.

Syntax

```
result = getfullname([user]);
```

Arguments

user	Optional. The name of the user ID for which a full name is retrieved. The value of the <code>runuser</code> variable is used when this argument is not specified.
-------------	---

Return Values

The full name of the user as specified in the **gecos** field of **/etc/passwd** or the NIS password map. An error is returned if the user is null or invalid.

Examples

In the example,

```
result = getfullname();
```

result is assigned the full name of the **runuser**.

In the next example,

```
result = getfullname("user1");
```

result is assigned the full name of **user1**.

getgroup

Description

The **getgroup()** function retrieves the first occurrence of the group name that is associated with the GID to which the specified user belongs. This information is taken from the **gecos** field of **/etc/passwd** on the policy server host or the password map in NIS.

Syntax

```
result = getgroup([user]);
```

Arguments

user

Optional. The name of the user for which the group should be retrieved. If this argument is not specified, the value of the **runuser** variable is used.

Return Values

If the user is found, **result** contains the first occurrence of the group name that is associated with the GID to which the specified user belongs as found in **/etc/passwd** or the NIS password map. An error is returned if the user is null or invalid.

Example

In this example,

```
result = getgroup("SysAdm001");
```

if **SysAdm001** is found, **result** contains the first occurrence of the group name that is associated with the GID to which the specified user belongs.

See Also

```
getgroups ()
```

getgrouppasswd

Description

The **getgrouppasswd()** function prompts first for a user (member of the specified group) then for the password of that user.

Syntax

```
result = getgrouppasswd(group[, prompt[, attempts]]);
```

Arguments

group	Required. The name of the group for which a username and password must be entered.
prompt	Optional. The password prompt that is displayed to the user. If a prompt is not provided, then the following default prompt is displayed: Enter the username and group of someone in the <group name> group.
attempts	Optional. Number of attempts that the user gets to enter the correct password. If the user does not enter the correct password in the specified number of attempts, then the task request is rejected. If the number of attempts is not specified, then the default value of 3 is used.

Return Values

true	Password matched the user password
false	Password did not match the user password

Example

In the example,

```
result = getgrouppasswd("HelpDeskUsers", "Please enter HelpDesk Password:", 1);
```

a user has one attempt to enter a correct username and password for a member of the **HelpDeskUsers** group. If the correct password is not entered in one attempt, then **result** contains **0**. If the correct password is entered in one attempt, then **result** contains **1**.

getgroups

Description

The **getgroups()** function retrieves a list of all groups to which the specified user belongs. This information is taken from the **/etc/groups** file on the policy server host or the group map in NIS.

Syntax

```
result = getgroups([user]);
```

Arguments

user

Optional. The name of the user for which the secondary group names should be retrieved. If this argument is not specified, then the value of the **runuser** variable is used.

Return Values

A list of character strings that contains all of the groups that the user belongs to. An error is returned if the user is invalid or null.

Example

```
result = getgroups(runuser);
```

See Also

```
getgroup()
```

gethome

Description

The **gethome()** function retrieves the home directory for the specified user. This information is obtained from the home directory field of **/etc/passwd** or the NIS password map.

Syntax

```
result = gethome([user]);
```

Arguments

user

Optional. The name of the user for which home directory information should be retrieved. If this argument is not specified, then the value of the **runuser** variable is used.

Return Values

A string that contains the specified user's home directory from the home directory field of **/etc/passwd** or the NIS map. If the user is not found, then **result** contains a blank string.

Example

In this example,

```
result = gethome("JSmith");
```

the home directory for the user **JSmith** is returned in **result**. For example, **/home/JSmith**.

getshell

Description

The **getshell()** function retrieves the default login shell of the specified user. This information is obtained from the shell field of **/etc/passwd** or the NIS password map.

Syntax

```
result = getshell([user]);
```

Arguments

user

Optional. The name of the user for which shell information should be retrieved. If the user is not specified, then the value of the **runuser** variable is used.

Return Values

A string that contains the default login shell for the specified user from the shell field of **/etc/password** or the password NIS map. If the username is not found or is invalid, then the policy will be rejected with an error code.

Example

In this example,

```
result = getshell("JSmith");
```

default shell information for the account **JSmith** is returned in **result**. For example, **/bin/sh**.

getstringpasswd

Description

The **getstringpasswd()** function prompts the user for a password and compares the answer against the previously encrypted password.



Note: The user's failure to provide the correct password does not automatically result in a rejection of the secured task request. The policy should examine the result of the **getstringpasswd()** function and respond accordingly.

Syntax

```
result = getstringpasswd(encryptedpassword[, prompt [, attempts]]);
```

Arguments

encryptedpassword	Required. An encrypted password, which can be generated by pbpasswd . The clear text form of this password is the password that the user is expected to enter.
prompt	Optional. A user prompt that describes the desired password. If none is specified, then the default prompt Password: is used.
attempts	Optional. Number of attempts the user gets to specify the correct password. The default value for attempts is 3 .

Return Values

true	The answer matched the password
false	The answer did not match the password

Example

In this example,

```
result = getstringpasswd(<encrypted string>, "Please enter the Backup Task Password: ", 2);
```

result contains **true** if the user enters the correct Backup Task password. If the correct password is not entered in two attempts, the function sets **result** to **false**.

getuid

Description

The **getuid()** function returns the user ID number for the specified user. This information is taken from the **gecos** field of **/etc/passwd** on the policy server host or the password map in NIS.

Syntax

```
result = getuid([user]);
```

Arguments

user	Optional. The name of the user for which a user ID number should be returned. If this argument is not specified, then the value of the runuser variable is used.
-------------	---

Return Values

result contains the uid of the specified user of **/etc/passwd** or the NIS password map. An error is returned if the user is null or invalid.

Example

```
result = getuid("root");
```

See Also

```
getfullname(), getgroup(), gethome(), getshell()
```

getuserpasswd

Description

The **getuserpasswd()** function prompts the user for the password that belongs to the specified user on the policy server. The password is not echoed to the screen as it is typed.



Note: The user's failure to provide the correct password does not automatically result in a rejection of the secured task request. The policy should examine the result of the **getuserpasswd()** function and respond accordingly.

Syntax

```
result = getuserpasswd(user[, prompt[, attempts[, name, time]]]);
```

Arguments

user	Required. The user whose password must be entered.
prompt	Optional. The prompt to display to the user
attempts	Optional. The number of attempts that the user has to enter the correct password. The default value for attempts is 3 .
name	Optional. The name of a file or persistent variable whose age/expiration determines the re-authentication grace period. If the value starts with a dollar sign (\$), it is treated as a persistent variable, otherwise it is treated as a filename. If name is specified, the time parameter (below) is required.
time	Required if name argument (above) is specified). The time/expiry date (number of seconds) after which a prompt is forced. getuserpasswd() returns true without prompting the user for a password if one of the following is true: <ol style="list-style-type: none"> 1. The file defined by the name argument exists, and has not been modified in the last time seconds. 2. The persistent variable defined by the name argument exists and its expiry date, defined by time, has not been exceeded.

Return Values

true	Password matched
-------------	------------------

false	Password did not match
--------------	------------------------

Example

In this example,

```
result = getuserpasswd(runuser, "Please enter " + runuser _ "'s Password:");
```

result contains **true** if the user enters the password for the **runuser**. If the correct password is not entered in three attempts, then the function sets **result** to **false**.

In this example,

```
getuserpasswd(user, "Passwd for "+user+": ", 3, "/opt/pbul/gp001", 300);
```

the file **/opt/pbul/gp001** will be created at initial successful user authentication and for 5 minutes (300 seconds) thereafter, the user will not be prompted for a password as long as the file is not modified.

See Also

```
submitconfirmuser(), runconfirmuser, getstringpasswd()
```

ingroup

Description

The **ingroup()** function determines whether the specified user is a member of the specified group.

Syntax

```
result = ingroup(user, group);
```

Arguments

users	Required. A username
group	Required. A group name

Return Values

true	user is a member of group
false	user is not a member of group or the user or group is null or invalid

Example

In this example,

```
result = ingroup("user1", "admgroup");
```

result contains an integer value **1** if **user1** belongs to the group **admgroup**. **result** contains an integer value **0** if **user1** does not belong to group **admgroup**.

See Also

```
getgroup(), getgroups()
```

submitconfirmuser

Description

The **submitconfirmuser()** function controls whether or not a user must enter a password before the current task request is accepted. When this function is set, the user submitting the request is prompted for the password that is associated with the submit host username set in this function.



Note: The user's failure to provide the correct password does not automatically result in a rejection of the secured task request. The policy should examine the result of the **submitconfirmuser()** function and respond accordingly.

Syntax

```
result = submitconfirmuser(user[, prompt[, attempts[, name, time]]]);
```

Arguments

user	Required. A string that contains a username that exists on the submit host
prompt	Optional. The prompt text for the password. The default is Enter password for <user>
attempts	Optional. The number of attempts that the user has to enter the correct password. The default value for attempts is 3 .
name	Optional. The name of a persistent variable whose expiration determines the re-authenticate grace period. The value must start with a dollar sign (\$), otherwise no grace period will be set and submitconfirmuser() will automatically prompt for a password. If name is specified, the time parameter (below) is required.
time	Required if name argument (above) is specified). The expiry date (number of seconds) after which a prompt is forced. submitconfirmuser() returns true without prompting the user for a password if the persistent variable, defined by the name argument, exists and its expiry date, defined by time , has not been exceeded.

Return Values

true	Password matched
false	Password did not match

Examples

In this example,

```
result = submitconfirmuser(user, "Please enter the user's password:", 3);
if (result != 1) {
reject;
}
```

the prompt **Please enter the user's password:** is displayed and the user is allowed three login attempts.

In this example,

```
submitconfirmuser(user, "Passwd for "+user+": ", 3, "$gpvar5", 300);
```

a persistent variable **gpvar5** will be created at initial successful user authentication and for 5 minutes (300 seconds) thereafter, the user will not be prompted for a password.

See Also

```
getgrouppasswd(), getstringpasswd(), getuserpasswd(), runconfirmuser, runconfirmmessage
```

PAM Policy Functions

getuserpasswdpam

- **Version 8.0 and earlier:** `getuserpasswdpam()` function not available
- **Version 8.5 and later:** `getuserpasswdpam()` function available

Description

The `getuserpasswdpam()` function uses PAM password authentication on the policy server host for the specified user.

It is similar to using the `getuserpasswd()` function with the `pampasswordservice` keyword in the policy server host's `/etc/pb.settings`.

When used, this policy function will override the `pampasswordservice` setting in the policy server host's settings file and will work even if the PAM setting is set to `no`.

The `getuserpasswdpam()` function prompts the user for the password that belongs to the specified user on the policy server. The password is not echoed to the screen as it is typed.



Note: The user's failure to provide the correct password does not automatically result in a rejection of the secured task request. The policy should examine the result of the `getuserpasswdpam()` function and respond accordingly.

Syntax

```
result = getuserpasswdpam(user, pampasswordservice[, prompt[, attempts[, name, time]]]);
```

Arguments

user	Required. The user whose password must be entered.
pampasswordservice	Required. The name of the PAM service that you want to use for PAM password authentication and account management.
prompt	Optional. Extra text that will appear before the PAM prompt that will display for the user. Enter a null argument ("") if you do not want to add text before the PAM prompt.
attempts	Optional. The number of attempts that the user has to enter the correct password. The default value for attempts is 3 .
name	Optional. The name of a file or persistent variable whose age/expiration determines the re-authentication grace period. If the value starts with a dollar sign (\$), it is treated as a persistent variable, otherwise it is treated as a file name. If name is specified, the time parameter (below) is required.
time	Required if name argument (above) is specified). The time/expiry date (number of seconds) after which a prompt is forced. <code>getuserpasswdpam()</code> returns true without prompting the user for a password if one of the following is true: <ol style="list-style-type: none"> 1. The file defined by the name argument exists, and has not been modified in the last time

- seconds.
- The persistent variable defined by the **name** argument exists and its expiry date, defined by **time**, has not been exceeded.

Return Values

true	Password matched
false	Password did not match or invalid password service.

Example

In this example,

```
result = getuserpasswdpam(runuser, "pbulpass", "Please enter " + runuser + "'s Password: ");
```

result contains **true** if the user enters the password for the **runuser**. If the correct password is not entered in three attempts, then the function sets **result** to **false**.

In this example,

```
getuserpasswdpam(user, "pbulpass", "Passwd for "+user+": ", 3, "/opt/pbul/gp001", 300);
```

the file **/opt/pbul/gp001** will be created at initial successful user authentication and for 5 minutes (300 seconds) thereafter, the user will not be prompted for a password as long as the file is not modified.

See Also

```
getuserpasswdpam(), submitconfirmuser(), runconfirmuser, getstringpasswd()
```



For more information, please see "[Persistent Variable Functions and Procedures](#)" on page 313.



For information about **pampasswordservice**, please see the [Privilege Management for Unix & Linux System Administration Guide](#) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

submitconfirmuserpam

- Version 8.0 and earlier:** **submitconfirmuserpam()** function not available
- Version 8.5 and later:** **submitconfirmuserpam()** function available

Description

The **submitconfirmuserpam()** function controls whether or not a user must enter a password before the current task request is accepted. Password authentication and account management is performed by PAM and name of the PAM service must be provided. When this function is set, the user submitting the request is prompted for the password that is associated with the submit host user name set in this function.

When used, this policy function will override the **pampasswordservice** setting in the submit host's settings file and will work even if the PAM setting is set to **no**.



Note: The user's failure to provide the correct password does not automatically result in a rejection of the secured task request. The policy should examine the result of the **submitconfirmuserpam()** function and respond accordingly.

Syntax

```
result = submitconfirmuserpam(user, pampasswordservice[, prompt[, attempts[, name, time]]]);
```

Arguments

user	Required. A string that contains a user name that exists on the submit host
pampasswordservice	Required. The name of the PAM service that you want to use for PAM password authentication and account management.
prompt	Optional. The prompt text for the password. The default is Enter password for <user> .
attempts	Optional. The number of attempts that the user has to enter the correct password. The default value for attempts is 3 .
name	Optional. The name of a persistent variable whose expiration determines the re-authenticate grace period. The value must start with a dollar sign (\$), otherwise no grace period will be set and submitconfirmuserpam() will automatically prompt for a password. If name is specified, the time parameter (below) is required.
time	Required if name argument (above) is specified). The expiry date (number of seconds) after which a prompt is forced. submitconfirmuserpam() returns true without prompting the user for a password if the persistent variable, defined by the name argument, exists and its expiry date, defined by time , has not been exceeded.

Return Values

true	Password matched
false	Password did not match or invalid password service.

Example

```
result = submitconfirmuserpam(user, "pbulpass", "Please enter the user's password:", 3);
if (result != 1) {reject;}
```

In this example,

```
submitconfirmuserpam(user, "pbulpass", "Passwd for "+user+": ", 3, "$gpvar5", 300);
```

a persistent variable **gpvar5** will be created at initial successful user authentication and for 5 minutes (300 seconds) thereafter, the user will not be prompted for a password.

See Also

```
submitconfirmuser()
```



For more information, please see "[Persistent Variable Functions and Procedures](#)" on page 313.



For information about **pampasswordservice**, please see the [Privilege Management for Unix & Linux System Administration Guide](#) at <https://www.beyondtrust.com/docs/privilege-management/unix-linux/index.htm>.

Persistent Variable Functions and Procedures

Persistent variables are a method of setting variables that persist for a specified time and are synchronized across all of the policy servers in the enterprise. Procedures are provided to list, get, set and delete persistent variables.

Function/Procedure	Description
listpersistentvars()	Returns a list of the current persistent variables. Version 9.4.4 and earlier: function not available Version 9.4.5 and later: function available
setpersistentvar()	Sets a persistent variable in the database. Version 9.4.4 and earlier: function not available Version 9.4.5 and later: function available
getpersistentvarint()	Returns an integer value persistent variable. Version 9.4.4 and earlier: function not available Version 9.4.5 and later: function available
getpersistentvarstring()	Returns a string value persistent variable. Version 9.4.4 and earlier: function not available Version 9.4.5 and later: function available
getpersistentvarlist()	Returns a List value persistent variable. Version 9.4.4 and earlier: function not available Version 9.4.5 and later: function available
delpersistentvar()	Delete a persistent variable from the database. Version 9.4.4 and earlier: function not available Version 9.4.5 and later: function available

listpersistentvars

- **Version 9.4.4 and earlier:** **listpersistentvars()** function not available
- **Version 9.4.5 and later:** **listpersistentvars()** function available

Description

The **listpersistentvars()** procedure returns a list of currently active persistent variables. Variables that expire are not retrieved.

Syntax

```
Var = listpersistentvars(wildcard)
```

Arguments

wildcard	Optional. A glob(3) wildcard limiting the returned values to those matched.
-----------------	--

Return Values

A list that contains the current active persistent variables.

Example

```
vars = listpersistentvars("a*");
```

See Also

```
setpersistentvar, getpersistentvarint, getpersistentvarstring, getpersistentvarlist,  
delpersistentvar
```

setpersistentvar

- **Version 9.4.4 and earlier:** **setpersistentvar()** function not available
- **Version 9.4.5 and later:** **setpersistentvar()** function available

Description

The **setpersistentvar()** procedure will set a persistent variable in the local database, and will synchronize the value to other specified policy servers. If **Registry Name Service** is enabled, it will synchronize to all of the other policy servers in the **Service Group**. If **Registry Name Service** is not enabled, it will synchronize to all of the other policy servers specified by the **submitmasters** setting on the current policy server.

Syntax

```
boolean setpersistentvar(name,value,[expiry])
```

Arguments

name	Required. The name of the variable to be set. This can be any text string.
Value	Required. The value of the variable. This can be an integer, string, or list values.
Expiry	Optional. This is the UNIX epoch (in seconds) of the expiry date of the variable. Suitable values can be calculated using unixtimestamp with additional seconds calculated using Date/Time functions.

Return Values

A boolean indicating success or failure of the procedure.

Example

```
setpersistentvar("flag_" + submituser,true,unixtimestamp+300)
```

See Also

```
listpersistentvars, getpersistentvarint, getpersistentvarstring, getpersistentvarlist,  
delpersistentvar
```

getpersistentvarint

- **Version 9.4.4 and earlier:** `getpersistentvarint()` function not available
- **Version 9.4.5 and later:** `getpersistentvarint()` function available

Description

The `getpersistentvarint()` procedure will retrieve a persistent variable from the local database. If the variable does not exist, or has expired it will return the default `0`.

Syntax

```
int getpersistentvarint(name)
```

Arguments

name	Required. The name of the variable to be retrieved. This can be any text string.
-------------	--

Return Values

An integer containing the variable contents, or zero if the variable does not exist or has expired.

Example

```
myflag = getpersistentvarint("flag_" + submituser)
```

See Also

```
listpersistentvars, setpersistentvar, getpersistentvarstring, getpersistentvarlist, delpersistentvar
```

getpersistentvarstring

- **Version 9.4.4 and earlier:** `getpersistentvarstring()` function not available
- **Version 9.4.5 and later:** `getpersistentvarstring()` function available

Description

The **getpersistentvarstring()** procedure will retrieve a persistent variable from the local database. If the variable does not exist, or has expired it will return the default empty string "".

Syntax

```
string getpersistentvarstring(name)
```

Arguments

name	Required. The name of the variable to be retrieved. This can be any text string.
-------------	--

Return Values

A string containing the variable contents, or an empty string ("") if the variable does not exist or has expired.

Example

```
mystr = getpersistentvarstring("msg_" + submituser)
```

See Also

`listpersistentvars`, `setpersistentvar`, `getpersistentvarsint`, `getpersistentvarlist`, `delpersistentvar`

getpersistentvarlist

- **Version 9.4.4 and earlier:** `getpersistentvarlist()` function not available
- **Version 9.4.5 and later:** `getpersistentvarlist()` function available

Description

The **getpersistentvarlist()** procedure will retrieve a persistent variable from the local database. If the variable does not exist, or has expired it will return the default empty list {}.

Syntax

```
list getpersistentvarlist(name) sna
```

Arguments

name	Required. The name of the variable to be retrieved. This can be any text string.
-------------	--

Return Values

A list containing the variable contents, or an empty list if the variable does not exist or has expired.

Example

```
mylist = getpersistentvarlist("hosts_" + submituser)
```

See Also

```
listpersistentvars, setpersistentvar, getpersistentvarstring, getpersistentvarint, delpersistentvar
```

delpersistentvar

- **Version 9.4.4 and earlier:** `delpersistentvar()` function not available
- **Version 9.4.5 and later:** `delpersistentvar()` function available

Description

The `delpersistentvar()` procedure will delete a persistent variable from the local database. This deletion will be synchronized to the other specified policy servers.

Syntax

```
boolean delpersistentvar(wildcard)
```

Arguments

name	Required. A glob(3) wildcard limiting the deleted variables to those matched.
-------------	--

Return Values

A boolean indicating success or failure of the procedure.

Example

```
delpersistentvar("flag*")
```

See Also

```
listpersistentvars, setpersistentvar, getpersistentvarstring, getpersistentvarlist,  
getpersistentvarint
```

Glossary

accept	The term that is used to indicate that a secured task request has passed all security checks and may now be executed
built-in function	Predefined function that comes with Privilege Management for Unix & Linux
character string list	A sequence of zero or more characters enclosed in double (") or single (') quotation marks
character string list	An ordered list of character strings separated by commas and enclosed in curly braces ({})
checksum	A unique value that is derived from an application. It can be used to determine if an application has been modified since the checksum value was created
constant	A value that cannot be modified. A read-only variable is an example of a constant.
decimal integer	Base 10 numeric value (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
event log	The file that Privilege Management for Unix & Linux uses to record information about each user task request that Privilege Management for Unix & Linux processes.
environment variable	One of a set of Unix/Linux variables that define the environment that is passed to child processes
false	A read-only Privilege Management for Unix & Linux variable that is equal to an integer value of 0.
format command character	Used to insert variable values into character strings. Format command characters specify not only where to insert values, but also how to format the inserted values.
function	A stand-alone unit of security verification logic that performs a specific task. Procedures are generally used to implement repetitive tasks. The difference between a function and a procedure is that a function returns a value, whereas a procedure does not.
function scope	Determines whether a variable that is defined in one security policy function or procedure can be used by another security policy function or procedure. In Privilege Management for Unix & Linux, functions and procedures have a global scope, meaning that variables that are used in one function or procedure can be used by any other function or procedure.
global variable	A Privilege Management for Unix & Linux variable that applies to the Privilege Management for Unix & Linux system, rather than to a specific task request
hexadecimal integer	Base 16 integer value (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
index	A number that is used to access a specific element within a list variable
integer	A numeric value; a member of the set of both positive and negative whole numbers
I/O log	A Privilege Management for Unix & Linux log that captures the input (keystroke), output, and error streams for an interactive Unix/Linux session.
LDAP connection	A special data type that is used to pass parameters to and from Privilege Management for Unix & Linux LDAP functions
LDAP message	A special data type that is used to pass parameters to and from Privilege Management for Unix

	& Linux LDAP functions
logging variables	Contain information that controls Privilege Management for Unix & Linux logging activities
log host	Machine on which the Privilege Management for Unix & Linux log server runs. See pblogd .
manual accept	A task request can bypass security policy file processing and be manually accepted from the Privilege Management for Unix & Linux web user interface.
octal integer	Base 8 integer value (0, 1, 2, 3, 4, 5, 6, 7)
operator	A symbol that performs a specific mathematical, relational, logical or other special function
pblocald	The Privilege Management for Unix & Linux daemon that is responsible for initiating task execution. See run host
pblogd	When used, pblogd is responsible for saving log records to the appropriate event log files and I/O log files. pblogd is not a required Privilege Management for Unix & Linux component. If pblogd is not used, then the policy server host and the run host write their own log records. See log host .
pbmasterd	The main Privilege Management for Unix & Linux daemon. pbmasterd is responsible for determining whether requests should be allowed to run (accepted) or be terminated (rejected). See policy server host .
pbrun	The Privilege Management for Unix & Linux daemon that intercepts task requests and determines if the task is subject to security policy rules. If so, then pbrun passes the request on to the policy server host. See submit host .
policy server host	Machine on which the main Privilege Management for Unix & Linux daemon (pbmasterd) runs. See pbmasterd .
policy server security policy file	The security policy files invoked by policy server host to start security validation processing for a task
procedure	A stand-alone unit of security verification logic that performs a specific task. Procedures are generally used to implement repetitive tasks. The difference between a function and a procedure is that a function returns a value, whereas a procedure does not.
read-only variable	A variable whose value cannot be changed; also known as a constant
reject	The term used to indicate that a secured task request did not pass all security checks and so may not be executed
run host	Machine on which the Privilege Management for Unix & Linux task-execution daemon is run. See pblocald .
run variable	Modifiable version of a task information variable. These variables contain properties that affect task execution.
secured activity	An activity that is checked against Privilege Management for Unix & Linux security policy files, before it is executed, to verify that it adheres to all security policy rules. See secured task .
secured task	A task that is checked against Privilege Management for Unix & Linux security policy files, before they are executed, to verify that they adhere to all security policy rules. See secured activity .

security administrator	The person who is responsible for implementing a company's network security policy
security policy file	A file that contains the actual security checks that are used to determine whether a specific task should be accepted or rejected
security policy scripting language	A C-like, interpreted programming language that is used to create security policy files
security policy sub-file	A security policy file that is included by another security policy file. Security policy sub-files generally focus on specific areas of security verification processing.
security verification processing	The process of checking a task request against security policy files to determine if that task adheres to all security policy rules. The policy server host controls task verification processing.
special characters	Character combinations that are used in place of characters that cannot be typed directly with a keyboard
submit host	Machine on which the Privilege Management for Unix & Linux task-receiving component runs. See pbrun .
syslog	An interface that enables Privilege Management for Unix & Linux to access the Unix/Linux logging daemon
submitting user	The user who submitted the current task request
task information variable	One of a set of variables that contain information about the current task. There are two types of task information variables: read-only variables and run variables.
task verification processing	The process of checking a task request against security policy files to determine if that task adheres to all security policy rules. The policy server host controls task verification processing.
task request	Any request to run a job.
true	A read-only Privilege Management for Unix & Linux variable that is equal to an integer value of 1
unsecured task	A task request that is not checked against Privilege Management for Unix & Linux security policy files. Unsecured task requests are allowed to execute without first undergoing Privilege Management for Unix & Linux task verification processing.
user-defined variable	Variable that is used within a security policy file to store information during task security verification processing
user-written function	A stand-alone unit of security verification logic that performs a specific task. These units of code are written using the security policy scripting language. They are generally used to implement repetitive tasks. The difference between a function and a procedure is that a function returns a value, whereas a procedure does not.
user-written procedure	A stand-alone unit of security verification logic that performs a specific task. These units of code are written using the security policy scripting language. They are generally used to implement repetitive tasks. The difference between a function and a procedure is that a function returns a value, whereas a procedure does not.
variable data type	Defines the type of information that can be stored in a variable, as well as the types of operations that can be performed on a variable

variable scope

Determines whether another security policy file can use a variable that is defined in one security policy file. In Privilege Management for Unix & Linux, all variables have a global scope, meaning that after they are created, any security policy file can reference them.