One Identity Privilege Manager for Unix 7.1

Administration Guide

Legend

WARNING: A WARNING icon highlights a potential risk of bodily injury or property damage, for which industry-standard safety precautions are advised. This icon is often associated with electrical hazards related to hardware.

CAUTION: A CAUTION icon indicates potential damage to hardware or loss of data if instructions are not followed.

Privilege Manager for Unix Administration Guide
Updated - February 2021
Version - 7.1

Table of Contents

About this guide

Welcome to the *One Identity Privilege Manager for Unix Administration Guide*. This guide is intended for Windows, Unix*, Linux, and Macintosh system administrators, network administrators, consultants, analysts, and any other IT professional who will be installing and configuring Privilege Manager for Unix for the first time.

To simplify the installation and configuration of the Privilege Manager for Unix components, One Identity recommends that you install Management Console for Unix. This installation provides a mangement console, a powerful and easy-to-use tool that dramatically simplifies deployment, enables management of local Unix users and groups, provides granular reports on key data and attributes, and streamlines the overall management of your Unix, Linux, and macOS hosts. Please refer to the *One Identity Management Console for Unix Administration Guide* for instructions on installing and configuring the mangement console.

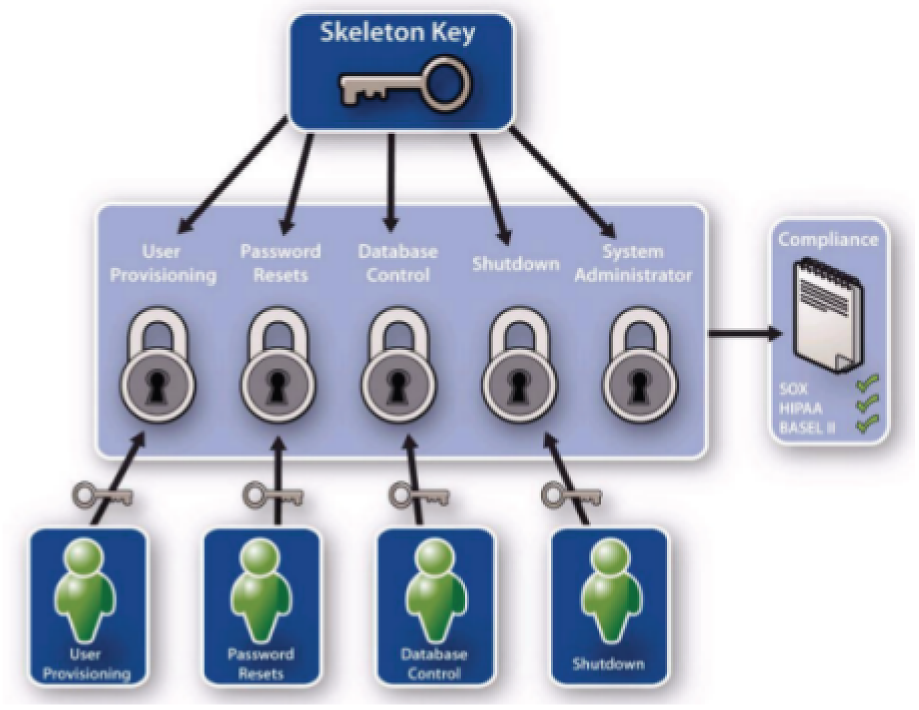
Of course, you can install Privilege Manager for Unix components without using the Management Console for Unix. This guide explains how to install and configure Privilege Manager for Unix components for the pmpolicy policy type directly from the command line.

* The term "Unix" is used informally throughout the Privilege Manager for Unix documentation to denote any operating system that closely resembles the trademarked system, UNIX.

Introducing Privilege Manager for Unix

Privilege Manager for Unix protects the full power of root access from potential misuse or abuse. Privilege Manager for Unix helps you to define a security policy that stipulates who has access to which root function, as well as when and where individuals can perform those functions. It controls access to existing programs as well as any purpose-built utilities used for common system administration tasks. With Privilege Manager for Unix, you do not need to worry about someone - whether inadvertently or maliciously - deleting critical files, modifying file permissions or databases, reformatting disks, or damaging UNIX systems in more subtle ways.

Figure 1: Privilege Manager for Unix protection



Within the UNIX world, common management tasks often require root access. Unfortunately, native root access is an all-or-nothing proposition. Consequently, as organizations add new users, fix printer queues, and perform other routine jobs on UNIX systems, the concern for control, compliance, and security grows. These routine tasks should not expose `root` passwords to those who don’t need them.

Privilege Manager for Unix also allows administrators to increase security as it protects sensitive data from network monitoring by encrypting `root` commands or sessions it controls. This capability includes control messages and input entered by users as they run commands through Privilege Manager for Unix.

What is Privilege Manager for Unix

Privilege Manager for Unix allows system administrators to safely share the power of `root` and other important accounts by partitioning them among users in a secure manner. System administrators can specify the circumstances under which users may run certain programs as `root` (or other privileged accounts).

The result is that you can safely assign the responsibility for such routine maintenance activities as adding user accounts and fixing line printer queues to the appropriate people without disclosing the `root` password. The full power of `root` is thus protected from potential misuse or abuse, reducing the risk of system administrator error or misuse (for example, modifying databases or file permissions, erasing disks, or more subtle damage).

Privilege Manager for Unix is capable of selectively recording all activities involving `root`, including all keyboard input and display output, if required. This indelible audit trail, combined with the safe partitioning of `root` functionality, provides an extremely secure means of sharing the power of `root`. A replay utility is provided to allow recorded sessions to be viewed at a later date. Privilege Manager for Unix can also require a checksum match before running any program, thereby guarding against virus or trojan horse attack on important accounts.

Additionally, Privilege Manager for Unix can provide an audit trail of:

- all users running commands on a particular host

  This may be required if, for example, the host is particularly sensitive, or because access to this host is chargeable.

- for a particular user

  This may be required if, for example, a temporary contractor has been provided with a login to a host, and the administrator needs to check which files the contractor has accessed.

Benefits of Privilege Manager for Unix

Privilege Manager for Unix is an important component of any heterogeneous organization's comprehensive compliance and identity management strategy. It perfectly complements UNIX identity integration initiatives using Authentication Services and compliance efforts enhanced through One Identity's Compliance Portal.

Some of the benefits that Privilege Manager for Unix brings to your organization are:

- enhanced security through fine-grained, policy-based control of `root` access
- compliance through compartmentalization of IT tasks that require `root` access
- visibility and control through automated, secure keystroke logging
- attainment of compliance and internal security standards through automated gathering of necessary data
- prevention of unapproved UNIX `root` activity

How Privilege Manager for Unix protects

Privilege Manager for Unix protects your systems by:

- partitioning `root` (and other important account) functionality to allow many different users to carry out system administration tasks
- creating an indelible audit trail of these administration tasks

Partition `root` safely

The ability to partition system administration actions without compromising the security of the `root` account is an extremely powerful one. Privilege Manager for Unix allows you, the system administrator, to set policies to determine whether and when a user request to run a program is accepted or rejected.

Through Privilege Manager for Unix, each user can request that a specific program is run on a specific machine as `root` (or as another important account such as `oracle` or `admin`). Privilege Manager for Unix evaluates the request; if accepted, it runs the program, locally or across a network, on behalf of the user.

With Privilege Manager for Unix, Helpdesk personnel can replace passwords for users or reinstate user accounts. Project members can clear a jammed line printer queue, kill hung programs, or reboot certain machines. Administration staff can print or delete resource usage logs or start backups.

Through partitioning, Privilege Manager for Unix allows different users to perform the `root` actions for which they are responsible, but prevents them from performing actions for which they do not have authorization.

Privilege Manager for Unix lets you specify:

- which users can perform a particular task
- which tasks can be run through the system
- when the user can perform the task
- which machine can perform a task

- from which machine the user may initiate a request to perform the task
- whether another user’s permission (in the form of a password) is required before the task is started
- decisions to be made by a program that you supply, which Privilege Manager for Unix calls to determine if a request should be accepted or rejected
- many other miscellaneous properties of requests

Create an indelible audit trail

Privilege Manager for Unix can record all activity which passes through it, down to the keystroke level. The power to accurately log `root` and other account activities in a safe environment allows you to implement a secure system administration regime with an indelible audit trail. You always know exactly what is happening in `root`, as well as who did it, when it happened and where.

Since `root` can modify any file, you must ensure that Privilege Manager for Unix logs are indelible. You can configure Privilege Manager for Unix to receive user requests from the submitting machine, run tasks on the execution machine, and log all activities on a third, very secure machine. See the illustration in How Privilege Manager for Unix works on page 6.

You can make the machine containing the log files physically inaccessible to users and isolated from remote login over the network. In addition, you can print the logs to hard copy on a secure printer or recorded to a WORM drive.

You can also assign this secure machine a `root` password which is unknown to the person who has physical access to it, but known to someone else without physical access. Two people would have to conspire to subvert system security.

You may use these and other techniques to achieve a high degree of security around Privilege Manager for Unix itself, as well as the logs of `root` activity that it creates.

Encryption

You can encrypt all communication among Privilege Manager for Unix programs, and between the user and the application being run to guard against network snooping or spoofing.

Privilege Manager for Unix supports the following encryption algorithms:

- AES
- Kerberos
- TripleDES and DES

Set the encryption method in the `/etc/opt/quest/qpm4u/pm.settings` file. See the `encryption` setting in PM settings variables on page 286 for details.

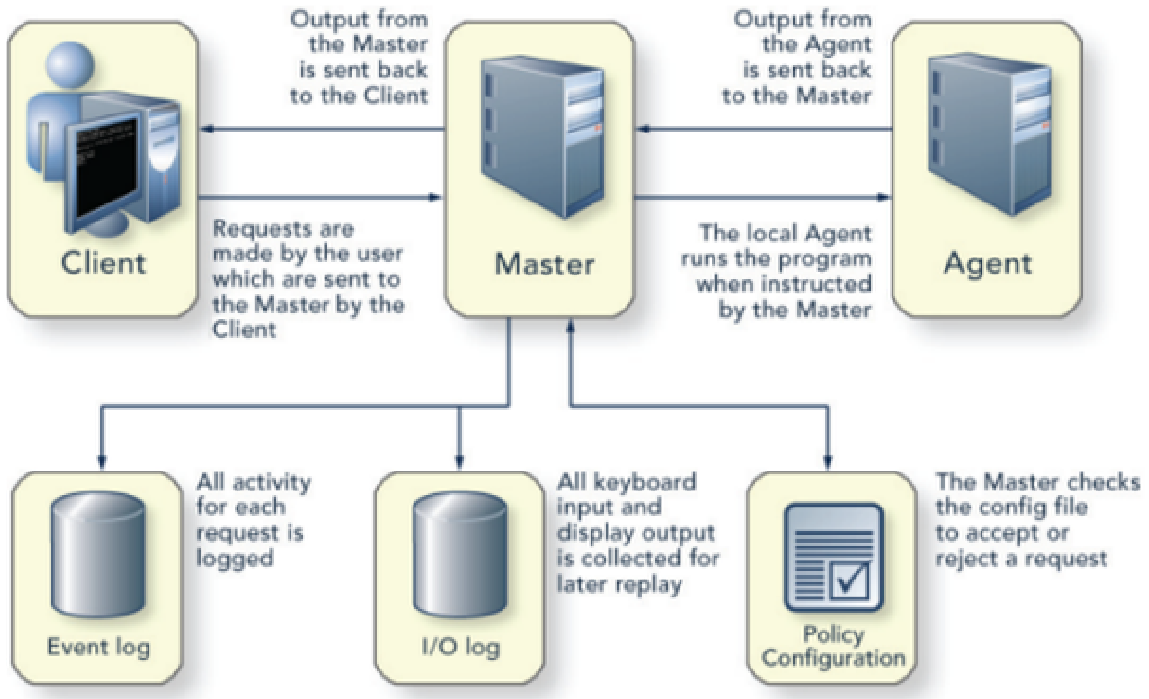
How Privilege Manager for Unix works

The three main Privilege Manager for Unix components are:

- **The Client**: The client is effectively the user who runs a command from their local machine by simply performing commands as `root` using the `pmrun` prefix.
- **The Policy Server**: The policy server checks all commands with the policy file to ensure that the user is allowed to run the command, it then passes the command on to the agent for action. The policy server also logs the output result (that is, whether the command was successfully actioned or not), whether you enable keystroke logging or not.

  If you enable keystroke logging, it creates a much more detailed set of log files. The input/output log stores everything from keystrokes to input and output data. The event log purely records all of the requests made and their result.
- **The Agent**: The agent performs the commands which are issued from the policy server and passes the result back to the client.

Figure 2: Privilege Manager for Unix components


Client
Output from the Master is sent back to the Client
Requests are made by the user which are sent to the Master by the Client
Master
Output from the Agent is sent back to the Master
The local Agent runs the program when instructed by the Master
Agent
Event log
All activity for each request is logged
I/O log
All keyboard input and display output is collected for later replay
Policy Configuration
The Master checks the config file to accept or reject a request

Privilege Manager for Unix comprises four main programs:

- pmrun
- pmmasterd

- pmlocald
- pmtunneld

Users submit their requests to run certain programs through Privilege Manager for Unix using `pmrun`. For each request, the user may specify a program name and optionally a host on which the program will run.

The configuration file policy server master daemon (`pmmasterd`) examines each user request and either accepts or rejects it based upon information in the Privilege Manager for Unix configuration file. You can have multiple `pmmasterd` daemons on the network to avoid having a single point of failure.

All Privilege Manager for Unix administrative tools, including the configuration commands are located in the `/opt/quest/sbin` directory.

Policy configuration file (pmpolicy security policy)

Users submit their requests to run certain programs as `root`, or another privileged account, through Privilege Manager for Unix using `pmrun`. The policy server daemon, `pmmasterd`, examines each request from `pmrun`, and either *accepts* or *rejects* it based upon the policies specified in the policy file.

The Privilege Manager for Unix configuration file (also referred to as the pmpolicy security policy) contains the security policy that the policy server master daemon (`pmmasterd`) considers when it *accepts* or *rejects* user requests. The configuration file can specify constraints based on certain attributes, such as:

- Username
- Group membership
- Application name
- Application arguments
- Environment variable values
- Umask (file permissions)
- Nice value (priority of jobs run)
- Working directory from which the request may be made
- Host from which a request can be submitted (submitting host)
- tty from which a request is submitted
- Host from which the request will be run (execution host)
- A remote, dedicated host to store iologs and/or eventlogs
- Time of day and day of week that the user is allowed to run the application
- Exit status or output of any specified program to be run as part of the decision-making process

- A challenge to the user to type in one or more specified user passwords (requires on-the-spot approval from those users, such as supervisors or managers)
- Whether the program being requested has a checksum that matches the one stored for that application in the configuration file (protects against possible virus or trojan horse attack)
- Store all information for each request in a log file
- Record all keystrokes and/or output in a dribble file
- Some other miscellaneous job properties

If Privilege Manager for Unix accepts the request, the Privilege Manager for Unix local daemon (`pmlocald`) runs the application program as the `runuser` selected in the policy file, piping all input/output back to the user's terminal. In addition, you can specify in the configuration file that you want to store all information for each request in a log file, and optionally record all keystrokes, output, or both, in an I/O file for later replay. You can replay the file in real time, so you can observe the commands as they are issued.

You can restrict responses to a small designated range of reserved port numbers by setting parameters in `/etc/opt/quest/qpm4u/pm.settings`. This enhances the security of communications between `pmlocald` and `pmmasterd` when the two must communicate across a firewall. See PM settings variables on page 286 for details.

Privilege Manager for Unix utilizes NAT (Network Address Translation) to further restrict responses to a single designated port when `pmlocald` and `pmmasterd` must communicate across a firewall.

You can issue commands either in the foreground or background. If you run them in the background, you can continue to use the same shell process to issue additional commands. See Privilege Manager for Unix shells on page 116 for details.

The policy file is:

- Located on the policy server daemon host
- Created in `pm.conf`

  By default, the policy file is named `pm.conf` and is located in the directory specified by `policyfile`. If the full path name for the `pm.conf` file is not specified in `policyfile`, the path is relative to `policydir`.
- Owned by `root`

Only `root` can have write permission for the configuration file. Otherwise, a user might gain illegal access to the `root` account through modification of the file. To prevent someone from replacing the entire `/etc` directory or its contents, both `/` and `/etc` have permission modes that do not allow users to modify them.

The configuration file contains statements and declarations in a language specifically designed to express policies concerning the use of `root` and other controlled accounts.

For example, if your policy is: Allow user robyn to run the `/bin/passwd` program as `root` on the `galileo` machine Monday through Friday, during office hours (8:00 a.m. to 5:00 p.m.), add the following to your policy file:

```
weekdays={"Mon", "Tue", "Wed", "Thu", "Fri"}; if (user=="robyn" && command=="passwd"
&& host=="galileo" && timebetween(800, 1700) && dayname in weekdays) {
runuser="root"; runcommand="/bin/passwd"; accept; }
```

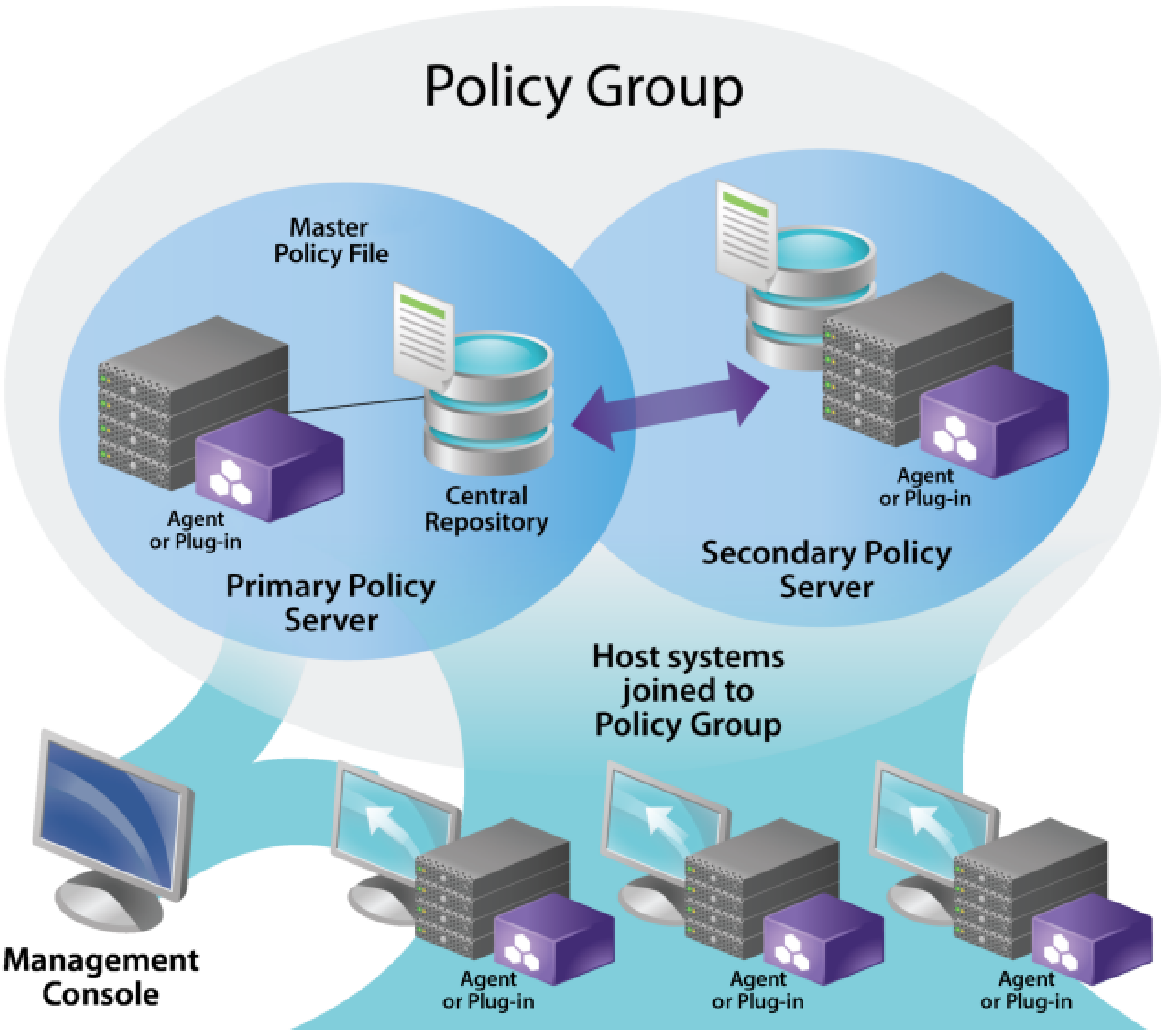
Do not use a leading zero for any time between 00:00 and 9:59 a.m. For example, when specifying 7:00 a.m., use 700 rather than 0700. Specify 12:30 am as 30 or 2430. Privilege Manager for Unix interprets numbers with leading zeroes as octal numbers: 0700 octal is 560 decimal, which is not a valid time.

Policy group

A policy group is a group of one or more policy servers – one primary server and any number of secondary servers. You can configure multiple policy servers in a policy group to share a common configuration for load balancing and redundancy.

Policy servers are responsible for evaluating the security policy and accepting or rejecting the agent request based on the constraints in the security policy. A policy group is one or more policy servers which have been configured to share a common policy.

Figure 3: Policy group



When the first policy server in the group is configured, it becomes the primary policy server and sole member of the policy group. To support load balancing and redundancy, you may add secondary policy servers to the policy group.

If a policy server becomes unavailable for any reason, hosts joined to the group will find the next available server in the policy group to service their requests. Any failover is transparent to the hosts, as the same policy is enforced by all policy servers within the policy group.

The primary policy server hosts the master copy of the policy from which the secondary servers receive updates. You can initiate changes to the policy from any policy server using the `pmpolicy` command. Once completed, the changes are committed to the master copy, and policy servers are automatically updated.

See pmpolicy on page 435 for more information about the syntax and usage of this command.

Planning Deployment

Before you run the installer, consider the following questions:

1. Which machines in your network will run policy servers?

   If you only plan to use one policy server for an entire network, it should be the most reliable and secure machine.

   You can specify multiple policy servers to avoid having a single point of failure.

   If more than 150 users will be using a single `pmmasterd` for validation, you will want to have multiple policy servers to avoid a UNIX network resource bottleneck. Plan to have a maximum of 150 users validating at a single policy server.

2. Which machines will be managed hosts?

   Only those hosts running the local daemon (PM Agent package) may receive and run Privilege Manager for Unix requests. See pmlocald on page 418 for details.

   One Identity recommends that you initially specify one policy server and three or four local hosts when you first install and experiment with Privilege Manager for Unix.

3. What level of protection do you require?

   If you require greater protection, you can select an encryption level such as AES, or a dedicated encryption system such as Kerberos. When configuring Privilege Manager for Unix in interactive mode, you are asked if you are using Kerberos. If you are using Kerberos, Privilege Manager for Unix automatically uses Kerberos for encryption.

   You can configure the policy file to require a checksum match to authorize program execution. If configured in the policy, Privilege Manager for Unix runs the program only if its checksum matches that configured in the policy file. By default, it uses a CRC algorithm, but you can configure the MD5 algorithm instead by setting the keyword `checksumtype` to MD5 in `pm.settings`.

4. Which port numbers should `pmmasterd` and `pmlocald` use to listen for network requests?

   Choose numbers that do not conflict with other numbers in the `/etc/services` file. Ensure these entries are propagated to all machines accessing Privilege Manager for Unix.

5. Which directory should contain the Privilege Manager for Unix log files?

   By default, the log files are placed in `/var/adm` or `/var/log` depending on the host architecture. The installer allows you to change the directory by specifying command line options to the Privilege Manager for Unix daemons. The partition needs to contain enough space for log files to increase in size.

System requirements

Prior to installing Privilege Manager for Unix, ensure your system meets the minimum hardware and software requirements for your platform.

Table 1: Hardware and software requirements

Component	Requirements
Operating systems	See Supported platforms to review a list of platforms that support Privilege Manager for Unix clients.
Disk space	80 MB of disk space for program binaries and manuals for each architecture. Considerations: • At a minimum, you must have 80 MB of free disk space. The directories in which the binaries are installed must have sufficient disk space available on a local disk drive rather than a network drive. Before you install Privilege Manager for Unix, ensure that the partitions that will contain `/opt/quest` have sufficient space available. • Sufficient space for the keystroke logs, application logs, and event logs. The size of this space depends on the number of servers, the number of commands, and the number of policies configured. • The space can be on a network disk drive rather than a local drive. • The server hosting Privilege Manager for Unix must be a separate machine dedicated to running the `pmmasterd` daemon.
SSH software	You must install and configure SSH client and server software on all policy server hosts. You must enable access to SSH as the `root` user on the policy server hosts during configuration of the policy servers. Both OpenSSH 4.3 (and later) and Tectia SSH 6.4 (and later) are supported.

Component	Requirements
Processor	Policy Servers: 4 cores
Policy Servers: 4GB	

Supported platforms

The following table provides a list of supported platforms for Privilege Manager for Unix clients.

Table 2: Linux supported platforms — server and client

Platform	Version	Architecture
Amazon Linux AMI		x86_64
CentOS Linux	5, 6, 7, 8	Current Linux architectures: s390, s390x, PPC64, PPC64le, ia64, x86, x86_64, AARCH64
Debian	Current supported releases	x86_64, x86, AARCH64
Fedora Linux	Current supported releases	x86_64, x86, AARCH64
OpenSuSE	Current supported releases	x86_64, x86, AARCH64
Oracle Enterprise Linux (OEL)	5, 6, 7, 8	Current Linux architectures: s390, s390x, PPC64, PPC64le, ia64, x86, x86_64, AARCH64
Red Hat Enterprise Linux (RHEL)	5, 6, 7, 8	Current Linux architectures: s390, s390x, PPC64, PPC64le, ia64, x86, x86_64, AARCH64
SuSE Linux Enterprise Server (SLES)/Work-station	11, 12, 15	Current Linux architectures: s390, s390x, PPC64, PPC64le, ia64, x86, x86_64, AARCH64
Ubuntu	Current supported releases	x86_64, x86, AARCH64

Table 3: Unix and Mac supported platforms — client

Platform	Version	Architecture
Apple macOS	10.12, 10.13, 10.14, 10.15	x86_64
FreeBSD	11.x, 12.x	x86_64
HP-UX	11.31	PA, IA-64
IBM AIX	7.1 Technology Level 3 and higher, 7.2	Power 4+
Solaris	10.x, 11.x	SPARC, x64

Reserve special user and group names

Reserve the following names for Privilege Manager for Unix usage:

- `pmpolicy` (user and group)
- `pmlog` (group)

For more information, see Reserve special user and group names on page 26.

Required privileges

You will need `root` privileges to install Privilege Manager for Unix software. Either log in as `root` or use the `su` program to acquire `root` privileges. Due to the importance of the `root` account, Privilege Manager for Unix carefully protects the system against certain accidental or deliberate situations that might lead to a breach in security. For example, if Privilege Manager for Unix discovers that its configuration files are open to modification by non-root users, it will reject all job requests. Furthermore, all Privilege Manager for Unix directories back to the / directory are checked for security in the same way, to guard against accidental or deliberate replacement.

Estimating size requirements

Keystroke and event log disk space requirements

The amount of disk space required to store keystroke logs will vary significantly based on the amount of terminal output generated by the user's daily activity and the level of logging configured. An average Privilege Manager for Unix keystroke log will contain an additional

4KB of data on top of the amount of data displayed to the user's terminal. Taking an average of the amount of terminal output generated by a few users over the course of a normal day would allow for an approximate estimation to be calculated. For example, a developer using a `vi` session throughout the day may generate 200KB of terminal output. A team of 200 developers each generating a similar amount of terminal output per working day could be expected to use 31GB of disk space over a three-year period [204 (200 + 4KB) x 200 (developers) x 260 (working days) x 3 (years) = 31,824,000].

The level of logging can also be configured to reduce the overhead on the Masters. For example, some customers only log the user's input (key presses) which will dramatically reduce the amount of logging.

Event log entries will typically use 4-5KB of storage per event, but may vary slightly depending on the data stored in the events. For example, events might be slightly larger for users that have lots of environment variables defined. Taking an average of the number of events that occur over the course of a normal day should allow you to estimate the disk space requirements for event logs. For example, if the same team of developers generate 1,000 events in a normal working day, they would be expected to use nearly 4GB of disk space over a three-year period [5 (KB) * 1000 (events) * 260 (days) * 3 (years) = 3,900,000].

Policy server deployment requirements

The following recommendations are only provided as a rough guideline. The number of policy servers required for your environment may vary greatly depending on usage.

- One policy server is suitable for small test environments with less than 50 hosts.
- Production environments should have a minimum of two policy servers.
- Add an additional policy server for every 150-200 Privilege Manager for Unix hosts.
- Additional policy servers may be required to support geographically disparate locations.

Privilege Manager for Unix licensing

Privilege Manager for Unix 7.1 licensing options include:

30-day evaluation licenses

Privilege Manager for Unix evaluation license allows you to manage unlimited PM Agent hosts for 30 days.

Commercial licenses

A **PM Policy** license is required for Privilege Manager for Unix features.

Although licenses are allocated on a per-agent basis, you install the licenses on Privilege Manager for Unix policy servers.

The `pmlicense` command allows you to display current license information, update a license (an expired one or a temporary one before it expires) or create a new one. See Installing licenses on page 53 or Displaying license usage on page 53 for more examples of using the `pmlicense` command.

Deployment scenarios

You can deploy Privilege Manager for Unix software within any organization using UNIX and/or Linux systems. Privilege Manager for Unix offers a scalable solution to meet the needs of the small business through to the extensive demands of the large or global organization.

There is no right or wrong way to deploy Privilege Manager for Unix, and an understanding of the flexibility and scope of the product will aid you in determining the most appropriate solution for your particular requirements. This section describes the following sample implementations:

- a single host installation
- a medium-sized business installation
- a large business installation
- an enterprise installation

Configuration options

Decide which of the following configurations you want to set up:

1. **Primary Server Configuration**: Configure a single host as the primary policy server hosting the security policy for the policy group using either the pmpolicy (Privilege Manager for Unix) or sudo (Safeguard for Sudo) policy type. See Security policy types on page 57 for more information about these policy types.

   If you are configuring the primary policy server using the sudo policy type, see the *One Identity Privilege Manager for Sudo Administration Guide*.
2. **Secondary Server Configuration**: Configure a secondary policy server in the policy server group to obtain a copy of the security policy from the primary policy server.
3. **PM Agent Configuration**: Join a Privilege Manager for Unix Agent host to a pmpolicy server group.

   Policy servers can only be joined to policy groups they host (that is, manage). You cannot join a Sudo Plugin host to a pmpolicy server group or the PM Agent host to a sudo policy server group.

Single host deployment

A single-host installation is typically appropriate for evaluations, proof of concept, and demonstrations of Privilege Manager for Unix. This configuration example installs all of the components on a single UNIX/Linux host, with protection offered only within this single host. All logging and auditing takes place on this host.

Medium business deployment

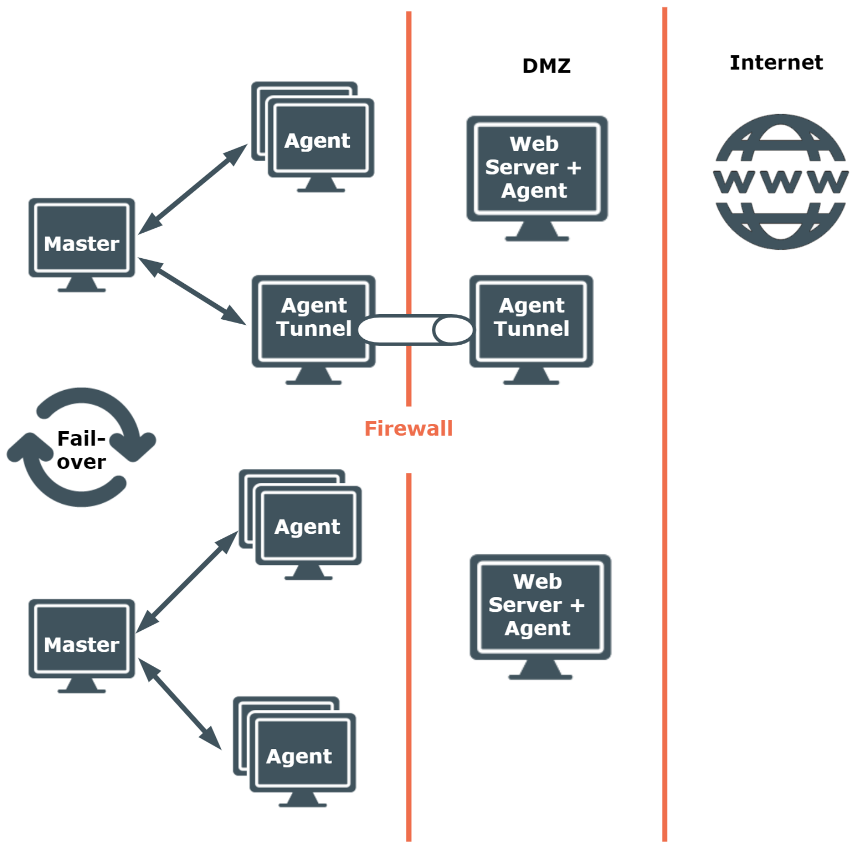
The medium business model is suitable for small organizations with relatively few hosts to protect, all of which may be located within a single data center.

This configuration example comprises multiple UNIX/Linux hosts located within the SME space and one or more web servers located in a DMZ.

The tunneling feature (pmtunneld on page 465), enables Privilege Manager for Unix to control privileged commands on the web servers across a firewall, within the DMZ. This configuration significantly reduces the number of open ports at the firewall.

Multiple policy server components (pmmasterd on page 433) are installed in a failover configuration, with groups of agents balanced between the policy servers. If a policy server is unavailable for any reason, the agents will failover to the alternative policy server.

Figure 4: Medium business implementation: Minimum 2 Masters and Circa 100 Agents

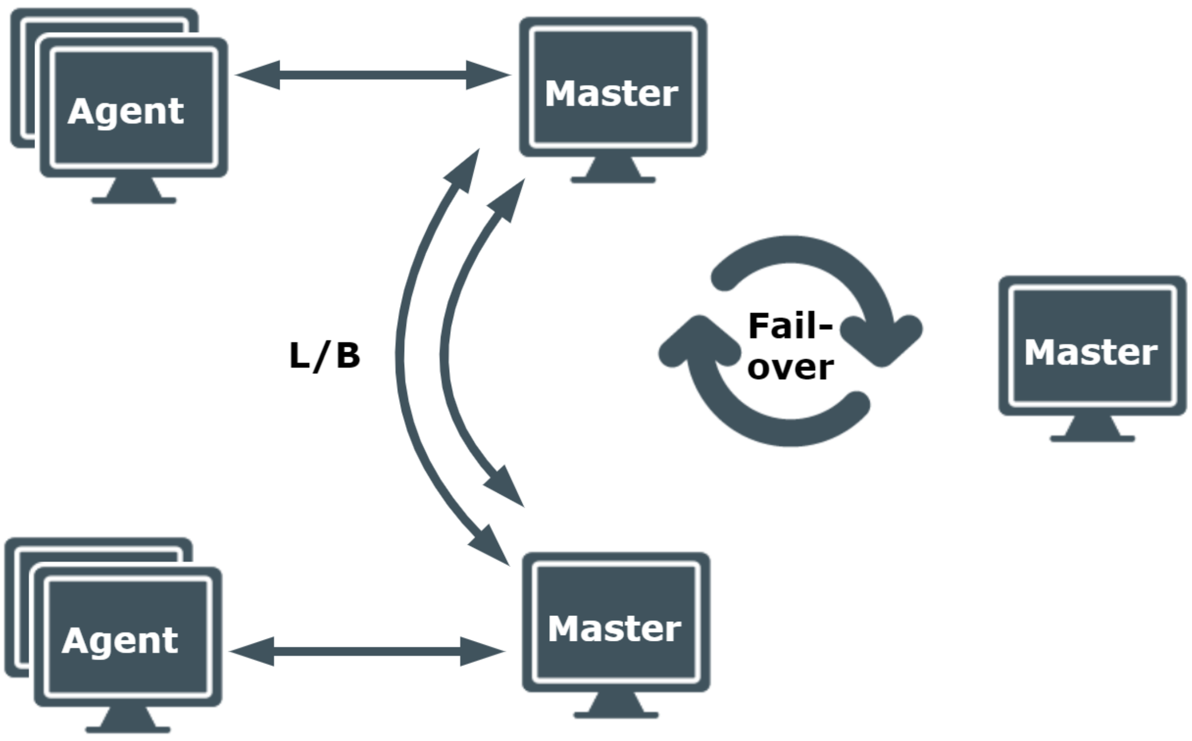


Large business deployment

This is an example of how a large business might deploy Privilege Manager for Unix. Some global companies prefer to fragment their requirement and deploy multiple instances as shown in the medium-sized business model.

This example comprises three policy servers, two are balancing the load of multiple agents. This may be necessary if there is a high level of audit and/or a significant volume of requested elevated privilege. Further, there is an additional policy server configured as a failover should one or both policy servers become unavailable.

Figure 5: Large business implementation: Minimum 3 Masters and less than 1000 Agents



Enterprise deployment

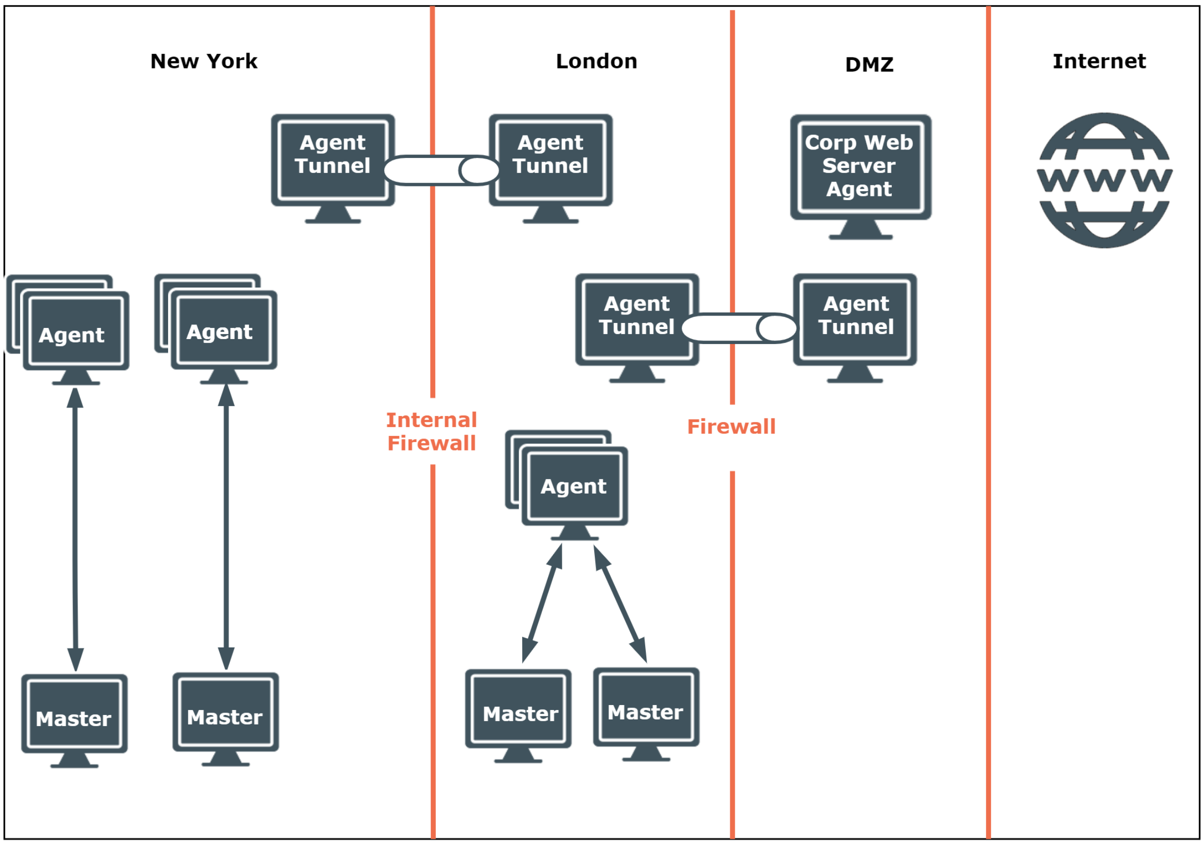
This example is based on an organization with offices in London and New York. Again, as with the medium-sized business example, the web servers and corporate web-based applications reside in a DMZ. The requirement to run commands at an elevated level from inside the firewall remains.

Access to the web server and web applications is predominantly, but not exclusively, from the London office. Privilege Manager for Unix tunnelling components are used to breach the firewall to the DMZ.

In addition, internal firewalls are located between the offices in London and New York, and tunneling components are deployed to enable access from office to office and indeed from anywhere to the DMZ.

Within each office, multiple policy servers are configured for load balancing, with each policy server serving a number of agents.

Figure 6: Enterprise deployment implementation: Minimum 4 Masters and 1000 Agents and above



You can extend each of the models described above by, for example, adding more policy servers, configuring additional load balancing, assigning dedicated audit, logging and reporting servers. The models provide a small indication of the flexibility and modular way in which you can configure and implement Privilege Manager for Unix to meet the precise requirements of any size business.

Installation and Configuration

This is an overview of the steps necessary to set up your environment to use Privilege Manager for Unix software:

To configure a primary policy server

1. Check the server for installation readiness.
2. Install the Privilege Manager for Unix policy server package.
3. Configure the primary policy server.
4. Join the primary policy server to policy group.

To configure a secondary policy server

1. Check the host for installation readiness.
2. Install the Privilege Manager for Unix policy server package.
3. Configure the secondary policy server.
4. Join the PM Agent to the secondary policy server.

To install the PM Agent on a remote host

1. Check the remote host for installation readiness.
2. Install the Privilege Manager for Unix software on the remote host.
3. Join the PM Agent to the policy server.

The following topics walk you through these steps.

Downloading Privilege Manager for Unix software packages

To download the Privilege Manager for Unix software packages

1. Go to https://support.oneidentity.com/privilege-manager-for-unix .
2. On the **Product Support - Privilege Manager for Unix** page, click **Software Downloads** under **Self Service Tools** in the left pane.
3. On the **Privilege Manager for Unix - Download Software** page, click **Download** to the right of the version to be downloaded.

   See Installation Packages on page 468 for more information about Privilege Manager for Unix native platform install packages.
4. Read the License Agreement, select the **I have read and accept the agreement** option, and click **Submit**.
5. Download the relevant package from the web page. The Privilege Manager for Unix server package includes the PM Agent and the Sudo Plugin components.

Quick start and evaluation

To simplify the installation and configuration of the Privilege Manager for Unix components, One Identity recommends that you install One IdentityManagement Console for Unix. Management Console for Unix provides a web-based mangement console, a powerful and easy-to-use tool that dramatically simplifies deployment, enables management of local Unix users and groups, provides granular reports on key data and attributes, and streamlines the overall management of your Unix, Linux, and macOS hosts.

You can download the Management Console for Unix install package from the same **Download Software** page where you downloaded the Privilege Manager for Unix software packages.

To test Privilege Manager for Unix, you must set up at least one primary policy server and one remote host system configured with the PM Agent.

Installing the Management Console

Management Console for Unix makes it easy for you to centrally manage a policy file on a primary policy server.

You can install the mangement console on Windows, Unix, or macOS computers. Each hosting platform prompts for similar information.

The following install files are located on the Privilege Manager for Unix distribution media under `console | server`:

- `ManagementConsoleForUnix_unix_2_5_2.sh` - for Unix and Linux
- `ManagementConsoleForUnix_windows_2_5_2.exe` - for Windows
- `ManagementConsoleForUnix_windows-x64_2_5_2.exe` - for Windows

The *One IdentityManagement Console for Unix Administration Guide* contains detailed instructions for installing the mangement console on all of these platforms. Use the following procedure to install the mangement console on a Unix computer from the command line using the installation script:

To install the mangement console on a Unix platform

1. Log in and open a root shell.
2. Mount the installation media and navigate to `console | server`.
3. Run the following command from the Unix command line as `root`:

   ```
   # sh ManagementConsoleForUnix_unix_2_5_2.sh
   ```

   You can optionally use one of these options:

   - `-q` option (quiet mode) to automatically accept all the default settings.
   - `-c` option (console mode) to prompt you for information interactively.

   Using no option starts the installer in a graphical user interface if you have an X server, making the installation experience similar to running it from the Windows autorun.

   In console mode, it asks you for the following information:

   a. Enter **1** to accept the user agreement.
   b. Enter the SSL Port number or press **Enter** to accept the default of 9443.
   c. Enter the Non-SSL Port number or press **Enter** to accept the default of 9080.

   The install wizard extracts and downloads the files, configures and starts the service, and so forth. On Unix, the install location is `/opt/quest/mcu` and you cannot specify an alternate path.

Uninstalling the Management Console

The default for the uninstaller is to remove everything. Before you uninstall Management Console for Unix, if you plan to re-install Management Console for Unix and want to preserve your data, backup your application database. The application database contains information about the hosts, settings, users, groups, passwords, and so forth.

By default, the database directory is at: `/var/opt/quest/mcu`.

To uninstall the mangement console from Unix

1. Run the following command as `root`:

   ```
   # /opt/quest/mcu/uninstall
   ```

   You can optionally use one of the following options with the `uninstall` command:

   - `-q` option (quiet mode) to automatically accept all the default settings, including removing the application database and logs.
   - `-c` option (console mode) to prompt you for information interactively.

   Using no option starts the installer in a graphical user interface.

2. If in console mode:

   a. Confirm that you want to remove Management Console for Unix.

   b. Confirm whether you want to remove the application database and application logs.

      This option is useful if you plan to re-install Management Console for Unix and want to preserve your data. The default for the uninstaller is to remove everything.

   The wizard uninstalls Management Console for Unix

Configure a Primary Policy Server

The first thing you must do is install and configure the host you want to use as your primary policy server.

Checking the server for installation readiness

Privilege Manager for Unix comes with a Preflight program that checks to see if your system meets the install requirements.

To check for installation readiness

1. Log on as the `root` user.
2. Change to the directory containing the `qpm-server` package for your specific platform.

   For example, on a 64-bit Red HatLinux, run:

   ```
   # cd server/linux-x86_64
   ```

3. To ensure that the `pmpreflight` command is executable, run:

   ```
   # chmod 755 pmpreflight
   ```

4. To verify your primary policy server host meets installation requirements, run:

   ```
   # sh pmpreflight.sh --server
   ```

   Running `pmpreflight.sh --server` performs these tests:

   - Basic Network Conditions:
     - Hostname is configured
     - Hostname can be resolved
     - Reverse lookup returns its own IP
   - Privilege Manager for Unix Server Network Requirements:
     - Policy server port is available (TCP/IP port 12345)
   - Privilege Manager for Unix Prerequisites:
     - SSH keyscan is available

5. Resolve any reported issues and rerun `pmpreflight` until all tests pass.

TCP/IP configuration

Privilege Manager for Unix uses TCP/IP to communicate with networked computers, so it is essential that you have TCP/IP correctly configured. If you cannot use programs such as `ssh` and `ping` to communicate between your computers, then TCP/IP is not working properly; consult your system administrator to find out why and make appropriate changes.

Ensure that your host has a statically assigned IP address and that your host name is not configured to the loopback IP address 127.0.0.1 in the `/etc/hosts` file.

Firewalls

When the agent and policy server are on different sides of a firewall, Privilege Manager for Unix needs a number of ports to be kept open. By default, Privilege Manager for Unix can use ports in the 600 to 31024 range, but when using a firewall, you may want to limit the ports that can be used.

You can restrict Privilege Manager for Unix to using a range of ports in the reserved ports range (600 to 1023) and the non-reserved ports range (1024 to 65535). We recommend that a minimum of six ports are assigned to Privilege Manager for Unix in the reserved ports range and twice that number of ports are assigned in the non-reserved ports range.

Use the `setreserveportrange` and `setnonreserveportrange` settings in the `/etc/opt/quest/qpm4u/pm.settings` file to open the ports in the required ranges. See PM settings variables on page 286 for details.

If configuring Privilege Manager for Unix to use NAT (Network Address Translation), you may need to configure the `pmtunneld` component. See Configuring firewalls on page 141 for more information about using Privilege Manager for Unix with NAT and restricting port numbers.

Hosts database

Ensure that each host on your network knows the names and IP addresses of all other hosts. This information is stored either in the `/etc/hosts` file on each machine, or in NIS maps or DNS files on a server. Whichever you use, ensure all host names and IP addresses are up-to-date and available.

Privilege Manager for Unix components must be able to use forward and reverse lookup of the host names and IP addresses of other components.

Reserve special user and group names

It is important for you to reserve the following special user and group names for Privilege Manager for Unix usage:

- Users: `questusr`, `pmpolicy`
- Groups: `questgrp`, `pmpolicy`, `pmlog`

The `questusr` account is a user service account created and used by Management Console for Unix to manage Privilege Manager for Unix policy and search event logs. It is a non-privileged account (that is, it does not require root-level permissions) that is used by the console to gather information about existing policy servers in a read-only fashion. The mangement console does not use `questusr` account to make changes to any configuration files. `questgrp` is the primary group (gid) for `questusr`.

The `pmpolicy` user is created on a primary or secondary server. It is a non-privileged service account (that is, it does not require root-level permissions) that is used to synchronize the security policy on policy servers.

The `pmlog` and `pmpolicy` groups are used to control access to log files and the security policy, respectively.

Applications and file availability

Since you can use Privilege Manager for Unix to run applications on remote machines, ensure that the applications and the files that they access are available from those machines. Typically, you can use a product such as NFS (supplied with most UNIX operating systems) to make users' home directories and other files available in a consistent location across all computers.

Policy server daemon hosts

Privilege Manager for Unix requires that you choose a host to act as the policy server. This machine will run the `pmmasterd` daemon and must be available to manage requests for the whole network.

Run the policy server daemon on the most secure and reliable node. To maximize security, ensure the computer is physically inaccessible and carefully isolated from the network.

The policy server requires that the `pmmasterd` port (TCP/IP port 12345, by default) is available, and that PM Agent hosts joined to the policy server are able to communicate with the policy server on this network port.

You can run multiple policy servers for redundancy and stability. Privilege Manager for Unix automatically selects an available policy server if more than one is on the network. For now, choose one machine to run `pmmasterd`. See pmmasterd on page 433 for more information.

Local daemon hosts

Each machine that runs requests using Privilege Manager for Unix must run a `pmlocald` daemon. Typically you will run `pmlocald` on all your machines. See pmlocald on page 418 for more information.

Installing the Privilege Manager for Unix packages

After you make sure your primary policy server host meets the system requirements, you are ready to install the Privilege Manager for Unix packages.

To install the Privilege Manager for Unix packages

1. From the command line of the host designated as your primary policy server, run the platform-specific installer. For example, run:

   ```
   # rpm --install qpm-server-*.rpm
   ```

   The Solaris server has a filename that starts with `QSFTpmsrv`.

   When you install the `qpm-server` package, it installs all three Privilege Manager for Unix components on that host: the Privilege Manager for Unix Policy Server, the PM Agent, and the Sudo Plugin.

For details instructions on installing and configuring Safeguard for Sudo, see the *One Identity Safeguard for Sudo Administration Guide*.

Modifying PATH environment variable

After you install the primary policy server, you may want to update your PATH to include the Privilege Manager for Unix commands.

To modify the user's PATH environment variable

1. If you are a Privilege Manager for Unix administrator, add these quest-specific directories to your PATH environment:

   ```
   /opt/quest/bin:/opt/quest/sbin
   ```

2. If you are a Privilege Manager for Unix user, add this path to your PATH environment:

   ```
   /opt/quest/bin
   ```

Configuring the primary policy server for Privilege Manager for Unix

Once you install the Privilege Manager for Unix server packages, the next task is to configure the primary policy server. The first policy server you setup is the *primary policy server*.

To configure the primary policy server for a pmpolicy type

1. From the command line of the primary policy server host, run:

   ```
   # /opt/quest/sbin/pmsrvconfig -m pmpolicy
   ```

   The `pmsrvconfig` command supports many command-line options; see pmsrvconfig on page 458 for details or run `pmsrvconfig` with the `-h` option to display the help.

   When you run `pmsrvconfig` with the `-i` (interactive) option, the configuration script gathers information from you by asking you a series of questions. During this interview, you are allowed to either accept a default setting or set an alternate setting.

   Once you have completed the policy server configuration script interview, it configures the policy server.

2. When you run `pmsrvconfig` for the first time, it asks you to read and accept the End User License Agreement (EULA).

3. Enter a password for the new `pmpolicy` service account and confirm it. This password is also called the "Join" password. You will use this password when you add secondary policy servers or join remote hosts to this policy group.

   The configuration process:

   - Creates the `/etc/opt/quest/qpm4u/pm.settings` file, which contains various parameters and settings
   - Installs service entries in the `/etc/services` file, which contains unique port numbers for `pmmasterd` and `pmlocald`
   - Generates a SSH key for log access
   - Generates the master policy, a profile-based policy
   - Creates the SVN database repository for the master policy
   - Checks out a production copy of the master policy
   - Performs a syntax check of the master policy
   - Starts the Privilege Manager for Unix service (`pmserviced`). See pmserviced on page 452 for details.
   - Reloads the `pmloadcheck` configuration. See pmloadcheck on page 417 for details.

pmpolicy server configuration settings

When you run `pmsrvconfig` with the `-i` (interactive) option, the configuration script gathers information from you by asking you a series of questions. During this interview, you are allowed to either accept a default setting or set an alternate setting.

The configuration script first asks you to read and accept the End User License Agreement (EULA). The second question asks if you want to configure the server as a sudo or a pmpolicy type server; the default is sudo. See Security policy types on page 57 for more information about policy types. Depending on which type of server you are configuring the interview asks different questions.

The following table lists the default and alternative configuration settings when configuring a pmpolicy server. See PM settings variables on page 286 for more information about the policy server configuration settings.

Table 4: pmpolicy server configuration settings

Configuration setting	Default	Alternate
Configure Privilege Manager for Unix Policy Mode		
Configure host as primary or secondary policy group server:	primary	Enter **secondary**, then supply the primary server host name.

Configuration setting	Default	Alternate
Set Policy Group Name:	**<FQDN name of policy server>**	Enter Policy Group Name of your choice.
Policy mode: See Security policy types on page 57 for more information about policy types. Sets `policymode` in `pm.settings`. (Policy "modes" are the same as policy "types" in the console.)	sudo	Enter **pmpolicy**
Configure Security Policy		
Initialize the security policy?	YES	Enter **No**
Configure Privilege Manager for Unix Daemon Settings		
Policy server command line options: Sets `pmmasterdopts` in `pm.settings`.	-ar	Enter: • **-a** to send job acceptance messages to syslog. • **-e <logfile>** to use the error log file identified by <logfile>. • **-r** to send job rejection messages to syslog. • **-s** to send error messages to syslog. none to assign no options. **-a**, **-r**, and **-s** override `syslog no` option; **-e <logfile>** overrides the `pmmasterdlog <logfile>` option.
Enable remote access functions? Sets `clients` in `pm.settings`.	NO Does not make system information on this host available to policy servers located on other hosts.	Enter **Yes** to allow remote policy servers to connect to this primary policy server for remote I/O logging, or to access functions in the policy file. Entering **Yes** allows you to list remote hosts.

Configuration setting	Default	Alternate
If **Yes**, list of remote hosts allowed to connect to this policy server?	NO	Enter **Yes**, then add remote hosts to list.
Configure host as a PM Agent?	NO	Enter **Yes**, then configure command line options.
If **Yes**, configure command line options for the agent daemon?	`pmlocaldopts` is not set	Enter: • **-s** to send error messages to syslog. • **-e <logfile>** to use the error log file identified by <**logfile**>. • **-m** to only accept connections from the policy server daemon on the specified host. (Use Multiple **-m** options to specify more than one host.) • **none** to assign no options. These command-line options override the `syslog` and `pmmasterdlog` options configured in the `pm.settings` file.
Configure `pmlocald` on this host?	NO	Enter **Yes**
Configure policy server host components to communicate with remote hosts through firewall?	NO	Enter **Yes**
Configure `pmtunneld` on this host?	NO	Enter **Yes**
Define host services? You must add service entries to either the `/etc/services` file or the NIS services map.	YES Adds services entries to the `/etc/services` file.	Enter **No**
Communications Settings for Privilege Manager for Unix		
Policy server daemon port number:	12345	Enter a port number for the policy server to communicate with agents

Configuration setting	Default	Alternate
Sets `masterport` in `pm.settings`.		and clients.
Specify a range of reserved port numbers for this host to connect to other defined Privilege Manager for Unix hosts across a firewall? Sets `setreserveportrange` in `pm.settings`.	NO	Enter **Yes**, then enter a value between 600 and 1023: 1. Minimum reserved port. (Default is 600.) 2. Maximum reserved port. (Default is 1023.)
Specify a range of non-reserved port numbers for this host to connect to other defined Privilege Manager for Unix hosts across a firewall? Sets `setnonreserveportrange` in `pm.settings`.	NO	Enter **Yes**, then enter a value between 1024 and 65535: 1. Minimum non-reserved port. (Default is 1024.) 2. Maximum non-reserved port. (Default is 31024.)
Allow short host names? Sets `shortnames` in `pm.settings`.	YES	Enter **No** to use fully-qualified host names instead.
Configure Kerberos on you Sets `kerberos` in `pm.settings`.r network?	NO	Enter **Yes**, then enter: 1. Policy server principal name. (Default is `host`.) 2. Local principal name. (Default is `host`.) 3. Directory for replay cache. (Default is `/var/tmp`.) 4. Path for the Kerberos configuration files [`krbconf` setting]. (Default is `/etc/opt/quest/vas/vas.conf`.) 5. Full pathname of the Kerberos keytab file [`keytab` setting].

Configuration setting	Default	Alternate
		(Default is `/etc/opt/quest/vas/host.keytab`.)
Encryption level: See Encryption on page 5 for details. Sets `encryption` in `pm.settings`.	AES	Enter one of these encryption options: • DES • TRIPLEDES • AES
Enable certificates? Sets `certificates` in `pm.settings`.	NO	Enter **Yes**, then answer: Generate a certificate on this host? (Default is NO.) Enter **Yes** and specify a **passphrase** for the certificate. Once configuration of this host is complete, swap and install keys for each host in your system that need to communicate with this host. See Swap and install keys on page 39 for details.
Activate the failover timeout?	YES	Enter **Yes**, then assign the failover timeout in seconds: (Default is 10.)
Failover timeout in seconds: Sets `failovertimeout` in `pm.settings`.	10	Enter timeout interval.
Configure Privilege Manager for Unix Logging Settings		
Send errors reported by the policy server and local daemons to syslog?	YES	Enter **No**
Policy server log location: Sets `pmmasterdlog` in `pm.settings`.	/var/log/pmmasterd.log	Enter a location.
Install Privilege Manager for Unix Licenses		
XML license file to	(use the freeware	Enter enter location of the `.xml`

Configuration setting	Default	Alternate
apply:	product license)	license file. Enter **Done** when finished.
Enter <**password**> This password is also called the "Join" password. You will use this password when you add secondary policy servers or join remote hosts to this policy group.		

You can find an installation log file at: `/opt/quest/qpm4u/install/pmsrvconfig_output_<Date>.log`

Verifying the primary policy server configuration

To verify the policy server configuration

1. From the command line of the primary policy server, run:

   ```
   # pmsrvinfo
   ```

   The `pmsrvinfo` command displays the current configuration settings. For example:

   ```
   Policy Server Configuration:
   ----------------------------
   Privilege Manager for Unix version                    : 6.0.0
   Listening port for pmmasterd daemon          : 12345
   Comms failover method                        : random
   Comms timeout(in seconds)                    : 10
   Policy type in use                           : pmpolicy
   Group ownership of logs                      : pmlog
   Group ownership of policy repository         : pmpolicy
   Policy server type                           : primary
   Primary policy server for this group         : <polsrv>.example.com
   Group name for this group                    : <polsrv>.example.com
   Location of the repository
     : file:////var/opt/quest/<polsrv>/.<polsrv>/.repository/pmpolicy_repos/trunk
   Hosts in the group                           : <polsrv>.example.com
   ```

   Note the entries for policy type (`pmpolicy`) and policy server type (`primary`). See Security policy types on page 57 for more information about security policy types.

Recompile the whatis database

If you are using the *whatis* database and you chose to install the man pages, you may wish to recompile the database to allow users to search the documentation using keywords.

Join hosts to policy group

Once you have installed and configured the primary policy server, you are ready to join it to a policy group. When you join a policy server to a policy group, it enables that host to validate security privileges against a single common policy file located on the primary policy server, instead of on the host.

For Unix agents (qpm-agent), you must "join" your policy servers to the policy group using the pmjoin command.

Joining PM Agent to a Privilege Manager for Unix policy server

To join a PM Agent to a policy server

1. Log on as the root user and change to the directory containing the qpm-agent package for your specific platform. For example, on a 64-bit Red HatLinux, enter:

   ```
   # cd agent/linux-x86_64
   ```

2. Run:

   ```
   # pmjoin <primary_policy_server>
   ```

   where <**primary_policy_server**> is the hostname of the primary policy server.

   Running pmjoin performs the configuration of the PM Agent, including modifying the pm.settings file The pmjoin command supports many command line options. See pmjoin on page 408 for details or run pmjoin with the -h option to display the help.

   - When you run pmjoin with no options, the configuration script automatically configures the agent with default settings. See Agent configuration settings on page 36 for details about the default and alternate agent configuration settings.

     You can modify the /etc/opt/quest/qpm4u/pm.settings file later, if you want to change one of the settings. See PM settings variables on page 286 for details.

   - When you run pmjoin with the -i (interactive) option, the configuration script gathers information from you by asking you a series of questions. During this interview, you are allowed to either accept a default setting or set an alternate setting.

Once you have completed the configuration script interview, it configures the agent and joins it to the policy server.

3. When you run `pmjoin` for the first time, it asks you to read and accept the End User License Agreement (EULA).

   Once you complete the agent configuration script (by running the `pmjoin` command), it:

   - Enables the `pmlocald` service
   - Updates the `pm.settings` file
   - Adds the Privilege Manager for Unix shells to the system's list of valid shells and creates wrappers for the installed (system) shells. The following shells are provided, based on standard shells:
     - `pmksh`, a Privilege Manager for Unix enabled version of the Korn shell
     - `pmsh`, a Privilege Manager for Unix enabled version of the Bourne shell
     - `pmcsh`, a Privilege Manager for Unix version of c shell
     - `pmbash`, a Privilege Manager for Unix version of the Bourne Again Shell

     Each shell provides command-control for every command entered by the user during a login session. You can configure each command the user enters to require authorization with the policy server for execution. This includes the shell built-in commands.
   - Updates `/etc/shells`
   - Reloads the `pmserviced` configuration
   - Checks the connection to the policy server host

4. To verify that the agent installation has been successful, as an unprivileged user, run a command that is permitted by the default Privilege Manager for Unix security policy, `demo.profile`. For example, the default security policy allows any user to run the `id` command as the `root` user:

   ```
   # pmrun id
   ```

   This returns the `root` user id, not the user's own id, to show that the command ran as `root`.

Agent configuration settings

The following table lists the `pmjoin` command options, the default settings, and alternatives. See PM settings variables on page 286 for more information about the policy server configuration settings.

Table 5: Agent configuration settings

Option	Default	Alternate setting
Enable agent daemon	none	Enter:

Option	Default	Alternate setting
command line options:		• **-e <logfile>** to use the error log file identified by <**logfile**>. • **-m** to only accept connections from the policy server daemon on the specified host. (Use multiple **-m** options to specify more than one host.) • **-s** to send error messages to syslog. none to assign no options. • These command-line options override the `syslog` and `pmlocaldlog` options configured in the `pm.settings` file.
Enable client daemon?	YES	Enter **No**
Configure host components to communicate with remote hosts through firewall?	NO	Enter **Yes**
Enable Privilege Manager for Unix shells (`pmksh`, `pmsh`, `pmcsh`, `pmbash`)?	YES That is, you want to use a Privilege Manager for Unix shell to control or log Privilege Manager for Unix sessions, regardless of how the user logs in (`telnet`, `ssh`, `rsh`, `rexec`).	Enter **No** if you do NOT want to add the Privilege Manager for Unix shells to the system. That is, you do not want to use the Privilege Manager for Unix shells as a login shell.
Add the entries to the `/etc/services` file?	YES	Enter **No** You must add service entries to either the `/etc/services` file or the NIS services map.
Edit list of policy servers with which this agent can communicate?	none	Enter valid policy server names to add to the list.
Indicate if the list is correct	YES	Enter **No**

Option	Default	Alternate setting
Policy Server daemon port #	12345	Enter a port number
Specify the agent daemon port number:	12346	Enter a port number for the agent to communicate with the policy server.
Specify a range of local port numbers for this host to connect to other defined Privilege Manager for Unix hosts across a firewall?	NO	Enter **Yes**, then enter: 1. Minimum reserved port (600-1024). (Default is 600.) 2. Maximum reserved port (600-1024). (Default is 1024.)
Allow short host names?	YES	Enter **No** to use fully qualified host names instead.
Configure Kerberos on your network?	NO	Enter **Yes**, then enter: 1. Policy server principal name. (Default is `host`.) 2. Local principal name. (Default is `host`.) 3. Directory for replay cache. (Default is `/var/tmp`. 4. Path for the Kerberos configuration files. (Default is `/etc/opt/quest/vas/vas.conf`.) 5. Full pathname of the Kerberos keytab file. (Default is `/etc/opt/quest/vas/host.keytab`.
Specify encryption level: See Encryption on page 5 for details.	AES	Enter one of these encryption options: • DES • TRIPLEDES • AES
Enable certificates?	NO	Enter **Yes**, then answer: Generate a certificate on this host? (Default is NO.) Enter **Yes** and specify a **passphrase** for the certificate.

Option	Default	Alternate setting
		Once configuration of this agent is complete, swap and install keys for each host in your system that need to communicate with this host. See Swap and install keys on page 39 for details.
Activate the failover timeout?	YES	Enter **No**, then assign the failover timeout in seconds. Default: 10 seconds
Assign the failover timeout	10	Enter a timeout value in seconds
Select random policy server	YES	Enter **No**
Send errors reported by agent to syslog?	YES	
Store errors reported by the agent daemon in `/var/log/pmlocald.log` ?	YES	Enter **No**, then enter a location.
Enter **No**, then enter a location.		

Swap and install keys

If certificates are enabled in the `/etc/opt/quest/qpm4u/pm.settings` file of the primary server, then you must exchange keys (swap certificates) prior to joining a client or secondary server to the primary server. Optionally, you can run the configuration or join with the `-i` option to interactively join and exchange keys.

One Identity recommends that you enable certificates for higher security.

The examples below use the keyfile paths that are created when using interactive configuration or join if certificates are enabled.

To swap certificate keys

1. Copy Host2's key to Host1. For example:

```
# scp /etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_localhost \
root@Host1:/etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_server2
```

2. Copy Host1's certificate to Host2. For example:

   ```
   # scp root@host1:/etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_localhost \
   /etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_host1
   ```

3. Install Host1's certificate on Host2. For example:

   ```
   # /opt/quest/sbin/pmkey -i /etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_host1
   ```

4. Log on to Host1 and install Host2's certificate. For example:

   ```
   # /opt/quest/sbin/pmkey -i /etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_host2
   ```

If you use the interactive configure or join, the script will exchange and install keyfiles automatically.

See Configuring certificates on page 145 for more information.

Configure a secondary policy server

The *primary* policy server is always the first server configured in the policy server group; *secondary* servers are subsequent policy servers set up in the policy server group to help with load balancing. The "master" copy of the policy is kept on the primary policy server.

All policy servers (primary and secondary) maintain a production copy of the security policy stored locally. The initial production copy is initialized by means of a `checkout` from the repository when you configure the policy server. Following this, the policy servers automatically retrieve updates as required.

By adding one or more secondary policy servers, the work of validating policy is balanced across all of the policy servers in the group, and provides failover in the event a policy server becomes unavailable. Use `pmsrvconfig` with the `-s` option to configure the policy server as a secondary server.

Installing secondary servers

To install the secondary server

1. From the command line of the host designated as your secondary policy server, log on as the `root` user.
2. Change to the directory containing the `qpm-server` package for your specific platform.

   For example, on a 64-bit Red Hat Linux, run:

```
# cd server/linux-x86_64
```

3. Run the platform-specific installer. For example, run:

   ```
   # rpm --install qpm-server-*.rpm
   ```

   The Solaris server has a filename that starts with `QSFTpmsrv`.

   When you install the `qpm-server` package, it installs all three Privilege Manager for Unix components on that host:

   - Privilege Manager for Unix Policy Server
   - PM Agent (which is used by Privilege Manager for Unix)
   - Sudo Plugin (which is used by Safeguard for Sudo)

   You can only join a PM Agent host to a Privilege Manager for Unix policy server or a Sudo Plugin host to a sudo policy server. See Security policy types on page 57 for more information about policy types.

Configuring a secondary server

You use the `pmsrvconfig -s <primary_policy_server>` command to configure a secondary server. See pmsrvconfig on page 458 for more information about the `pmsrvconfig` command options.

To configure the secondary server

1. From the command line of the secondary server host, run:

   ```
   # pmsrvconfig –s <primary_policy_server>
   ```

   where `<primary_policy_server>` is the hostname of your primary policy server.

   `pmsrvconfig` prompts you for the "Join" password from the primary policy server, exchanges ssh keys for the pmpolicy service user, and updates the new secondary policy server with a copy of the *master* (production) policy.

Once you have installed and configured a secondary server, you are ready to join the PM Agent to it. See Join hosts to policy group on page 35 for details.

Synchronizing policy servers within a group

Privilege Manager for Unix generates log files containing event timestamps based on the local clock of the authorizing policy server.

To synchronize all policy servers in the policy group, use Network Time Protocol (NTP) or a similar method of your choice.

Install PM Agent on a remote host

Once you have installed and configured the primary policy server, you are ready to install a PM Agent on a remote host.

Checking PM Agent host for installation readiness

To check a PM Agent host for installation readiness

1. Log on to the remote host system as the *root* user and navigate to the files you extracted on the primary policy server.
2. From the root directory, run a readiness check to verify the host meets the requirements for installing and using the PM Agent, by running:

   ```
   # sh preflight.sh --pmpolicy --policyserver <primary_policy_server>
   ```

   where `<primary_policy_server>` is the hostname of the primary policy server.

   Running `preflight.sh --pmpolicy` performs these tests:

   - Basic Network Conditions:
     - Hostname is configured
     - Hostname can be resolved
     - Reverse lookup returns it own IP
   - Privilege Manager for Unix Client Network Requirements
     - PM Agent port is available (TCP/IP port 12346)
     - Tunnel port is available (TCP/IP port 12347)
   - Policy Server Connectivity
     - Hostname of policy server can be resolved
     - Can ping the policy server
     - Can make a connection to policy server
     - Policy server is eligible for a join
     - Policy server can make a connection to the PM Agent on port 12346
3. Resolve any reported issues and rerun `pmpreflight` until all tests pass.

Installing a PM Agent on a remote host

To install an agent on a remote host

1. Log on as the root user.
2. Change to the directory containing the qpm-agent package for your specific platform. For example, on a 64-bit Red Hat Linux, enter:

   ```
   # cd agent/linux-x86_64
   ```

3. Run the platform-specific installer. For example, on Red Hat Linux run:

   ```
   # rpm --install qpm-agent-*.rpm
   ```

   Once you install the Privilege Manager for Unix agent package, the next task is to join the agent to the policy server.

Joining the PM Agent to the primary policy server

Once you have installed a Privilege Manager for Unix agent on a remote host you are ready to join it to the primary policy server.

To join a PM Agent to the primary policy server

1. From the command line of the remote host, run:

   ```
   # /opt/quest/sbin/pmjoin <primary_policy_server>.example.com
   ```

   where <primary_policy_server> is the name of the primary policy server host.

   If you are not running the pmjoin command on a policy server, it requires that you specify the name of a policy server within a policy group.

   The pmjoin command supports many command line options. See pmjoin on page 408 for details or run pmjoin with the -h option to display the help.

   - When you run pmjoin with no options, the configuration script automatically configures the agent with default settings. See Agent configuration settings on page 36 for details about the default and alternate agent configuration settings.

     You can modify the /etc/opt/quest/qpm4u/pm.settings file later, if you want to change one of the settings. See PM settings variables on page 286 for details.

- When you run pmjoin with the -i (interactive) option, the configuration script gathers information from you by asking you a series of questions. During this interview, you are allowed to either accept a default setting or set an alternate setting.

  Once you have completed the configuration script interview, it configures the agent and joins it to the policy server.

Running pmjoin performs the configuration of the Privilege Manager for Unix agent, including modifying the pm.settings file and starting up the pmserviced daemon.

2. When you run pmjoin for the first time, it asks you to read and accept the End User License Agreement (EULA).

   Once you complete the agent configuration script (by running the pmjoin command), it:

   - Enables the pmlocald service
   - Updates the pm.settings file
   - Creates wrappers for the installed shells
   - Updates /etc/shells
   - Reloads the pmserviced configuration
   - Checks the connection to the policy server host

3. To verify that the agent installation has been successful, run

   ```
   # pmclientinfo
   ```

   This returns displays configuration information about a client host. See pmclientinfo on page 402 for details.

Verifying PM Agent configuration

To verify the PM Agent configuration

1. From the command line, run:

   ```
   # pmclientinfo
   ```

   The pmclientinfo command displays the current configuration settings. For example:

   ```
   [0][root@host1 /]# pmclientinfo
      - Joined to a policy group                : YES
      - Name of policy group                    : polsrv1.example.com
      - Hostname of primary policy server       : polsrv1.example.com
      - Policy type configured on policy group  : pmpolicy
   [0][root@host1 /]#
   ```

The secondary server PM Agent will be joined to the secondary server. This is unique because all other PM Agent hosts must join to the primary server.

Load balancing on the client

Load balancing is handled on each client, using information that is returned from the policy server each time a session is established.

If a session cannot be established because the policy server is unavailable (or offline) that policy server is marked as *unavailable*, and no further `pmrun` sessions are sent to it until the next retry interval.

`pmloadcheck`runs transparently on each host to check the availability and loading of the policy server. When a policy server is marked as *unavailable*, `pmloadcheck` attempts to connect to it at intervals. If it succeeds, the policy server is marked as *available* and able to run Privilege Manager for Unix sessions.

To view the current status of the policy server

- Run the following command:

  ```
  # pmloadcheck [-f]
  ```

If the policy server cannot be contacted, the last known information for this host is reported.

Remove configurations

You can remove the Privilege Manager for Unix Server or PM Agent configurations by using the `-u` option with the following commands:

- `pmsrvconfig` to remove the Privilege Manager for Unix Server configuration
- `pmjoin` to remove the PM Agent configuration

Take care when you remove the configuration from a policy server, particularly if the policy server is a primary server with secondary policy servers in the policy group, as agents joined to the policy group will be affected.

Uninstalling the Privilege Manager for Unix software packages

To uninstall the Privilege Manager for Unix packages

1. Log in and open a root shell.
2. Use the package manager for your operating system to remove the packages:

Table 6: Privilege Manager for Unix Server uninstall commands

Package	Command
RPM	# rpm -e qpm-server
DEB	# dpkg -r qpm-server

Table 7: PM Agent uninstall commands

Package	Command
RPM	# rpm -e qpm-agent
DEB	# dpkg -r qpm-agent
Solaris	# pkgrm QSFTpmagt
HP-UX	# swremove qpm-agent
AIX	# installp -u qpm-agent

Upgrade Privilege Manager for Unix

Privilege Manager for Unix supports a direct upgrade installation from version 6.0. The Privilege Manager for Unix software in this release is provided using platform-specific installation packages.

If you are currently running Privilege Manager for Unix 6.0, it may be possible to perform a direct upgrade installation depending on the package management software on your platform (Note: Direct upgrade installations are not possible with Solaris`.pkg` packages). If you perform a direct upgrade installation, your previous configuration details are retained. Where a direct upgrade is not possible, you must first remove the previously installed package, and install and configure Privilege Manager for Unix as a new product installation.

Before you upgrade

Because the Privilege Manager for Unix 7.1 original platform installer packages do not provide an automated rollback script, One Identity highly recommends that you back up important data such as your license, `pm.settings` file, policy, and log files before you attempt to upgrade your existing Privilege Manager for Unix policy servers.

To install Privilege Manager for Unix 7.1, change to the directory where the install package is located for your platform and run the package installer. See Installing the Privilege Manager for Unix packages on page 27 for details about how to install the Privilege Manager for Unix software.

Upgrading Privilege Manager for Unix packages

Privilege Manager for Unix has the following three packages:

- Server (qpm-server)
- PM Agent (qpm-agent) - Used by Privilege Manager for Unix only
- Sudo Plugin (qpm-plugin) - Used by Safeguard for Sudo only

These packages are mutually exclusive, that is, you can only install one of these packages on a host at any given time.

For more information on installing/upgrading the Sudo Plugin, see the *One Identity Safeguard for Sudo Administration Guide*.

Upgrading the server package

To upgrade the server package

1. Change to the directory containing the `qpm-server` package for your specific platform. For example, on a 64-bit Red Hat Linux system, run:

   ```
   # cd server/linux-x86_64
   ```

2. Run the platform-specific installer. For example, run:

   ```
   # rpm --upgrade qpm-server-*.rpm
   ```

Upgrading the PM Agent package

To upgrade the PM Agent package

1. Change to the directory containing the `qpm-agent` package for your specific platform. For example, on a 64-bit Red Hat Linux 5 system, run:

   ```
   # cd agent/linux-x86_64
   ```

2. Run the platform-specific installer. For example, run:

   ```
   # rpm --upgrade qpm-agent*.rpm
   ```

Removing Privilege Manager for Unix packages

Removing the server package

To remove the server package

1. Run the package uninstall command for your operating system.

   For example, to remove the qpm-server package on a 64-bit Red Hat Enterprise Linux 5 system, run:

   ```
   # rpm --erase qpm-server
   ```

2. To complete the removal of the qpm-server package, delete:
   - pmpolicy service user
   - pmpolicy group
   - pmlog group
   - policy repository directories in /etc/opt/quest/qpm4u/

Removing the PM Agent package

To remove the agent package

1. Run the package uninstall command for your operating system.

   For example, to remove the qpm-agent package on a 64-bit Red Hat Enterprise Linux 5 system, run:

   ```
   # rpm --erase qpm-agent
   ```

System Administration

Privilege Manager for Unix provides command line utilities to help you manage your policy servers. They can be used to check the status of your policy servers, edit the policy, or to simply report the information.

Reporting basic policy server configuration information

To report basic information about the configuration of a policy server

1. From the command line, enter:

   ```
   # pmsrvinfo
   ```

   This command returns output similar to this:

   ```
   Policy Server Configuration:
   ----------------------------
   Privilege Manager for Unix version            : 7.1.0 (nnn)
   Listening port for pmmasterd daemon  : 12345
   Comms failover method                : random
   Comms timeout(in seconds)            : 10
   Policy type in use                   : pmpolicy
   Group ownership of logs              : pmlog
   Group ownership of policy repository : pmpolicy
   Policy server type                   : primary
   Primary policy server for this group : myhost.example.com
   Group name for this group            : MyPolicyGroup
   Location of the repository           : file:
                              ////var/opt/quest/qpm4u/.qpm4u/.repository/sudo_
   repos/trunk
   Hosts in the group                   : myhost.example.com
   ```

Checking the status of the master policy

The "master" copy of the policy file resides in a repository on the primary policy server. Each primary and secondary policy server maintains a "production" copy of the policy file or files. Use the `pmpolicy` utility to verify that the production copy is current with the master policy.

To compare the production policy file against the master policy on the primary server

1. From the command line, enter:

   ```
   # pmpolicy masterstatus
   ```

   If the files are in sync, the `Current Revision` number will match the `Latest Trunk Revision` number. If someone hand-edited the local copy without using `pmpolicy` utility commands to commit the changes, "Locally modified" will indicate "YES".

   If the production policy is not current with the master policy you can update the production policy with `pmpolicy sync`.

Related Topics

pmpolicy

Checking the policy server

When the policy server is not working as expected, use the `pmsrvcheck` command to determine the state of the server and its configuration.

To verify the policy server is running

1. From the command line, enter:

   ```
   # pmsrvcheck
   ```

   This command returns output similar to this:

   ```
   testing policy server [ Pass ]
   ```

   If the policy server is working properly, the output returns '`pass`', otherwise it returns, '`fail`'.

Related Topics

pmsrvcheck

Checking policy server status

The primary and secondary policy servers need to communicate with each other. Run the `pmloadcheck` command on a policy server host to verify that it can communicate with other policy servers in the policy group.

To determine if there any issues with policy servers in the policy group

From the Privilege Manager for Unix host command line, enter:

```
# pmloadcheck -r
```

This command has output similar to this:

```
[0][root@sol10-x86 /]# pmloadcheck -r
** Reporting current availability of each configured master...
   * Host:myhost1.example.com (172.16.1.129) ... [ OK ]
** Based on this data, the server list is currently ordered as:
1.     myhosts.example.com
```

Related Topics

pmloadcheck

Checking the PM Agent configuration status

To check the PM Agent configuration status

1. From the command line, enter:

   ```
   # pmclientinfo
   ```

   This command returns output similar to this:

   ```
   # pmclientinfo
      - Joined to a policy group                 : YES
      - Name of policy group                     : MyPolicyGroup
      - Hostname of primary policy server        : myhost.example.com
      - Policy type configured on policy group : pmpolicy
   ```

   If the PM Agent has been properly configured, it will say ‘Joined to a Policy Group: YES’ and give the policy group name and primary policy server’s hostname.

Related Topics

pmclientinfo

Installing licenses

To install a license file

1. Copy the `.dlv` license file to the policy server.
2. To install the license, run:

   ```
   # /opt/quest/sbin/pmlicense –l <license_file>
   ```

   This command displays your currently installed license and the details of the new license to be installed.
3. When it asks, "`Would you like to install the new license (Y/N) [Y]?`", press **Enter**, or type: **Y**
4. If there are other policy servers configured in your policy server group, it forwards the license configuration to the other servers.

Related Topics

pmlicense

Displaying license usage

Use the `pmlicense` command to display how many client licenses are installed on the policy server on which you run the command.

Use `pmlicense` without any arguments to show an overall status summary, including the number of licenses configured and the total licenses in use for each license option.

To display current license status information

1. At the command line, enter:

   ```
   # pmlicense
   ```

   Privilege Manager for Unix displays the current license information, noting the status of the license. Your output will be similar to the following:

```
*** One Identity Privilege Manager for Unix ***
*** QPM4U VERSION 7.1.0 (0xx) ***
*** CHECKING LICENSE ON HOSTNAME:user123.example.com, IP ADDRESS:10.10.178.123
***
*** SUMMARY OF ALL LICENSES CURRENTLY INSTALLED ***
   * License Type PERMANENT
   * Commercial/Freeware License COMMERCIAL
   * Expiration Date NEVER
   * Max QPM4U Client Licenses 10
   * Max Sudo Policy Plugin Licenses 0
   * Max Sudo Keystroke Plugin Licenses 0
   * Authorization Policy Type permitted ALL
   * Total QPM4U Client Licenses In Use 4
   * Total Sudo Policy Plugins Licenses In Use 0
   * Total Sudo Keystroke Plugins Licenses In Use 0
```

The above example shows that the current license allows for ten QPM4U clients (PM Agent licenses) and four licenses are currently in use.

Use `pmlicense` with the `–us` option to view a summary usage report; use `-uf` to view the full usage report.

To show a full usage report including last use dates

1. At the command line, enter:

   ```
   # pmlicense -uf
   ```

   Your output will be similar to the following:

   ```
   Detailed Licensed Hosts Report
   ----------------------------------------------------------------------------
   Number | Last Access Time                              | Hostname
   ----------------------------------------------------------------------------
          | QPM4U             | SudoPolicy | SudoKeystroke |
   ----------------------------------------------------------------------------
   1      | 2012/07/01 17:14 |            |               | admin1.example.com
   2      | 2012/07/01 17:14 |            |               | user101.example.com
   3      | 2012/07/01 16:28 |            |               | user123.example.com
   4      | 2012/07/01 17:14 |            |               | dev023.example.com
   ```

The above output shows the full report, including the host names and dates the Unix agents used the policy server.

The `pmlicense` command supports many other command-line options.

Related Topics

pmlicense

Listing policy file revisions

After you have made several revisions to your policy file under source control, you can view the list of policy file versions stored in the repository.

To display all previous version numbers with timestamps and commit logs

1. From the command line, enter:

   ```
   # pmpolicy log
   ```

   This command returns output similar to this:

   ```
   ** Validate options          [ OK ]
   ** Check out working copy    [ OK ]
   ** Retrieve revision details [ OK ]
   version="3",user="pmpolicy",date=2011-05-11,time=19:27:01,msg=""
   version="2",user="pmpolicy",date=2011-05-11,time=19:19:47,msg="added tuser"
   version="1",user="pmpolicy",date=2011-05-11,time=15:56:12,msg="First import"
   ```

Viewing differences between revisions

You can view the changes from revision to revision of a policy file.

To show the differences between version 1 and version 3

1. From the command line, enter:

   ```
   # pmpolicy diff –r:1:2
   ```

   This command returns output similar to this:

   ```
   ** Validate options                                            [ OK ]
   ** Check out working copy (trunk revision)                     [ OK ]
   ** Check differences                                           [ OK ]
   ** Report differences between selected revisions               [ OK ]
      - Differences were detected between the selected versions
   Details:
   Index: profiles/helpdesk.profile
   ===================================================================
   --- profiles/helpdesk.profile (revision 1)
   +++ profiles/helpdesk.profile (revision 2)
   @@ -18,6 +18,7 @@
   enableRemoteCmds = false;   # Should remote cmds be allowed for privilege cmds
   ```

```
?
                            # - ie should it allow cmds if: submithost !=
runhost
                            #
+shellProfile = "helpdesk";
authUser = "root";          # runuser to use when running the authCommands
                            # Set to 1 of the following:
```

The output reports lines removed and lines added in a unified diff format.

Backup and recovery

It is important for you to perform systematic backups of the following directories on all policy servers:

- `/var/opt/quest/qpm4u` which contains:
    - Event Logs
    - Keystroke Logs (I/O logs)
    - SVN Repository
    - SSH Keys
    - pmpolicy
- `/etc/opt/quest/qpm4u` which contains:
    - Settings File
    - Production Policy
- `/opt/quest/qpm4u/.license*` which contains:
    - License Files
- `/opt/quest/qpm4u/license*` which contains:
    - License Files
- `/opt/quest/qpm4u/install` which contains:
    - Install Logs
    - End User License Agreement (EULA)

When recovering from a failure, keep the same hostname and IP address.

Managing Security Policy

The Privilege Manager for Unix security system consists of one or more centralized policy servers and one or more remote clients. A user wishing to run a command secured by Privilege Manager for Unix makes a request to their client. The request is then propagated to the policy server which consults a security policy to determine whether to allow or disallow the command. A typical Privilege Manager for Unix installation has several policy servers to provide adequate fail-over and load-balancing coverage.

The Privilege Manager for Unix policy servers are capable of recording all the activity which passes through them. The power to accurately log `root`, and other account activities in a safe environment allows you to implement a secure system administration regime with an indelible audit trail. You always know exactly what is happening in `root`, as well as who did it, when it happened, and where.

The data created by the Privilege Manager for Unix policy servers is stored in a log file called an event log. An entry in the event log is made every time a policy server is used to run a command.

Security policy types

The security policy lies at the heart of Privilege Manager for Unix. Privilege Manager for Unix guards access to privileged functions on your systems according to rules specified in the security policy. It stipulates which users may access which commands with escalated privileges.

Privilege Manager for Unix supports two security policy types (or modes):

- **sudo policy type**: Safeguard for Sudo uses a standard sudoers file as its security policy; that is, the sudo policy is defined by the `sudoers` file which contains a list of rules that control the behavior of sudo. The `sudo` command allows users to get elevated access to commands even if they do not have `root` access.

  Safeguard uses the sudo policy type by default. The sudo policy type is only supported with the One Identity Safeguard for Sudo product.
- **pmpolicy type**: Privilege Manager for Unix uses an advanced security policy which employs a high-level scripting language to specify access to commands based on a wide variety of constraints. The Privilege Manager for Unix policy is defined in

`pm.conf`, the default policy configuration file which contains statements and declarations in a language specifically designed to express policies concerning the use of `root` and other controlled accounts.

Beginning with release 7.0, both Privilege Manager for Unix and Safeguard for Sudo support the pmpolicy type.

Management Console for Unix gives you the ability to centrally manage policy located on the primary policy server. You view and edit both pmpolicy and sudo policy from the **Policy** tab on the mangement console.

By default, the policy server configuration tool (`pmsrvconfig`) uses the sudo policy type on new installations; if you want to run Privilege Manager for Unix using the pmpolicy type you must specify that explicitly when using the policy server configuration script.

The `pmsrvconfig` program is used by both Privilege Manager for Unix and Safeguard for Sudo. Run `pmsrvconfig -m sudo` or `pmsrvconfig -m pmpolicy` to specify the policy type. See pmsrvconfig on page 458 for more information about the `pmsrvconfig` command options.

When you join a Sudo Plugin to a policy server, Privilege Manager for Unix adds the following lines to the current local sudoers file, generally found in `/etc/sudoers`.

```
##
 ## WARNING: Sudoers rules are being managed by Safeguard for Sudo
 ## WARNING: Do not edit this file, it is no longer used.
 ##
 ## Run "/opt/quest/sbin/pmpolicy edit" to edit the actual sudoers rules.
 ##
```

When you unjoin the Sudo Plugin, Privilege Manager for Unix removes those lines from the local sudoers file.

If you configure Privilege Manager for Unix using the pmpolicy type, `pmsrvconfig` creates a profile-based policy. This security policy simplifies setup and maintenance through use of easy-to-manage profile templates. See pmpolicy type policy on page 59 for more information about profile-based policy.

Use the `pmsrvconfig -f <path>` command to override the default and import the initial security policy from the specified location.

Privilege Manager for Unix uses a version control system to manage and maintain the security policy. This allows auditors and system administrators to track changes that have been made to the policy and also allows a single policy to be shared and distributed among several policy servers. The "master" copy of the security policy and all version information is kept in a repository on the primary policy server.

You manage the security policy using the `pmpolicy` command and a number of `pmpolicy` subcommands. It is important that you only make changes to the policy using the `pmpolicy` command. Using `pmpolicy` ensures that the policy is updated in the repository and across all policy servers in the policy group. You can run the `pmpolicy` command from any policy server in the policy group.

Do not edit the security policy on a policy server directly. Changes made using `vi` will eventually be overwritten by the version control system.

The primary policy server uses a local service account, `pmpolicy`, to own and manage the security policy repository. The `pmpolicy` service account is set when you configure the primary policy server. At that time you assign the `pmpolicy` service account a password and set its home directory to `/var/opt/quest/qpm4u/pmpolicy`. This password is also called the "Join" password because you use it when you add secondary policy servers or join remote hosts to this policy group.

You can manually create the `pmpolicy` user prior to running the `pmsrvconfig` script, but if the user account does not exist, the script creates the user and asks you for a password.

When you run the `pmsrvconfig` command, it attempts to initialize the security policy by reusing an existing policy file on this host. If a security policy does not exist, it generates a default policy.

Specifying security policy type

To configure a Privilege Manager for Unix policy server, you must specify the pmpolicy type.

To specify the security policy type

1. To specify the pmpolicy type, run:

   ```
   # pmsrvconfig -m pmpolicy
   ```

   For more information about pmpolicy language, see *Privilege Manager for Unix Administration Guide*.

Related Topics

pmsrvconfig

pmpolicy type policy

The Privilege Manager for Unix product uses a specialized policy (pmpolicy type policy), which allows for a more advanced security policy than is possible with the sudo policy type. The pmpolicy type uses a powerful scripting language to evaluate whether `pmmasterd` should allow requests based on a wide variety of criteria of what, where, when, and how users should be permitted to perform various privileged account actions.

By default, the main pmpolicy file is located in `/etc/opt/quest/qpm4u/policy/pm.conf`, but is not meant to be accessed directly.

pmpolicy type policy code looks like this:

```
if (user == "root" || "wheel" in getgroups(user)) {
     runuser = requestuser;
     accept;
 }
```

The above pmpolicy type code segment accepts requests from `root` or any user in the *wheel* group to run any command as any user.

Modifying complex policies

If your policy consists of several files (the default pmpolicy, for example) or if you want to add files to or remove files from your policy, use a checkout, change, and commit method for implementing the changes. The `pmpolicy checkout` command creates a working copy of the policy where you can make any necessary changes and then use the `pmpolicy commit` command to apply the changes back to the repository. You can use the `pmpolicy add` and `pmpolicy remove` commands to add or delete files to your working copy, respectively.

Checkout, change, and commit example

The following example modifies the default pmpolicy type profile. For example, say you wanted to create a new backup profile to allow backup operators to run the `dump` and `restore` commands. Use one of the existing profiles, `helpdesk.profile`, as a template. First, checkout a working copy to a temporary directory, like this:

```
# pmpolicy checkout -d /tmp
** Checkout to /tmp/policy_pmpolicy
** Create directory                                                  [ OK ]
** Check out working copy                                            [ OK ]
** Copy files                                                        [ OK ]
** Perform syntax check                                              [ OK ]
```

As seen in the command output, the working copy is placed in `/tmp/policy_pmpolicy`.

Next, change to the profiles directory within the working copy, copy `helpdesk.profile` to `backup.profile`, and run `pmpolicy add` to record that a file has been added to the working copy of the policy, as follows:

```
# cd /tmp/policy_pmpolicy/profiles
# cp -p helpdesk.profile backup.profile
# pmpolicy add -p profiles/backup.profile -d /tmp
   ** Validate options                                               [ OK ]
   ** Add file: profiles/backup.profile
```

```
    ** Validate arguments                                           [ OK ]
    ** Check if directory contains a working copy                   [ OK ]
       - Directory contains an svn working copy:/tmp/policy_pmpolicy
    ** Check current status of working copy                         [ OK ]
    ** Check working copy is up to date                             [ OK ]
    ** Check file status                                            [ OK ]
    ** Add entry:/tmp/policy_pmpolicy//profiles/backup.profile      [ OK ]
```

After editing `backup.profile` to make the necessary changes, use the `pmpolicy commit` command to apply the changes to the repository, as follows:

```
# pmpolicy commit -d /tmp -l "added backup.profile"
** Validate options                                                 [ OK ]
** Commit copy in directory:/tmp/policy_pmpolicy
** Check directory                                                  [ OK ]
** Perform syntax check                                             [ OK ]
** Verify files to commit                                           [ OK ]
** Commit change from working copy                                  [ OK ]
** Committed revision 3
```

Related Topics

pmpolicy type policy

Viewing the security profile changes

To view a summary of the changes you made to your security policy

1. At the command line, run:

```
# pmpolicy log
```

```
** Validate options          [ OK ]
** Check out working copy    [ OK ]
** Retrieve revision details [ OK ]
version="3",user="pmpolicy",date=2012-07-11,time=15:43:30,msg="add
helpdesk.shellprofile "
version="2",user="pmpolicy",date=2012-07-11,time=15:38:21,msg="add
shellProfile to helpdesk "
version="1",user="pmpolicy",date=2012-07-11,time=15:35:19,msg="First import"
```

2. To examine the differences between two versions, run:

```
# pmpolicy diff -r1:2
```

```
** Validate options                                              [ OK ]
** Check out working copy (trunk revision)                       [ OK ]
** Check differences                                             [ OK ]
** Report differences between selected revisions                 [ OK ]
   - Differences were detected between the selected versions
Details:
Index: profiles/helpdesk.profile
===================================================================
--- profiles/helpdesk.profile (revision 1)
+++ profiles/helpdesk.profile (revision 2)
@@ -18,6 +18,7 @@
enableRemoteCmds = false;   # Should remote cmds be allowed for privilege cmds
?
                            # - ie should it allow cmds if: submithost !=
runhost
                            #
+shellProfile = "helpdesk";
authUser = "root";          # runuser to use when running the authCommands
                            # Set to 1 of the following:
```

The output shows the `helpdesk.profile` file from line 18. The line that was added in the change between version 1 and version 2 is marked with a preceding "+".

8

The Privilege Manager for Unix Security Policy

Privilege Manager for Unix uses a feature full, high-level scripting language as its security policy. This is also known as the pmpolicy or legacy type security policy. As an alternative to learning the policy scripting language and developing a security policy from scratch, the default configuration installs a "ready to use" profile-based security policy and a number of pre-defined profiles.

This section examines the profile-based policy and provides specific examples of how to modify the profiles and add custom code to adapt the policy to your needs.

Default profile-based policy (pmpolicy)

The default configuration for the pmpolicy type is a profile-based security policy, which consists of several files. The main policy code resides in the `global_profile.conf` and `profileBasedPolicy.conf` files. One Identity recommends that you do not enter customized code in these files because it will impact the effectiveness and accuracy of the reports produced by Management Console for Unix. Instead, One Identity recommends that you use the profiles to affect changes in policy.

Best practice suggestion: Create custom code in `profile_customer_policy.conf`.

Related Topics

Policy scripting tutorial

Policy profiles

If you configure Privilege Manager for Unix using the pmpolicy type, `pmsrvconfig` creates a group of default profile-based policy files that you can customize to define which commands you want to allow your users to run. This provides a convenient way to experience the benefits of Privilege Manager for Unix while familiarizing yourself with the basics of policy scripting. The default security policy is made up of four sample profiles (`admin`, `demo`, `helpdesk`, `webadmin`) and three shell profiles (`root`, `restricted`, `qpm4u_login`).

Profiles

These profiles are enabled by default:

- `admin.profile` allows its members to run any command as the `root` user with full keystroke logging. You can add users to this profile by adding either their user ID or primary group ID to the `pf_authusers` or `pf_authgroups` variables, respectively. By default, the only member is the `root` user.
- `demo.profile` allows its members to run the `id` command as the `root` user to demonstrate how rights are delegated to non-privileged users. By default, all users are members of this profile.

These profiles are disabled by default:

- `helpdesk.profile` allows simple helpdesk functions.
- `webadmin.profile` allows for web server administration commands.

These profiles provide additional examples of how to create and configure profiles. They are disabled by default to prevent the granting of unwanted access.

Shell profiles

In addition, available shell profiles are also included in the `/profiles/shellprofiles` directory that permit the users to run specified shell programs.

These shell profiles are enabled by default:

- `root.shellprofile` allows the `root` user unrestricted access to any of the pmshells (`pmksh`, `pmcsh`, `pmsh`, and `pmbash`) as the `root` user.
- `qpm4u_login.shellprofile` allows any user unrestricted access to any of the `pmshellwrapper` wrapped shells that are configured on your system. See Privilege Manager for Unix shell features on page 117.

This shell profile is disabled by default:

- `restricted.shellprofile` allows any user to restrict access to any of the pmshells (`pmksh`, `pmcsh`, `pmsh`, and `pmbash`) as the `root` user with access to programs in `/opt/quest/bin` and `/sbin` only.

Profile-based policy files

The profiles and shell profiles allow for easy management of your policy, but the core of the policy is included in other policy files. The following table briefly describes the files that are used in the profile-based policy.

Table 8: Profile-based policy files

File	Description
pm.conf	Main policy file. includes: `global_profile.conf`, `profileBasedPolicy.conf` included by: NONE Do not put custom code in this policy file.
global_profile.conf	Defines default global variables. Also includes extensive comments documenting the variables. includes: NONE included by: `pm.conf` Do not put custom code in this policy file; however, you may change the default settings.
profileBasedPolicy.conf	Primary decision making policy file for the profile-based policy. (Not meant to be edited by customers.) includes: `profile_customer_policy.conf`, `*.profile`, `*.shellprofile` included by: `pm.conf` Special hook functions defined in `profile_customer_policy.conf` are called from this policy file.
profile_customer_ policy.conf	Custom policy file for customer-defined global variables and policy code. You can modify special hook functions to run custom policy code at certain points in the profile evaluation: • `fn_log_and_accept_custom` • `fn_custom_profile_init` • `pr_custom_profile_reset fn_customer_init` includes: NONE included by: `profileBasedPolicy.conf` You can create custom policies in this file. However, custom policies may affect the accuracy of the reports generated in Management Console for Unix. See The Privilege Manager for Unix Security Policy on page 63.
*.profile in profiles directory	Profile configuration file for allowing certain commands to be run by `pmrun`. includes: NONE included by: `profileBasedPolicy.conf` Do not put custom code in this policy file.

File	Description
*.shellprofile in profiles directory	Profile configuration file for interactive Privilege Manager for Unix shells (including wrapped shells). includes: NONE included by: `profileBasedPolicy.conf`

Profiles and shell profiles only contain variable assignments that are used in the policy decision making.

Profile selection

When evaluating the profile-based policy, the policy server must first determine which of the profiles match the incoming request. The policy uses the *Who*, *What*, *Where*, and *When* criteria specified in the profiles to determine a match. Note that the filename used for the profile is significant. The policy checks each of the profiles sequentially, in lexical order until a match is found. Once the a profile is selected, the remaining profiles are not evaluated.

Profile variables

Privilege Manager for Unix profiles (or roles) define who, what, where, when, and how users are permitted to perform various privileged account actions using variable values in the policy configuration profiles.

Management Console for Unix gives you the ability to centrally manage policy within a graphical user interface. You may view and edit both pmpolicy and sudo policy from the **Policy** tab on the mangement console.

The following tables list the predefined variables used in profile-based policy.

Table 9: General variables

Profile variable	Value type	Explanation
General		
pf_profile	String	The profile name. This variable is set by the `profileBasedPolicy.conf` file to be the base filename of the profile, minus the .profile or `.shellprofile` extension.
pf_profiledescription	String	A description of the profile. EXAMPLE:

Profile variable	Value type	Explanation
		`pf_profiledescription = "This is a description of this profile."`
pf_enableprofile	Boolean	Set to **true** to enable the profile. EXAMPLE: `pf_enableprofile = true;`
pf_tracelevel	Number	Enables tracing/debugging output at different levels: • 1: show reason for reject • 2: verbose output • 3: show debug trace EXAMPLE: `pf_tracelevel = 1;`
pf_enablekey-strokelogging	Boolean	Set to **true** to enable keystroke logging. EXAMPLE: `pf_enablekeystrokelogging = true;`
pf_iologdir	String	The directory in which to store I/O logs. A unique file is created in this directory for each keystroke logging session. EXAMPLE: `pf_iologdir = "/var/opt/quest/qpm4u/iolog/";`
pf_logpasswords	Boolean	Set to **false** to avoid writing passwords to the keystroke log. The password detection is determined by the `pf_passprompts` list. EXAMPLE: `pf_logpasswords = false;`
pf_passprompts	List	A list of strings interpreted as password prompts in stdout. EXAMPLE: `pf_passprompts = {"[pP]assword[ :]*"};`
Authentication		
pf_enableau-thentication	Boolean	Set to **true** to enable PAM authentication. By default, the submit user is authenticated on the master host using the `sshd` service. EXAMPLE:

Profile variable	Value type	Explanation
		`pf_enableauthentication = true;`
pf_authen-ticateonclient	Boolean	Set to **true** to require authentication on the client. If set to **false**, users are authenticated on the server, not on the client. EXAMPLE: `pf_authenticateonclient = true;` Authentication is only required if `pf_enableauthentication = true`.
pf_pamservice	String	Identifies the PAM service to use when authenticating to PAM. EXAMPLE: `pf_pamservice = "sshd";`
pf_pam_prompt	String	Configures the prompt to use with PAM. EXAMPLE: `pf_pam_prompt = "Password: ";`
pf_allowscp		Set to **true** to allow `scp` and non-interactive `SSH` commands when authentication for the shell is enabled. Only applies to `pmksh`, `pmcsh`, `pmsh`, `pmbash`, and `pmshellwrapper`. EXAMPLE: `pf_allowscp = false;`

Table 10: What settings

Profile variable	Value type	Explanation
Commands		
pf_authpaths	List	Specifies the paths from which commands are permitted to run. Empty lists are ignored. If not empty, this variable is passed to the agent for authorization at the point where the command is about to be run; the agent will then reject a command unless it is run from one of these paths. For a shell profile, this restriction is applied to the shell program itself, and to commands run from within the shell.

Profile variable	Value type	Explanation

		EXAMPLE:
		`pf_authpaths = {`
		`   # no path restrictions`
		`};`

pf_authcmds	List	Commands authorized to run; commands not in the list are rejected.
		Considerations:
		• If you specify a fully qualified path in `pf_authcmds`, you must specify the fully qualified path in the requested command.
		• glob is used to match the path, so be careful when using wild cards in the path.
		• You can precede an entry with an optional NOEXEC flag to ensure that the run command is blocked from forking any child processes. Put the flag at the beginning of the string and enclose the flag with '[]'.
		EXAMPLE:
		`pf_authcmds = {`
		`"/usr/bin/id *"`
		`};`

pf_enableremotecmds	Boolean	Set to **true** to allow commands to run on a different host when running `pmrun` with the `-h` option . EXAMPLE: `pf_enableremotecmds = false;`
Shell commands		
pf_shellcommandsaccept	List	Specifies the list of commands accepted by `pmmasterd` when run from within the shell. `pmmasterd` authorizes listed commands and they produce an event in the audit log. If `pf_shellcommandsaccept` is not empty, any matching command is accepted; all others are rejected. Considerations: • Only configure `pf_shellcommandsreject` or `pf_shellcommandsaccept`. • If both lists are empty, then all commands are

Profile variable	Value type	Explanation

		accepted.
		• Only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`.
		EXAMPLE:
		`pf_shellcommandsaccept = {`
		`};`

pf_shellcom-mandsreject	List	Specifies the list of commands rejected by `pmmasterd` when run from within the shell. `pmmasterd` authorizes listed commands and they produce an event in the audit log. If `pf_shellcommandsreject` is not empty, any matching command is be rejected; all others are accepted.
		Considerations:
		• Only configure `pf_shellcommandsreject` or `pf_shellcommandsaccept`.
		• If both lists are empty, then all commands are accepted.
		• Only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`.
		EXAMPLE:
		`pf_shellcommandsreject = {`
		`};`

pf_checkbuiltins	Boolean	Set to **true** to use shell builtins just like commands. This only applies to `pmksh`, `pmcsh`, and `pmsh`. EXAMPLE: `pf_checkbuiltins = true;`
pf_shellreject	String	Message to display when a user attempts to run a forbidden command. This only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`. EXAMPLE: `pf_shellreject = "You are not permitted to run this command";`
Pre-authorized Commands		
pf_shellallow	List	Defines the list of pre-authorized commands allowed by the shell without further authorization by the master. The shell interprets each item in this list as a regular

Profile variable	Value type	Explanation

		expression. Listed commands do not result in an audit event in the event log.
		
		This only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`.
		
		EXAMPLE:
		
		`pf_shellallow = {`
		
		`"(^|/)(exit|pwd|echo)$",`
		
		`};`

pf_shellallowpipe	List	Defines the list of pre-authorized commands allowed by the shell without further authorization by the master, but only in the case where std input is from a pipe (for example, ls | more). The shell interprets each item in this list as a regular expression. Listed commands do not result in an audit event in the eventlog.
		
		This only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`.
		
		EXAMPLE:
		
		`pf_shellallowpipe = {`
		
		`"(^|/)(awk|more|grep)$",`
		
		`   # allow pipe to innocuous common commands`
		
		`};`

pf_shell_forbid	List	Defines the list of commands rejected by the shell without further authorization by the master. The shell interprets each item in this list as a regular expression. Listed commands do not result in an audit event in the event log.
		
		This only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`.
		
		EXAMPLE:
		
		`pf_shellforbid = {`
		
		`"(^|/)(passwd|kill|shutdown)$",`
		
		`   # forbid sensitive commands`
		
		`"(^|/)(a|b|c|k|z)?sh$",`
		
		`   # forbid normal shells`
		
		`"(^|/)(bash|tcsh)$",`
		
		`   # forbid normal shells`
		
		`"(^|/)nc$",`

Profile variable	Value type	Explanation

		`# forbid cmds that allow remote execution`
		`};`

Table 11: Where settings

Profile Variable	Value Type	Explanation
Run Hosts		

pf_authrunhosts	List	Hosts where commands can run. If not empty, you can submit commands from any host in this list.
		EXAMPLES:
		`pf_authsubmithosts = {"host1"};`
		`   # allow cmds from host host1 only`
		`pf_authsubmithosts = {"*.one.two"};`
		`   # allow cmds from *.one.two only`
		`pf_authsubmithosts = {ALL};`
		`   # allow cmds from all hosts`
		`SAFEHOSTS = {"*.one.two"};`
		`pf_authsubmithosts = {SAFEHOSTS};`
		`   # allow cmds from *.one.two only`

pf_authrun-hostsad	List	Active Directory host groups where commands can run. If not empty, you can submit commands from any host in this list. You can specify an Active Directory domain name as part of the arguments; for example, as `<domain>/<name>`, `<domain>\\<name>`, or `<name>`. If a domain is not specified, then it uses the default joined domain.
		These lists do not support wild cards.
		EXAMPLES:
		`pf_authrunhostsad = { "TESTDOM1/testhosts1", "TESTDOM2/dbhosts1"};`
		`   # match any member of either AD group`
		`pf_authrunhostsad = { "testhosts1" };`
		`   # match members in the specified AD group in the default joined domain`
		`pf_authrunhostsad = {`
		`};`

Profile Variable	Value Type	Explanation
		`# match no AD groups`
Submit Hosts		

pf_authsub-mithosts	List	Hosts where commands can be submitted. If not empty, you can submit commands from any host in this list.
		EXAMPLES:
		`pf_authsubmithosts = {"host1"};`
		`   # allow cmds from host host1 only`
		`pf_authsubmithosts = {"*.one.two"};`
		`   # allow cmds from *.one.two only`
		`pf_authsubmithosts = {ALL};`
		`   # allow cmds from all hosts`
		`SAFEHOSTS = {"*.one.two"};`
		`pf_authsubmithosts = {SAFEHOSTS};`
		`   # allow cmds from *.one.two only`

pf_authsub-mithostsad	List	Active Directory host groups where commands can be submitted. If not empty, you can submit commands from any host in this list. You can specify a domain name as part of the arguments; for example, as `<domain>/<name>`, `<domain>\\<name>`, or `<name>`. If a domain is not specified, then it uses the default joined domain.
		These lists do not support wild cards.
		EXAMPLES:
		`pf_authsubmithostsad = { "TESTDOM1/testhosts1", "TESTDOM2/dbhosts1"};`
		`   # match any member of either AD group`
		`pf_authsubmithostsad = { "testhosts1" };`
		`   # match members in the specified AD group`
		`   # in the default joined domain`
		`pf_authsubmithostsad = {`
		`};`
		`   # match no AD groups`

Forbidden Run Hosts		
pf_forbidrunhosts	List	Hosts where members are forbidden to run commands. If

Profile Variable	Value Type	Explanation

		not empty, you can submit commands from any host NOT in this list.
		EXAMPLES:
		`pf_forbidrunhosts = {"fred"}; i`
		`   # allow cmds to all hosts except fred`
		`pf_forbidrunhosts = {"*.one.two"};`
		`   # allow cmds to all hosts except *.one.two`
		`BADHOSTS = {"*.one.two"};`
		`pf_forbidrunhosts = {BADHOSTS};`
		`   # allow cmds to all hosts except *.one.two`

pf_forbidrun-hostsad	List	Active Directory host groups where members are forbidden to run commands. If not empty, you can submit commands from any host NOT in this list. You can specify a domain name as part of the arguments; for example, as `<domain>/<name>`, `<domain>\\<name>`, or `<name>`. If a domain is not specified, then it uses the default joined domain.
		These lists do not support wild cards.
		EXAMPLES:
		`pf_forbidrunhostsad = { "TESTDOM1/testhosts1", "TESTDOM2/dbhosts1"};`
		`   # match any member of either AD group`
		`pf_forbidrunhostsad = { "testhosts1" };`
		`   # match members in the specified AD group in the default joined domain`
		`pf_forbidrunhostsad = { };`
		`   # match no AD groups`

Forbidden Submit Hosts		

pf_forbid-submithosts	List	Hosts where members are forbidden to submit commands. If not empty, you can submit commands from any host NOT in this list.
		EXAMPLES:
		`pf_forbidsubmithosts = {"host1"};`
		`   # allow cmds from all hosts except host1`
		`pf_forbidsubmithosts = {"*.one.two"};`

Profile Variable	Value Type	Explanation

		`# allow cmds from all hosts except *.one.two`
		`BADHOSTS = {"*.one.two"};`
		`pf_forbidsubmithosts = {BADHOSTS};`
		`# allow cmds from all hosts except *.one.two`

pf_ forbidrunhostsad	List	Active Directory host groups where members are forbidden to submit commands. If not empty, you can submit commands from any host NOT in this list. You can specify a domain name as part of the arguments; for example, as `<domain>/<name>`, `<domain>\\<name>`, or `<name>`. If a domain is not specified, then it uses the default joined domain.
		These lists do not support wild cards.
		EXAMPLES:
		`pf_forbidrunhostsad = { "TESTDOM1/testhosts1", "TESTDOM2/dbhosts1"};`
		`# match any member of either AD group`
		`pf_forbidrunhostsad = { "testhosts1" };`
		`# match members in the specified AD group in the default joined domain`
		`pf_forbidrunhostsad = { };`
		`# match no AD groups`

If a member (a user for the *group* lists, or a host for the *hosts* lists) is found in both forbid and auth lists, the request is rejected; the forbid list takes precedence.

Table 12: Who settings

Profile variable	Value type	Explanation
Users		

pf_authusers	List	Identifies the list of users that match this profile.
		EXAMPLES:
		`pf_authusers = {`
		`};`
		`# No users assigned to this profile`
		`pf_authusers = { "jsmith", "dbrown"};`
		`# match either user`

Profile variable	Value type	Explanation

		`pf_authusers = { ALL};`
		`   # match all users`
		`DBUSERS={"TESTDOM1/fred", "TESTDOM2/john"};`
		`   # allow cmds from /bin,/usr/bin,/tmp`
		`pf_authusers = { "jsmith*", DBUSERS};`
		`   # match fred, john & jsmith*`

pf_authuser	String	Identifies the `runas` user.
		EXAMPLE:
		`pf_authuser = user;`
		The `runas` user can be:
		• Any valid user name on the agent, such as:
		`pf_authuser = "fred";`
		`   # run command as fred`
		• A user variable or empty string ("") to run the command as the submit user; that is, set `runuser=user` (the default)
		`pf_authuser = user;`
		`   # run command as submit user`
		`pf_authuser = "";`
		`   # run command as submit user`
		• The `requestuser` variable to run the command as the user selected using the `pmrun -u user` option.
		`pf_authuser = requestuser;`
		`   # run command as the requested user`

Groups		
pf_authgroups	List	You can assign users to this profile by group membership on the client or server host, or by assigning individual user names. By default the group membership is verified against the submit user's group information passed on from the client host by `pmrun`. You can configure it to verify the group membership on the master host instead, using the `pf_useservergroupinfo` variable. EXAMPLES: `pf_authgroups = { "admins", "dbas"};`

Profile variable	Value type	Explanation

		`# match any member of either group`
		`pf_authgroups = { ALL};`
		`# match all groups`
		`DBGROUPS = {"db*"};`
		`pf_authgroups = { DBGROUPS, "root"};`
		`# match all db* groups and root`

pf_authgroup	String	If accepted, the request runs with the specified group as the `rungroup`.
		EXAMPLE:
		`pf_authgroup = use_rungroup;`
		The `rungroup` can be:
		• Any valid group name on the agent, such as:
		`pf_authgroup ="fred";`
		`# run command as group fred`
		• A group variable or empty string ("") to run the command as the submit group; that is, set rungroup=group (the default)
		`pf_authgroup = group;`
		`# run command as submit group`
		`pf_authgroup = "";`
		`# run command as submit group`
		• The `use_rungroup` constant to defer setting the rungroup to `pmlocald`; `pmlocald` will obtain the runuser's primary group and use that.
		`pf_authgroup = use_rungroup;`
		`# run command as runuser's group on the agent`

pf_useserver-groupinfo	Boolean	Set to **true** to check that the user is a member of one of the `pf_authgroups` on the master host, otherwise check the user's group membership on the client host. EXAMPLE: `pf_useservergroupinfo = false;`
AD Groups		
pf_authgroupsad	List	Identifies the list of non Unix-enabled AD groups that

Profile variable	Value type	Explanation

		match this profile. Use the format: `<domain>/<name>`, `<domain>\\<name>`, or `<name>`. If you do not specify the domain, it uses the default joined domain.
		This list does not support wild cards.
		EXAMPLES:
		`pf_authgroupsad = { "TESTDOM1/testgroup1", "TESTDOM2/dbgroup1"};`
		`# match any member of either AD group`
		`pf_authgroupsad = { };`
		`# match no AD groups`

Table 13: When settings

Profile Variable	Value Type	Explanation
Time Restrictions		
pf_enable-timerestrictions	Boolean	Set to **true** to enforce the time restrictions in the `restrictionHours` list. EXAMPLE: `pf_enabletimerestrictions = true;`

pf_restrictionhours	List	Start and End time of allowed time period. Set to "*" or empty string to disable time restrictions. Use 24-hour format, with no leading zero.
		EXAMPLES:
		`pf_restrictionhours = {"8:00", "18:00};`
		`# 8am - 8pm`
		`pf_restrictionhours = {"22:00", "07:00};`
		`# 10pm - 7am`
		`pf_restrictionhours = {"", ""};`
		`# no restrictions`
		`pf_restrictionhours = {"*", "*"};`
		`# no restrictions`

pf_restrictiondates	List	Configures the actual date restrictions applied if `pf_enabletimerestrictions` is set to **true**. Specify Start and End dates using yyyy/mm/dd format. EXAMPLES:

Profile Variable	Value Type	Explanation

		`pf_restrictiondates = {"2012/01/01", ""};`
		`    # no expiry`
		`pf_restrictiondates = {"", "2012/01/01"};`
		`    # no start date`
		`pf_restrictiondates = {"2012/01/01", "2012/12/31"};`
		`    # check start and end date`
		`pf_restrictiondates = {"", ""};`
		`    # no restrictions`
		`pf_restrictiondates = {"*", "*"};`
		`    # no restrictions`

pf_restrictiondow	List	Configures the day of the week restrictions applied if `pf_enabletimerestrictions` is set to **true**. Specify Days in any order using the lower case 3-letter abbreviation {"fri","sat","sun","mon","tue","wed","thu"};
		EXAMPLES:
		`pf_restrictiondow = {"mon","tue","wed","thu","fri"};`
		`    # weekdays only`
		`pf_restrictiondow = {"fri","sat","sun","mon","tue","wed","thu"};`
		`    # all days`
		`pf_restrictiondow = {};`
		`    # no restrictions`

Table 14: How settings

Profile variable	Value type	Explanation
Shell Settings		
pf_allow-shells	List	List of allowed shells. Do not specify full paths. This list is not compared with the `runcommand`, instead it is compared with the special `pmshell_prog` variable set by a Privilege Manager for Unix shell. Only applies to `pmksh`, `pmcsh`, `pmsh`, `pmbash`, and `pmshellwrapper`. EXAMPLE: `pf_allowshells ={"pmksh", "pmcsh", "pmsh", "pmshellwrapper"};`
pf_restric-	Boolean	Set to **true** to run the shell in restricted mode.

Profile variable	Value type	Explanation
ted		This means: • user cannot change directory • user cannot change PATH, ENV, SHELL • user can only run programs in PATH • no absolute/relative paths allowed • user cannot use io redirection with the '>' or '<' characters Only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`. EXAMPLE: `pf_restricted = true;`
pf_ shellreadonly	List	List of environment variables to treat as read-only. In restricted mode, the PATH, ENV, and SHELL variables are always read-only. Only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`. EXAMPLE: `pf_shellreadonly = {};`
pf_shellcwd	Sting	Defines the initial directory where the shell program will be run. The default is to use the runuser's home directory. This is particularly relevant for shells running in restricted mode, where the user cannot change the directory. This only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`. EXAMPLE: `pf_shellcwd = use_rundir;`

pf_shellpath	String	Defines the PATH that will be applied for the shell session. The default is to set standard paths, and add the runuser's home directory, and the current directory. This is particularly relevant for shells running in restricted mode, where the user cannot change the PATH, and can only run commands relative to the configured PATH.
		This only applies to `pmksh`, `pmcsh`, `pmsh`, and `pmbash`.
		EXAMPLE:
		`pf_shellpath = {`
		`"/usr/bin",`
		`"/bin",`

Profile variable	Value type	Explanation
		use_rundir, ".", };

Exploring profiles

To understand what happens when the Privilege Manager for Unix policy server receives a request, let's assume the default profile-based policy (pmpolicy) has been configured and user jbloggs issued a pmrun id command from host qpmhost01.

A pmmasterd process on the policy server receives the request, and pmrun sends it details about the request which are recorded as event variables (for example, user="jbloggs", command="id", submithost="qpmhost01", runhost="qpmhost01", date="2013/01/01", time="15:00:00").

With instructions from the code in profileBasedPolicy.conf, pmmasterd looks through each of the profiles until it finds a match between the profile variables (such as, pf_authusers) and the corresponding variables from the request (such as, user).

Note that pmshell and pmshellwrapper requests (such as, pmksh or pmshellwrapper_bash), the code directs pmmasterd to look through the shell profiles instead.

The default profile-based policy (pmpolicy) comes with four profiles: admin, demo, helpdesk, and webadmin.

- The admin profile is skipped because its pf_authusers lists only includes the root user.
- The helpdesk and webadmin profiles are disabled because their pf_enableprofile variables are set to **false**.
- This only leaves the demo profile, which is listed below.

```
################################################################################
# Privilege Manager for Unix Profile: demo
#
# This profile permits any user from any host to submit the "id" and "whoami"
commands
# to be executed as the root user on the local host.
################################################################################

pf_profiledescription= "Permit root access to id and whoami for demo purposes";

# Enable profile
pf_enableprofile= true;

# Enable Keystroke Logging
```

```
pf_enablekeystrokelogging= true;

# No authentication required
pf_enableauthentication= false;

# Apply time restrictions
pf_enabletimerestrictions= true;

# Only permit execution between 7am and 7pm
pf_restrictionhours= {"7:00","19:00"};

# No date restrictions
pf_restrictiondates= {"",""};

# Do not permit user to run remotely using pmrun -h
pf_enableremotecmds= false;

# Run these commands as root user
pf_authuser= "root";

# Run these commands as root's primary group on runhost
pf_authgroup= use_rungroup;

################################################################################
# Profile Membership
################################################################################
# Allow all users to run these commands
pf_authusers={
ALL
};

# allow session to be requested from any host
pf_authsubmithosts={
ALL,
};

# allow session to run on any host
pf_authrunhosts={
ALL,
};

# Only permit commands if run from /usr/bin or /bin
pf_authpaths={
"/usr/bin",
"/bin",
};

# permit id with any number of args (or none)
# permit whoami, only if run with no args
pf_authcmds={
"id **",
"whoami",
};
```

The demo profile is selected because the who, what, where, and when criteria match the request.

Table 15: Matching the request to the demo profile

Criteria	Demo Profile Variables	Request Event Variables	Match?
Who	pf_authusers={ALL};	user="jbloggs"	Yes
What	pf_authcmds= {"id **", "whoami"}; pf_authpaths= {"/usr/bin","/bin");	command="id" (n/a, path validated by pmlocald)	Yes
Where	pf_authsubmithosts={ALL}; pf_autrunhosts={ALL};	submithost="qpmhost01" runhost="qpmhost01"	Yes
Yes			

The policy is not able to validate the command path against pf_authpaths, since an absolute path to the command was not provided with the request. Because of this, pmmasterd accepts the request without checking the path, and leaves pf_authpaths to be validated by the pmlocald.

Once the policy selects a profile, other profile variables may affect how requests are processed. For example, pf_enableauthentication specifies whether password authentication is required.

If the root user issued the same pmrun request, the admin profile would have been selected. Even though both the admin and demo profiles match the request, the admin profile matches first.

Customizing the default profile-based policy (pmpolicy)

The default profile-based policy (pmpolicy) includes a profile_customer_policy.conf file, which you may edit to include customized policy code. This policy file defines the following stub functions and procedures that allow your custom code to run at specific points during the policy evaluation.

fn_customer_init()

This function is called once per policy evaluation, at the start of the policy's main body (located near the end of the profileBasedPolicy.conf file), just after the policy includes the profile_customer_policy.conf file.

fn_custom_profile_init()

This function is called after matching the user or group to a profile (or shell profile) but before checking anything else. You can find the function in procedure pr_processProfile() in the profileBasedPolicy.conf file.

This function can cause the current profile selection to fail by returning a false value.

pr_custom_profile_reset()

Use this procedure to reset custom profile variables added to the profile_customer_policy.conf file. This procedure is called when the profile match fails.

fn_log_and_accept_custom()

This function is called just before the request is accepted, after the request has been successfully matched to a profile. The function is called from the fn_log_and_accept() function in the profileBaseProfile.conf file.

Customization example - pf_forbidusers list

This example demonstrates how to create a new profile variable, pf_forbidusers, that you can use in any profile or shell profile. The customization will cause the profile selection to fail when the user is in the pf_forbidusers list, even if the user matches pf_authusers. This would allow you to blacklist specific users from any profile or shell profile.

The following is an updated profile_customer_policy.conf file indicating the modifications in bold.

```
############################################################################
# One Identity Privilege Manager for Unix Profile Policy V600 (XXX)
# One Identity 2013
#
# Sample Default Policy Generated  for QPM4U
#
# This policy is included by file: profileBasedPolicy.conf
#
# This allows customization at certain points while reading profiles. The
# following functions are provided:
# - fn_log_and_accept_custom
# - fn_custom_profile_init
# - pr_custom_profile_reset
# - fn_customer_init
############################################################################
# custom profile variables
pf_forbidusers={};

#########################################################################
```

```
# FUNCTION: fn_log_and_accept_custom
#
# This function is called by pr_log_and_accept to do any
# customer-specific actions required, just before accepting the request.
#
##########################################################################
function fn_log_and_accept_custom()
{
   return true;
}

##########################################################################
# FUNCTION: fn_custom_profile_init
# Do any custom config required for a profile.
# This is called after matching user/group to a profile,
# but before checking anything else.
##########################################################################
function fn_custom_profile_init()

{
   if (user in pf_forbidusers)
      return false;
   return true;
}

#########################################################################
# PROCEDURE: pr_custom_profile_reset
# Reset any custom variables after processing a profile
##########################################################################
procedure pr_custom_profile_reset()
{
   #reset these for each profile read
   pf_forbidusers={};
   return;
}

##########################################################################
# FUNCTION: fn_customer_init
# Do any custom config required for the policy
# This is called before processing any profiles.
##########################################################################
function fn_customer_init()
{
   return true;
}
```

The initial definition of the variable (`pf_forbidusers={};`) is near the top of the file. In order to be globally accessible, the variable must be defined outside of any function or procedure call. The same statement is also in the `pr_custom_profile_reset()` procedure so that the variable is reset before a new profile (or shell profile) is read. Finally, some code was added to `fn_custom_profile_init()` to return **false** if the user is listed in the variable.

If you add the following to the `demo` profile, user `jbloggs` would no longer be able to successfully run `pmrun id` using that profile:

```
pf_forbidusers={"jbloggs"};
```

Policy scripting tutorial

This section introduces you to the basics of policy scripting through a series of seven semi-interactive lessons. However, before you begin, please note: One Identity assumes you:

- have Privilege Manager for Unix installed successfully
- are running Privilege Manager for Unix with the pmpolicy type

The first seven lessons introduce you to some of the simpler constructs and capabilities of Privilege Manager for Unix's policies. Each lesson is designed to allow you to run the policy files on your own test system, with minimal changes, enabling you to learn the basics of policy scripting quickly.

Following the seven basic lessons are three advanced lessons designed to extend your knowledge and understanding of creating policies.

Before you start the lessons

1. Install the example policy file.
2. Create test users
3. Set Lesson number variable

Install the example policy file

Before you start the lessons, you must install the example policy file. This procedure instructs you to create a temporary directory and then use the `pmpolicy` command with a `checkout` sub-command to checkout the current policy into the temporary directory you just created.

To install the main example policy file

1. Create a temporary directory:

   ```
   # mkdir /tmp/policy
   ```

2. Checkout the current policy:

```
# /opt/quest/sbin/pmpolicy checkout -d /tmp/policy
```

```
** Validate options                                 [ OK ]
** Checkout to /tmp/policy/policy_pmpolicy
** Create directory                                 [ OK ]
** Check out working copy                           [ OK ]
** Copy files                                       [ OK ]
** Perform syntax check                             [ OK ]
```

3. Change to the temporary directory:

```
# cd /tmp/policy/policy_pmpolicy
```

4. Run the `pmpolicy masterstatus` command and note the current revision number.

```
#pmpolicy masterstatus
** Validate options                                 [ OK ]
** Report details of production copy
** Check out working copy (HEAD revision)           [ OK ]
** Check if directory contains a working copy       [ OK ]
   - Directory contains an svn working
copy:/var/opt/quest/qpm4u/pmpolicy/.scratch/._29076
** Check current status of working copy             [ OK ]
** Report details of production copy                [ OK ]
   - Production Policy File                   :
/etc/opt/quest/qpm4u/policy/pm.conf
   - Checked out at                           : 2012-11-30 16:23
   - Current Revision                         : 1
   - Latest Trunk Revision                    : 1
   - Locally modified                         : NO
```

5. Copy the main example policy into place:

```
# cp /opt/quest/qpm4u/examples/pm.conf pm.conf
cp: overwrite `pm.conf'? y
```

Policy file

This is the main policy file that Privilege Manager for Unix uses to drive through the lessons.

The other sample policy files for the lessons are also in the `examples` directory:

```
/opt/quest/qpm4u/examples/example1.conf
/opt/quest/qpm4u/examples/example2.conf
/opt/quest/qpm4u/examples/example3.conf
/opt/quest/qpm4u/examples/example4.conf
/opt/quest/qpm4u/examples/example5.conf
/opt/quest/qpm4u/examples/example6.conf
/opt/quest/qpm4u/examples/example7.conf
/opt/quest/qpm4u/examples/example8.conf
/opt/quest/qpm4u/examples/example9.conf
/opt/quest/qpm4u/examples/example10.conf
```

6. Use the `commit` sub-command to start using the policy:

   ```
   # pmpolicy commit -d /tmp/policy
   ```

   ```
   ** Validate options                                              [ OK ]
   ** Commit copy in directory:/tmp/policy/policy_pmpolicy
   ** Check directory                                               [ OK ]
   ** Perform syntax check                                          [ OK ]
   ** Verify files to commit                                        [ OK ]
   Please enter the commit log message:                  example pm.conf
   ** Commit change form working copy                               [ OK ]
   ** Committed revision 2
   ```

7. When you are finished with the examples, revert the original main policy file, as follows:

   ```
   # pmpolicy revert -r 1
   ** Validate options                                              [ OK ]
   ** Revert to revision:1
   ** Check out working copy (trunk revision)                       [ OK ]
   ** Check out working copy (revision 1)                           [ OK ]
   ** Check required revision                                       [ OK ]
   ** Get file list for trunk                                       [ OK ]
   ** Get file list for selected revision                           [ OK ]
   ** Copy file:pm.conf                                             [ OK ]
   ** Perform syntax check                                          [ OK ]
   ** Verify files to commit                                        [ OK ]
   Please enter the commit log message: revert to original
   ** Commit change from working copy                               [ OK ]
   ** Committed revision 3
   ```

See Main policy configuration file on page 101 to see the example policy file used in these lessons.

Create test users

For each lesson in this hands-on tutorial, you are required to log on as `root` and then switch to a test user. Then, at the conclusion of each lesson, switch back to `root` to get ready to start the next lesson.

To work through these lessons, you need to create users called *demo*, *dan*, and *robyn* on your test system, as the policy file is based around these default users.

To create the test users

1. Log in to your test system as the `root` user.
2. Create the *demo*, *dan,* and *robyn* test users to use during the lessons.

Set Lesson number variable

Lessons 1-10 are controlled by an environment variable called LESSON. Set this to a number in the range 1 through 6, using the following command:

```
LESSON=1; export LESSON
```

The main policy file, `pm.conf`, reads the LESSON and LESSON_USER environment variables and assigns their values to the PMLESSON and PMLESSON_USER policy variables, respectively.

The following example instructs you to run a fictitious command, `fred`, under Lesson 1.

You use the `pmrun` command to submit commands to Privilege Manager for Unix. Try entering `fred` using `pmrun`.

To enter a fictitious command

1. At the command line, run:

   ```
   # su demo
   $ pmrun fred
   ```

   ```
   Lesson 1 is selected
   -------------LESSON 1 DESCRIPTION--------------------------
   Policy file /opt/quest/pm4u/examples/linux-intel/example1.conf
   ------------------------------------------------------------
   This basic lesson uses a policy allowing users dan and demo
   the rights to run any command as root.
   ```

```
For example, to test this, enter the command pmrun whoami
which will return the value root as the logged in user.
-----------------------------------------------------------
fred
3201.063 Exec of fred failed: Command not found
```

As you can see, the policy informs you which lesson is selected and also provides the path to the associated policy file which contains this lesson fragment.

The policy files are reproduced in Sample policy files on page 101 for your reference, but you are encouraged to look at the digital copies of these files and experiment with the constructs that they contain once you have completed the lessons.

Introductory lessons

The first seven lesson introduce you to some of the simpler constructs and capabilities of Privilege Manager for Unix's policies. Each lesson builds upon the precepts of the last lesson. By the end of the seventh lesson you will have sufficient knowledge to start building your own policies.

These are the introductory lessons:

- Lesson 1: Basic policy
- Lesson 2: Conditional privilege
- Lesson 3: Specific commands
- Lesson 4: Policy optimization with list variables
- Lesson 5: Keystroke logging
- Lesson 6: Conditional keystroke logging
- Lesson 7: Policy optimizations

Lesson 1: Basic policy

This lesson introduces the basic concept of running a command at a privileged level. For a given list of users (in this case, `dan` and your defined `LESSON_USER`), run the command as `root`.

Here is the relevant policy code:

```
if (user=="dan" || user==PMLESSON_USER) {
 runuser="root";
 accept;
 }
```

If the policy server evaluates the policy without reaching an explicit "accept" statement, the request is rejected.

Be sure to:

- Set the LESSON variable to 1.
- Switch to your test user.
- Enter the command `pmrun whoami`.

Text in bold represents commands you enter; the resulting output is shown in normal font. The command output for the `pmrun` commands below has been slightly modified for brevity.

```
# LESSON_USER=demo; export LESSON_USER
# LESSON=1; export LESSON
# su demo
$ whoami
demo
$ pmrun whoami
root
$ exit
```

As you can see the result of the `whoami` command without a `pmrun` prefix shows that you are logged in as user `demo`. Repeating the command with a `pmrun` prefix, shows that you ran the command as `root`.

Here is the policy code that implements this behavior:

```
if (user=="dan" || user==PMLESSON_USER) {
runuser="root";
accept;
}
```

If the user who submitted the `pmrun` request matches either "dan" or the PMLESSON_USER variable, the `runuser` is set to "root" and the request is accepted.

The `exit` command at the end returns you to the `root` shell before proceeding to the next lesson.

Refer to Lesson 1 Sample: Basic policy on page 103 to see the sample policy used in this lesson.

Lesson 2: Conditional privilege

This lesson builds upon the previous lesson by narrowing the conditions under which you can run the commands as `root`. It introduces the use of a policy variable, `dayname`, and the function, `timebetween()`, to ensure that you can only run commands within the predetermined time frame of typical office hours (weekdays, between 8:00 a.m. and 5:00 p.m.).

The `dayname` variable and the `timebetween()` policy function are used to reject requests outside office hours:

```
if(dayname=="Sat" || dayname=="Sun" || !timebetween(800,1700))
   reject;
```

This lesson assumes that the current date and time are within this time frame.

```
# LESSON=2; export LESSON
 # su demo
```

Now, change the system date and attempt the command again using the following commands:

```
$ pmrun date mmdd2100
 Thu Feb 26 21:00:00 EDT 2012
 $ pmrun date mmdd2100
 Request Rejected by pmmasterd on UPMhost
 $ exit
```

where:

- mm stands for month (for example, 03 for March)
- dd stands for day (for example, 10 for the 10th)

The output shown above illustrates that the first attempt to set the date succeeded because the system date was within normal office hours. The second attempt fails because the time is now set outside of normal office hours.

Remember to reset the correct time on your system by running the date command as the root user.

Refer to Lesson 2 Sample: Conditional privilege on page 104 to see the sample policy used in this lesson.

Lesson 3: Specific commands

This lesson narrows the scope of which commands you can run with root privilege. The permitted list of commands is ls, hostname, and kill. Any other attempt to run a privileged command is rejected.

The "command" variable stores the command name issued by pmrun:

```
if (command == "ls" || command == "hostname" || command == "kill") {
      runuser = "root";
      accept;
 }
```

```
# LESSON=3; export LESSON
# su demo
$ pmrun shutdown
Request Rejected by pmmasterd on <UPMhost>
$ pmrun hostname
UPMhost
$ exit
```

where <UPMhost> is the host name

Refer to Lesson 3 Sample: Specific commands on page 105 to see the sample policy used in this lesson.

Lesson 4: Policy optimization with list variables

This lesson improves upon the design of Lesson 3, making the policy easier to read and faster to interpret with the introduction of list variables. List variables represent groups of data, in this case `users` and `commands`, which you can use in multiple places as values for test constraints.

```
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill"};

if (user in adminusers || user==PMLESSON_USER)
     { if (command in adminprogs)
          { runuser = "root";
               accept;
          }
     }
```

The "in" operator is used to test whether a variable matches a member of a list:

```
# su demo
$ pmrun shutdown
Request Rejected by pmmasterd on UPMhost
$ pmrun ls /etc/opt/quest/qpm4u
pm.settings policy
```

Refer to Lesson 4 Sample: Policy optimizations with list variables on page 106 to see the sample policy used in this lesson.

Lesson 5: Keystroke logging

This lesson introduces two new and important elements of policy writing. You can enable keystroke logging (I/O logging) at any point, and you can configure it to be conditional on any required elements.

This example enables keystroke logging when the permitted user runs these two commands, the csh and ksh shells; the user can run all other commands as root but without logging keystrokes.

Setting the "iolog" variable to a filename creates a keystroke log with that filename:

```
iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
```

You must choose the filename of the log file carefully. Its location and name are under the complete control of the policy script and in order to ensure that the file is unique, use the mktmp() function.

```
# LESSON=5; export LESSON
# su demo
$ pmrun csh
```

This request is logged in: /var/adm/pm.demo.csh.wXYeyn

In the example shown above, the log filename is displayed and the csh session is started. Now enter commands to create I/O logging and then exit back to the parent shell.

```
#  date
 #  cal
 #  hostname
 #  whoami
 #  exit
 $  exit
```

The output from these commands has been omitted for clarity.

It is now possible to replay this keystroke log file to display the session as seen by the demo user. Run the following command as root.

```
# /opt/quest/sbin/pmreplay /var/adm/pm.demo.csh.wXYeyn
```

Experiment with the controls within pmreplay to move backwards and forwards within the log session, using these commands:

Table 16: Replay controls

Control	Description
g	Go to start
G	Go to end
[Space] bar	Go to next input
t	Display time stamp
v	Dump variables

Control	Description
Backspace	Previous position
Ctrl	Next position
Quit	

Refer to Lesson 5 Sample: Keystroke logging to see the sample policy used in this lesson.

Lesson 6: Conditional keystroke logging

This lesson extends the logging example from the previous lesson, adding an exclusion to prevent privileged access outside of office hours, effectively combining the functionality you saw in lesson two, and displaying a message to the requesting user in such a situation:

```
adminusers = {"dan", "robyn"};
 adminprogs = {"ls", "hostname", "kill", "csh", "ksh", "pmreplay"};

 adminusers=append(adminusers,PMLESSON_USER); #Add the lesson user to list

 if (user in adminusers && command in adminprogs)
      { runuser = "root";
           if (command in {"csh", "ksh"})
                { iolog = mktemp("/var/adm/pm." + user + "." + command +
".XXXXXX");
                   print("This command will be logged to:", iolog);
                }

           if (user in adminusers && (!timebetween(800,1700) || dayname in
{"Sat", "Sun"}))
                { print ("Sorry, you can't use that command outside office
hours.");
                   reject;
                }

           accept;
 }
```

The above policy allows several admin programs to run, but only enables keystroke logging for the interactive shells.

```
# LESSON=6; export LESSON
# date mmdd1000
# su demo
$ pmrun hostname
UPMhost
$ exit
```

```
# date mmdd2200
# su demo
$ pmrun hostname
Sorry, you can't use that command outside office hours.
Request Rejected by pmmasterd on UPMhost
$ exit
```

where in the date commands, mm and dd refer to the two-digit representations of the month and day respectively.

In this example, you set the date as root before switching to demo, your test user. With the date initially set to a date/time combination which falls within office hours, Privilege Manager for Unix accepts the command.

Privilege Manager for Unix rejected the command and displayed a message when you exited back to the root shell, set the date/time to one outside of office hours, switched back to the test user, demo, and repeated the exercise.

Having reached this point you have established a good repertoire of policy constructs which form the basis of most policy file definitions. The use of list variables to hold constraint information used in combination with conditional tests using the if() construct represents the core function of most policy rules.

You use the print() and printf() functions to display messages and information throughout the policy. To control the keystroke logging, you use the value of the iolog system variable and the mktemp() function.

Remember to reset the correct time on your system by running the date command as the root user.

Refer to Lesson 6 Sample: Conditional keystroke logging on page 108 to see the sample policy used in this lesson.

Lesson 7: Policy optimizations

In this final interactive lesson, you will look at methods you can use to optimize your policy using all of the constructions we have covered so far:

- list variables
- constraint tests
- I/O logging
- message display

Additionally, you are introduced to the concept of requesting a password as confirmation before a certain command can be run.

One Identity recommends that you examine the policy and make any necessary modifications to establish the password validation test performs as expected.

```
# LESSON=7; export LESSON
# date mmdd2200
# su dan
$ pmrun hostname
Sorry, you can't use that command outside office hours.
Request Rejected by pmmasterd on UPMhost
$ exit
# su robyn
$ pmrun hostname
$ Password: <type in Robyn's password>
UPMhost
```

Remember to reset the correct time on your system by running the `date` command as the `root` user.

This lesson expands on the example in lesson 6. First, you forbid `dan` from running `admin` commands outside normal office hours. Then, because you saved the boolean value "officehours" earlier, you can check it again, this time to request for `Robyn`'s password if they attempt to run a command outside office hours.

```
officehours = timebetween(800, 1700) && dayname !in {"Sat", "Sun"};
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill", "csh", "ksh"};
if (user in adminusers && command in adminprogs) {
runuser = "root";
if (user == "dan" && !officehours) {
print("Sorry, you can't use that command outside office hours.");
reject;
}
if (user == "robyn" && !officehours) {
if (!getuserpasswd(user)) reject;
}
accept;
}
```

Refer to Lesson 7 Sample: Policy optimizations on page 109 to see the sample policy used in this lesson.

Advanced lessons

The remaining lessons are theoretical discussions covering the changes to scripts and leave the reader to consider modification and experimentation as exercises.

These lessons are not designed to be interactive. However, if you work through the sample policies, making changes, and trying out the policy files in the same way you did for Lesson 1 through 7, you will extend your understanding of the process, approach, and style required to create policies.

The advanced lessons are:

Lesson 8: Controlling the execution environment

This policy file introduces a number of environmental controls that give you greater flexibility and control over the command and user execution environment.

```
if (cwd != "/usr" && !glob("/usr/*", cwd))
   runcwd = "/tmp";
```

The first uses the `runcwd` variable which gives you the ability to examine and override the working directory in which the requested command runs. In this example, you allow commands to be run from `/usr` and its subdirectories, but you run all other commands from `/tmp`.

```
if (argc > 2)
   runargv = range(argv, 0, 2);
```

You can also control the number of arguments specified on the requested command line. You can examine the number of arguments together with the value of each argument, as well as remove, modify, or supplement them with additional arguments not previously present on the original command line.

```
runuser = "root";
rungroup = "bin";
if (command != "hostname")
   runhost = submithost;
```

You can also examine the rungroup and the host on which the command is destined to run and override them.

```
keepenv("TERM", "DISPLAY", "HOME", "TZ", "PWD", "WINDOWID", "COLUMNS", "LINES");
setenv("PATH",
"/usr/ucb:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:" +
"/usr/X11/bin:/usr/etc:/etc:/usr/local/etc:/usr/sbin");
```

Control of the run environment is vitally important as you can use environment variables to exploit security vulnerabilities in some UNIX programs, so one aspect of the policy can be to cleanse the execution environment to make sure there is nothing which could be considered unsafe. A common requirement within policy files is to ensure that the PATH is cleansed, removing any user appended paths which may be higher up the search path, where a user-created script may be lurking.

```
runumask = 022;
runnice = -4;
```

You can control many other aspects of the execution environment including the `nice` value and `umask`.

Refer to Lesson 8 Sample: Controlling the execution environment on page 111 to see the sample policy used in this lesson.

Lesson 9: Flow control

This lesson introduces you to another execution control construct using `switch()`, `case`, and `break` statements which allow you control which parts of the script are to run.

```
adminprogs = {"ls", "hostname", "kill", "csh", "ksh", "echo"};

 if (command in adminprogs) {
      switch (dayname) {
           case "Mon":
           case "Wed":
           case "Fri":
                adminusers = {"dan", "robyn"};
                break;
           case "Tue":
           case "Thu":
                adminusers = {"robyn", "cory"};
                break;
           default:
                adminusers = {};
      }
      if (user in adminusers) {
           runuser = "root";
           accept;
      }
 }
```

In this example, you use the `switch` and `case` statements to control which users are considered to be *admin* users on any given day of the week. Execution commences when the first `case` statement matches the condition. It proceeds until it encounters the end of the `switch` statement or reaches a `break` statement.

Refer to Lesson 9 Sample: Flow control on page 113 to see the sample policy used in this lesson.

Lesson 10: Basic menus

This final lesson demonstrates the use of a rudimentary menu system which you can present to the user when he enters the `adminmenu` command.

```
if(command=="adminmenu") {
     print("========= Admin Menu =========");
     print("1) Add users");
     print("2) Start a backup");
     print("3) Change ownership of a file");
     print("4) Fix line printer queues");
     choice = input("Please choose one: ");

     switch(choice) {
          case "1":
               if(!getstringpasswd("m9xxg7B4.v8Ck", "Type in the adduser
     password: ", 2))
                    reject;
               runcommand = "/usr/local/bin/adduser";
               runuser = "root";
               break;
          case "2":
               runcommand = "/usr/local/bin/dobackup";
               runuser = "backup";
               break;
          case "3":
               runcommand = "/usr/bin/chown";
               runuser = "root";
               break;
          case "4":
               runcommand = "/usr/lib/lpadmin";
               runuser = "root";
               break;
          default:
               printf("\"%s\" was not a valid choice. Sorry.\n", choice);
               reject;
     }

     if (choice == "3") {
          file_name=input("Please enter the new owner's name then file name:
");
          arguments = split(file_name);
          runargv = insert(arguments, 0, "Spacer");
     }
     print("** Command to be run :", runcommand);
     print("** User to run command as :", runuser);
     accept;
}
```

This example shows how to gather input from the user, check the value of a literal hard-coded password, and manipulate command line arguments. It is purely illustrative of the scope and scale of what you can achieve from within a policy file, although there is much more that has not been covered in this lesson.

Refer to Lesson 10 Sample: Basic menus on page 114 to see the sample policy used in this lesson.

Sample policy files

Electronic copies of the policy file samples used in each lesson are located in the `/opt/quest/qpm4u/examples` directory and they are reproduced for you in this section.

Main policy configuration file

```
##########################################################################
# Privilege Manager for Unix example configuration file
# One Identity 2013
# Example File : pm.conf
#
# Establish which Lesson has been selected and include the appropriate file
# accordingly
##########################################################################
PMINST=getenv("INSTBASE","/opt/quest/qpm4u");
PMLESSON=atoi(getenv("LESSON","1"));
EXAMPLEDIR=PMINST + "/examples";
if (PMLESSON<1 || PMLESSON>11)
   { printf("Invalid lesson %i selected, resetting to Lesson 1\n",PMLESSON);
      PMLESSON=1;
   }
system("clear");
printf("Lesson %i is selected\n",PMLESSON);
# The lessons take a user from the environment so that
# none of the scripts require modification before use
# this is taken from the environment variable LESSON_USER
# Make sure that you have set this a valid user which will
# be used for the purposes of this series of lessons.
PMLESSON_USER=getenv("LESSON_USER","demo");
if (PMLESSON_USER=="")
   { print("No user has been specified, user 'demo' will be assumed\n");
   }
if (user!=PMLESSON_USER)
   { print("----------------------- WARNING --------------------------");
     printf("Your currently logged in as %s\n",user);
     printf("Your selected user for the lessons is %s\n",PMLESSON_USER);
```

```
      printf("This may not be what you intended, try 'su %s'\n",PMLESSON_USER);
      print("------------------------------------------------------------\n");
   }
PML=sprintf("%i",PMLESSON);
switch (PML)
   {
      case "1":
         { include EXAMPLEDIR + "/example1.conf";
           break;
         }
      case "2":
         { include EXAMPLEDIR + "/example2.conf";
           break;
         }
      case "3":
         { include EXAMPLEDIR + "/example3.conf";
           break;
         }
      case "4":
         { include EXAMPLEDIR + "/example4.conf";
           break;
         }
      case "5":
         { include EXAMPLEDIR + "/example5.conf";
           break;
         }
      case "6":
         { include EXAMPLEDIR + "/example6.conf";
           break;
         }
      case "7":
         { include EXAMPLEDIR + "/example7.conf";
           break;
         }
      case "8":
         { include EXAMPLEDIR + "/example8.conf";
           break;
         }
      case "9":
         { include EXAMPLEDIR + "/example9.conf";
           break;
         }
      case "10":
         { include EXAMPLEDIR + "/example10.conf";
           break;
         }
}
```

```
reject;
```

See Install the example policy file on page 86 for details on installing the example policy file.

Lesson 1 Sample: Basic policy

```
#=================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example1
#
# This file to have permissions of 600 (rw-------), and be owned by
# root.
#=================================================================
#=================================================================
print("-------------LESSON 1 DESCRIPTION--------------------------");
printf("Policy file %s/examples/example1.conf\n",PMINST);
print("-----------------------------------------------------------");
printf("This basic lesson uses a policy allowing users %s and
dan\n",PMLESSON_USER);
print("the rights to run any command as root.\n");
print("For example, to test this enter the command pmrun whoami");
print("which will return the value root as the logged in user.");
print("-----------------------------------------------------------");
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");
if (user=="dan" || user==PMLESSON_USER) {
   runuser="root";
   accept;
}
#=================================================================
```

See Lesson 1: Basic policy on page 90 for details on using this sample policy file.

Lesson 2 Sample: Conditional privilege

```
#===============================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example2
#
# This file should have permissions of 600
# (rw-------).
# It must be owned by root.
#===============================================================
print("--------------- LESSON 2 DESCRIPTION ------------------");
printf("Policy file %s/examples/example2.conf\n",PMINST);
print("-------------------------------------------------------");
printf("This policy rejects attempts to run commands outside of normal\n");
printf("office hours for users %s and dan.\n",PMLESSON_USER);
print("Otherwise all commands will be run as root.\n");
print("Try running a few different programs like date, hostname");
print("and even your favourite shell (csh, bash, ksh)");
print("Try these with the time/date set both in and outside office hours");
print("Remember to prefix them with pmrun");
print("-------------------------------------------------------");
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");
#===============================================================
if (user=="dan" || user==PMLESSON_USER) {
   # Explicitly disallow commands run outside of regular office hours
   if(dayname=="Sat" || dayname=="Sun" || !timebetween(800,1700))
      reject;
   runuser = "root";
   accept;
}
#===============================================================
```

See Lesson 2: Conditional privilege on page 91 for details on using this sample policy file.

Lesson 3 Sample: Specific commands

```
#==================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example3
#
# This file should have permissions of 600
# (rw-------).
# It must be owned by root.
#==================================================================
print("------------------ LESSON 3 DESCRIPTION ------------------------");
printf("Policy file %s/examples/example3.conf\n",PMINST);
print("---------------------------------------------------------");
printf("This policy allows users %s and dan to run *some* programs as
root.\n",PMLESSON_USER);
print("Otherwise all other commands will be rejected.\n");
print("The permitted commands are kill, ls and hostname.");
print("Try running a few different programs and see what happens.");
print("Again, remember to prefix them with pmrun.");
print("---------------------------------------------------------");
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");
#==================================================================
if (user=="dan" || user==PMLESSON_USER)
   if (command == "ls" || command == "hostname" || command == "kill") {
      runuser = "root";
      accept;
   }
#==================================================================
```

See Lesson 3: Specific commands on page 92 for details on using this sample policy file.

Lesson 4 Sample: Policy optimizations with list variables

```
#=====================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example4
#
# This file should have permissions of 600 (rw-------).
# It must be owned by root.
#===========================================================================
print("------------------- LESSON 4 DESCRIPTION
-------------------------");
printf("Policy file %s/examples/example4.conf\n",PMINST);
print("------------------------------------------------------------------"
);
print("This lesson is identical to Lesson 3, but uses a different policy");
print("construct known as a list variable, making the policy simpler");
print("shorter and clearer to understand.");
print("Look at the policy files for lessons 3 & 4 and note the
differences.\n");
printf("This policy allows users %s, robyn and dan to run *some* programs as
root.\n",PMLESSON_USER);
print("Otherwise all other commands will be rejected.\n");
print("The permitted commands are kill, ls and hostname.");
print("Try running a few different programs and see what happens.");
print("Again, remember to prefix them with pmrun.");
print("------------------------------------------------------------------"
);
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");

#===========================================================================
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill"};
if (user in adminusers || user==PMLESSON_USER)
   { if (command in adminprogs)
      { runuser = "root";
         accept;
      }
   }
#===========================================================================
```

See Lesson 4: Policy optimization with list variables on page 93 for details on using this sample policy file.

Lesson 5 Sample: Keystroke logging

```
#===================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example5
#
# This file should go in /etc/pm.conf with permissions of 600 (rw-------).
# It must be owned by root.
#===================================================================
print("---------------- LESSON 5 DESCRIPTION -------------------");
printf("Policy file %s/examples/example5.conf\n",PMINST);
print("--------------------------------------------------------");
print("This lesson introduces keystroke logging.");
printf("Users %s, robyn and dan are permitted to run everything as
root,\n",PMLESSON_USER);
print("but commands csh and ksh will be fully keystroke logged.");
print("This means that all I/O during these shell sessions will be logged.");
print("The log file is created with mktmp() and the name is displayed.");
print("The logfile will be something like pm.dan.ksh.a545456\n");
print("Look closely at Lesson 5 to see how logging is enabled.\n");
print("The log files can be replayed with the pmreplay utility.\n");
print("Don't forget to prefix commands with pmrun.");
print("--------------------------------------------------------");
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");
#===================================================================
adminusers = {"dan", "robyn"};
# Add the provided lesson user so they need not adjust the policy
adminusers = append(adminusers,PMLESSON_USER);
if (user in adminusers)
   { runuser = "root";
     if (command in {"csh", "ksh"})
         { iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
            iolog_opmax=10000
            print("This request will be logged in:", iolog);
         }
      accept;
   }
=================================================================
```

See Lesson 5: Keystroke logging on page 93 for details on using this sample policy file.

Lesson 6 Sample: Conditional keystroke logging

```
#=================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example6
#
# This file should go in /etc/pm.conf with permissions of 600
# (rw-------).
# It must be owned by root.
#=================================================================
print("-------------- LESSON 6 DESCRIPTION --------------------");
os=osname();
printf("Policy file %s/examples/"+os+"/example6.conf\n",PMINST);
print("--------------------------------------------------------");
print("This lesson extends lesson 5 by adding some statements that cause");
printf("requests by %s, dan and robyn to be rejected if they arrive
outside\n",PMLESSON_USER);
print("of regular office hours (8AM until 5PM Monday to Friday).");
print("A message is printed to the user's screen if this happens.");
print("Once again examine the policy file, noting use of logical not
operator.");
print("Try altering the timebetween() and dayname tests and check the
results");
print("--------------------------------------------------------");
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");
#=================================================================
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill", "csh", "ksh", "pmreplay"};
adminusers=append(adminusers,PMLESSON_USER); #Add the lesson user to list
if (user in adminusers && command in adminprogs)
   { runuser = "root";
```

```
      if (command in {"csh", "ksh"})
         { iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
            print("This command will be logged to:", iolog);
         }
      if (user in adminusers && (!timebetween(800,1700) || dayname in {"Sat",
"Sun"}))
         { print ("Sorry, you can't use that command outside office hours.");
            reject;
         }
         accept;
}
#==================================================================
```

See Lesson 6: Conditional keystroke logging on page 95 for details on using this sample policy file.

Lesson 7 Sample: Policy optimizations

```
#==================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example7
#
# This file should go in /etc/pm.conf with permissions of 600
# (rw-------).
# It must be owned by root.
#==================================================================
print("---------------- LESSON 7 DESCRIPTION -------------------");
os=osname();
printf("Policy file %s/examples/"+os+"/example7.conf\n",PMINST);
print("---------------------------------------------------------");
print("This lesson extends lesson 6 using variables to store
constraints");
print("which you might want to use several times in the policy file.");
print("Here, we set a variable to store whether or not it is currently");
print("within office hours or not. By storing it in a variable, we can
refer");
print("to it several times later on in the file if need be, without having");
print("enter and resolve the whole lengthly constraint each time.");
print("\nIn this example, there are two bits which we are interested in");
print("whether or not it is currently within office hours. The first bit is");
print("the same as in lesson 6, disallowing dan's requests outside of");
print("office hours. The second bit, near the end, requires the user");
print("to type in robyn's password if robyn makes a request outside of
normal");
```

```
print("office hours. This would be useful to protect against the situation");
print("where a user leaves a terminal logged in overnight.");
print("--------------------------------------------------------");
i=0;
while (i<argc)
   { printf("%s ",argv[i]); # Redisplay the original command line for clarity
      i=i+1;
   }
printf("\n");
#==================================================================
# Here, we set officehours to true if it is within office hours (8AM until 5PM
# Monday to Friday), false otherwise.
officehours = timebetween(800, 1700) && dayname !in {"Sat", "Sun"};
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill", "csh", "ksh"};
# Add the provided lesson user
adminusers=append(adminusers,PMLESSON_USER);
if (user in adminusers && command in adminprogs)
   { runuser = "root";
      if (command in {"csh", "ksh"})
         { iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
            print("This command will be logged to:", iolog);
         }
      # Note how much more compact this is compared to example6.conf,
      # now that we can refer to the "officehours" variable.
      if (user == "dan" && !officehours)
         { print ("Sorry, you can't use that command outside office hours.");
         reject;
   }
   # Now we refer to "officehours" again. This time, if "robyn" is making
   # the request outside of office hours, robyn is asked to correctly
   # type in robyn's password. If it is not typed in correctly, the request
   # is rejected.
   if (user == "robyn" && !officehours)
      { if(!getuserpasswd(user)) reject;
      }
      accept;
}
#==================================================================
```

See Lesson 7: Policy optimizations on page 96 for details on using this sample policy file.

Lesson 8 Sample: Controlling the execution environment

```
#==================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example8
#
# This file should have permissions of 600
# (rw-------).
# It must be owned by root.
#==================================================================
#==================================================================
# This example shows how facets of a job's run-time operating
# environment can be set up using Privilege Manager for Unix.
# Although the policies listed here are arbitrary, their structure
# can be used as examples or how to implement your own real policies.
# For experimental purposes, replace "dan" and "robyn" with user
# names from your own site.
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill", "csh", "ksh", "echo"};
if (user in adminusers && command in adminprogs) {
   # What directory should this job run in? For this example, we
   # want to say that if the job is executed from any directory
   # under /usr, it can be allowed to execute in that directory.
   # If it is not being executed from a directory under /usr, it
   # should execute in /tmp.
   if(cwd != "/usr" && !glob("/usr/*", cwd))
      runcwd = "/tmp";
   # Do not allow more than 2 arguments to be specified to the
   # command. The range function is used here to return only the
   # first 3 arguments of the argv list. The first element is the
   # command name, the second element is the first argument to
   # the command, and the third element is the second argument
   # to the command.
   if(argc > 2)
      runargv = range(argv, 0, 2);
   # Require the request to run as root.
   runuser = "root";
   # Require the request to run in the "bin" group.
   rungroup = "bin";
   # if the command being run is "hostname", run that command on
   # whatever machine the user requests (by default, the same
   # machine that pmrun is run from, but this can be changed
   # using pmrun's -h argument). Otherwise, requests should only
   # run on the same machine that the pmrun request was
   # submitted from.
   if(command != "hostname")
      runhost = submithost;
   # Since environment variables can sometimes be used to
   # exploit security holes in UNIX programs and shell scripts,
   # we should be careful to set up the job's environment
   # variables safely. We start by deleting any and all
   # environment variables except those specified in the
   # following list.
   keepenv("TERM", "DISPLAY", "HOME", "TZ", "PWD", "WINDOWID",
   "COLUMNS", "LINES");
   # Next we explicitly set up the PATH variable, so that only
   # safe directories are on it. Note the use of + to
   # concatenate the value that we want to assign to the PATH
   # variable. We use + so that we can split it up over 2 lines
   # to avoid ugly end-of-line wrapping.
   setenv("PATH",
   "/usr/ucb:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:" +
   "/usr/X11/bin:/usr/etc:/etc:/usr/local/etc:/usr/sbin");
   # We ensure that the SHELL variable is set safely. If the
   # existing SHELL variable is set to a safe value, which we
   # define as any of /bin/sh, /bin/csh, or /bin/ksh, then we
   # use that value. If not, then we use /bin/sh.
   # Note: getenv reads from the "env" variable, setenv and
   # keepenv write to the "runenv" variable.
   safeshells = {"/bin/sh", "/bin/csh", "/bin/ksh"};
   if(getenv("SHELL") in safeshells)
      setenv("SHELL", getenv("SHELL"));
   else
      setenv("SHELL", "/bin/sh");
   # Set the command's umask to 022 -- this means that data
   # files created by the command will have rw-r--r--
   # permissions, and executable files will have rwxr-xr-x
   # permissions. Since the command will run as root, root will
   # own the files. Note that we specify a leading zero when
   # typing in umask values, so that the values will be interpreted in
   # octal.
   runumask = 022;
   # The command should run with a "nice" value of -4, so that it
   # runs with a high priority relative to other jobs on the
   # system.
   runnice = -4;
accept;
}
#==================================================================
```

See Lesson 8: Controlling the execution environment on page 98 for details on using this sample policy file.

Lesson 9 Sample: Flow control

```
#==================================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example9
# This file should have permissions of 600
# (rw-------).
# It must be owned by root.
#==================================================================
#==================================================================
# This example shows how the switch and case statement can be used.
# In this case, we allow different users to act as system
# administrators on different days of the week.
# For experimental purposes, replace "dan", "cory", and "robyn" with
# user names from your own site.
adminprogs = {"ls", "hostname", "kill", "csh", "ksh", "echo"};
if (command in adminprogs) {
   switch (dayname) {
      case "Mon":
      case "Wed":
      case "Fri":
         adminusers = {"dan", "robyn"};
         break;
      case "Tue":
      case "Thu":
         adminusers = {"robyn", "cory"};
         break;
      default:
         adminusers = {};
   }
   if (user in adminusers) {
      runuser = "root";
         accept;
   }
}
#==================================================================
```

See Lesson 9: Flow control on page 99 for details on using this sample policy file.

Lesson 10 Sample: Basic menus

```
#==============================================================
# Privilege Manager for Unix example configuration file
# One Identity 2013
#
# Example File : example10
#
# This file should have permissions of 600
# (rw-------).
# It must be owned by root.
#==============================================================
#==============================================================
# This example shows how to implement a menu system with 4 choices.
# Also, if the "adduser" program is to be run, a password must be
# entered correctly.
# For experimental purposes, replace "dan", "cory", and "robyn" with
# user names from your own site.
if(command=="adminmenu") {
   print("========= Admin Menu =========");
   print("1) Add users");
   print("2) Start a backup");
   print("3) Change ownership of a file");
   print("4) Fix line printer queues");
   choice = input("Please choose one: ");
   switch(choice) {
   case "1":
      # Reject the request if the password "123456" is not entered
      # correctly. The user gets only 2 chances to type in the
      # password. The encrypted version of the password seen here
      # was generated using pmpasswd. If you store encrypted
      # passwords in your config file, make sure you turn off read
      # permission on the file so that people cannot use password
      # cracking programs to guess them.
      if(!getstringpasswd("m9xxg7B4.v8Ck", "Type in the adduser password: ",2))
         reject;
      runcommand = "/usr/local/bin/adduser";
      runuser = "root";
      break;
   case "2":
      runcommand = "/usr/local/bin/dobackup";
      runuser = "backup";
      break;
   case "3":
      runcommand = "/usr/bin/chown";
      runuser = "root";
      break;
   case "4":
      runcommand = "/usr/lib/lpadmin";
      runuser = "root";
      break;
   default:
      printf("\"%s\" was not a valid choice. Sorry.\n", choice);
      reject;
}
if (choice == "3") {
   file_name=input("Please enter the new owner's name then file name: ");
   arguments = split(file_name);
   runargv = insert(arguments, 0, "Spacer");
}
   print("** Command to be run :", runcommand);
   print("** User to run command as :", runuser);
   accept;
}
#==================================================================
```

See Lesson 10: Basic menus on page 100 for details on using this sample policy file.

Advanced Privilege Manager for Unix Configuration

This section provides advanced information on how to configure and implement Privilege Manager for Unix:

- Privilege Manager for Unix shells
- Configuring Privilege Manager for Unix for policy scripting
- Configuring firewalls
- Configuring Kerberos encryption
- Configuring certificates
- Configuring alerts
- Configuring Pluggable Authentication Method (PAM)

Privilege Manager for Unix shells

Privilege Manager for Unix shells provide a means of auditing and controlling a user’s login session in a way that is transparent to the user, without the user having to preface commands with `pmrun`.

Privilege Manager for Unix provides enabled versions of these standard shells: `pmksh`, `pmsh`, `pmcsh`, and `pmbash`. Each shell uses the same policy file variables to control the behavior of the shell.

By default, all built-in shell commands are allowed to run without any further authorization by the shell; however, you must authorize all non-built-in shell commands. Once authorized, all commands are run locally by the shell with the authority of the user running the shell.

You can configure the level of control required for commands running from a shell in the policy file by configuring the policy file to either *forbid* commands or *allow* them to be run by the shell program without any further authorization to the policy server. You can also configure the policy file to authorize them as they are presented to the policy server for audit logging. Furthermore, you can configure keystroke logging for the shell session to be logged to a single I/O log file.

Privilege Manager for Unix shell features

Use a Privilege Manager for Unix shell to control or log Privilege Manager for Unix sessions, regardless of how you are logged in (for example, `telnet`, `ssh`, `rsh`, `rexec`).

You can use one of these Privilege Manager for Unix-enabled shells to create a fully featured shell environment for a user:

- `pmksh`: a Privilege Manager for Unix-enabled version of Korn Shell
- `pmsh`: a Privilege Manager for Unix-enabled version of Bourne Shell
- `pmcsh`: a Privilege Manager for Unix version of C Shell
- `pmbash`: a Privilege Manager for Unix version of Bourne Again Shell

Each shell provides command control for every command entered by a user during a login session. You can configure each command the user enters to be authorized with the policy server before it runs. This includes the shell built-in commands.

You can configure keystroke logging for the entire login session and login to a single file.

Alternatively, you can use `pmshellwrapper` to act as a Privilege Manager for Unix wrapper for any valid shell program on a host, or create a custom Privilege Manager for Unix shell by means of a shell script. In these cases, however, the individual commands run during the login session are not controlled by Privilege Manager for Unix.

To use `pmshellwrapper`, create a link using the name of the system shell you want to run. For example, to create a wrapper for bash, enter:

```
ln -s /opt/quest/libexe/pmshellwrapper/opt/quest/libexe/pmshellwrapper_bash
```

When you run the `pmshellwrapper_bash` program, it transparently runs `pmrun bash` instead.

For example, to create a custom Privilege Manager for Unix shell (a shell script that runs the actual shell using `pmrun`), run:

```
#!/bin/ksh
 tty 2>/dev/null 1>/dev/null
 x=$?
 if [ $x -ne 0 ]
 then
 exec /opt/quest/bin/pmrun ksh "$@"
 else
 exec /opt/quest/bin/pmrun -c -ksh "$@"
 fi
```

Add the full pathname of the shell program to the `/etc/shells` file if you are using `pmksh`, `pmsh`, `pmcsh`, `pmbash`, or `pmshellwrapper` on your system.

Forbidden commands

Use the `pmshell_forbid` list variable in the policy file to define a list of commands you want the shell to forbid without any further authorization by the policy server. The shell program interprets this list as a list of regular expressions. Privilege Manager for Unix checks each command a user enters against this list. If a match is found, it rejects the command without further authorization. These commands do not result in a *reject* entry in the event log as they are forbidden by the shell. You can also configure the message that is displayed when it issues one of these commands.

Allowed commands

Use the `pmshell_allow` list variable in the policy file to define a list of commands you want the shell to allow without any further authorization by the policy server. The shell program interprets this list as a list of regular expressions. Privilege Manager for Unix checks each command the user enters against this list. If a match is found, it allows the command without further authorization. These commands do not result in an *accept* entry in the event log as they are allowed by the shell.

Allowed piped commands

Use the `pmshell_allowpipe` variable in the policy file to configure a list of commands you want the shell to allow without further authorization by the policy server if the input to the command is a pipe. The shell program interprets this list as a list of regular expressions. Privilege Manager for Unix checks each command a user enters against this list if the input to the command is a pipe. If a match is found, it allows the command without further authorization. These commands do not result in an *accept* entry in the event log as they are allowed by the shell. This allows the shell to authorize commands only within a particular context.

For example, if the allowed pipe command list contains `grep`, as in:

```
grep "root" /etc/shadow
```

the shell authorizes the `grep` command as its input does not come from a pipe.

On the other hand, if you enter:

```
cat /etc/shadow | grep "root"
```

the shell only authorizes the `cat` command. The `grep` command is allowed without authorization.

Check shell built-in commands

Built-in shell commands are functions defined internally to the shell. You can apply a policy to shell built-in commands by setting `pmshell_checkbuiltins=1`. The shell does not create a new UNIX process to run a built-in command and does not access or run any program outside the shell to run a built-in command. The shell built-in commands usually include functions like `echo` and `cd`. The full list of shell built-in commands depends on the shell you are using; to see the command list for a particular shell, run the shell with the `-?` argument.

By default, shell built-in commands are not authorized to the policy server or checked against the *allow* and *forbid* lists.

You can set a flag to force the shell to treat all shell built-in commands as if they are normal, executable commands. If this flag is set, all built-in commands are compared with the *forbid* and *allow* lists, and if no match is found, they are presented to the policy server for authorization.

Read-only variable list

Use the `pmshell_readonly` list variable to define a list of environment variables in the policy file to be read-only in the shell. You can not change read-only variables during a shell session.

Running a shell in restricted mode

Set `pmshell_restricted=1` to configure the shell to run in restricted mode. Restricted mode applies these restrictions to the shell:

- A user cannot change the directory.
- A user cannot change the value of these parameters: `PATH`, `SHELL`, or `ENV`. You must set these up using the secure profile (if the user is running a login shell), or by setting these variables in the policy file.
- A user cannot run any command that is identified by an absolute or relative pathname, including absolute paths defined in shell aliases. The user can only run shell built-in commands or executable files that are in the read-only PATH. For example, the following commands are not allowed:
    - `/usr/bin/ls`
    - `./script.sh`
    - `alias ll='/bin/ls -F'`

The commands ls and script.sh are allowed if /usr/bin and . are in the PATH; the command ll would not be allowed because the substituted command is an absolute path.

- A user cannot use I/O redirection with the ">" or "<" characters.

  For example, the following command will fail:

  ```
  echo "hello" > /tmp/file
  ```

- A user cannot run in privileged mode (if supported by the shell).

If the shell is run as a login shell for a user, then during the login process, the relevant system and user profiles are loaded for that particular shell. During this sequence, the shell checks the ownership and permissions of each startup file loaded.

Any restrictions configured for the shell are not applied while loading a secure profile; that is, a file owned by root and only writable by root. Any restrictions configured for the shell are only applied if the profile is not secure. For example, if PATH is configured as a read-only variable in the policy file, and the built-in command cd is forbidden, then the PATH initialization in the secure system profile /etc/profile is allowed without restriction or authorization, but any attempt to change the PATH variable or to run the cd command in the insecure user's personal profile, or during the interactive login session will be forbidden.

Additional shell considerations

The order in which the restrictions are applied to the shell are:

1. forbidden commands list
2. allowed commands list
3. allowed pipe list, if the input is a pipe

The shell, and the commands run from within it, run as the selected runuser and rungroup for the shell program. Once the shell is running, you cannot change the runuser or rungroup for authorized commands within the shell. To run an individual shell command as a different user, run the pmrun <cmd>.

You can change the arguments to a command running within a shell, the environment variables, and the priority for a command. For example, if you configure the shell to authorize built-in commands, then you can prevent a user from changing to any directory other than the user's home directory by removing all except the first argument from the cd command. For example:

```
if (runcommand=="cd")
 {
      len=length(runargv);
      runargv=replace(runargv,1,len);
 }
```

The exec command is always forbidden if an attempt is made to run it from the top-level interactive shell process, as this would overlay the existing controlled Privilege Manager for Unix shell with an unrestricted shell. For example, an attempt to run this command from an interactive shell is forbidden:

```
exec /bin/sh
```

A Privilege Manager for Unix-enabled shell requires two connections to the policy server host. One is used for keystroke logging by the shell program itself, and one is used for authorizing commands to be run during the shell session.

Example

```
allowed_pmshells = { "pmsh", "pmcsh", "pmksh" };
# pmshell only defined if a shell or cmd within a shell
if (defined pmshell)
{
   # Configure Privilege Manager for Unix Shells
   if ( pmshell_cmd == 0) {
      if ( pmshell_prog in allowed_pmshells ) {
         print("Starting Privilege Manager for Unix Shell");

         pmshell_restricted=0;
            # Restricted Shell: 0=disable|1=enable
         pmshell_checkbuiltins=0;
            # Force checking of Shell BuiltIns: 0=disable|1=enable
      pmshell_allow={"ls", "man"};
            # list of commands to accept without further authorization.
      accept;
   }
   else {
      reject "You are not authorized to run this shell";
   }
}
# Authorize all commands executed from within a shell
else {
   # Define list of commands allowed to run as the root user.
   privileged_cmds = { "/sbin/service", "/usr/bin/kill", "/usr/bin/id" };
      if ( command in privileged_cmds ) {
         runuser = "root";
         rungroup = "root";
```

```
        }
        print("Executing command as user: " + runuser);
        accept;
    }
}
```

Configuring Privilege Manager for Unix for policy scripting

If you have successfully completed the Privilege Manager for Unix installation and you are new to Privilege Manager for Unix, One Identity recommends that you work through the semi-interactive lessons in Policy scripting tutorial on page 86. This will help familiarize you with the basic functionality of Privilege Manager for Unix.

Configuration prerequisites

Before you configure Privilege Manager for Unix, make sure

- TCP/IP is configured and running on all relevant machines.
- Applications, files, and accounts you wish to access using Privilege Manager for Unix are available from all servers.
- `pmrun` is in a directory in the user's `PATH` and is executable. `pmrun` is owned by `root`, and has the `SETUID` bit turned on.
- `pmmasterd` and `pmlocald` are set up in `/etc/services` (this is created by the `pmsrvconfig` installation script).

  This is a sample services file:

  ```
  pmmasterd 12345/tcp
  pmlocald 12346/tcp
  ```

- The `/etc/opt/quest/qpm4u/pm.settings` file has been set up (this is done by `pmsrvconfig`).

  This is a sample `pm.settings` file, showing you the defaults for each setting:

```
kerberos NO
encryption AES
reconnectClient NO
reconnectAgent NO
clientVerify NONE
FailOverTimeOut 10
Certificates NO
selecthostrandom YES
shortnames YES
syslog YES
pmservicedLog /var/log/pmserviced.log
masterport 12345
localport 12346
tunnelport 12347
masters qpm4u
pmmasterdlog /var/log/pmmasterd.log
pmmasterdEnabled YES
pmmasterdOpts -ar
policymode pmpolicy
pmlogGroup pmlog
```

Configuration file examples

The topics that follow walk you through some detailed examples for the configuration file policy.

To install the configuration file examples on your machine

1. Checkout the policy file:

   ```
   # pmpolicy checkout -d /tmp/example
   ```

2. Copy `example` to the `checkout` directory and rename to `pm.conf`.

   ```
   cp /opt/quest/qpm4u/examples/exampleX.conf /tmp/example/policy_
   pmpolicy/pm.conf
   ```

   where `X` in `exampleX.conf` is 1, 2, 3,...10.

3. Edit the configuration file and change the user name to a user name on your machine.

   ```
   # vi /tmp/example/policy_pmpolicy/pm.conf
   ```

4. Commit the changes and enter a commit log message:

```
# pmpolicy commit -d /tmp/example
** Validate options                                                    [ OK
]
** Commit copy in directory:/tmp/example/policy_pmpolicy

   ** Check directory                                                  [ OK
]
   ** Perform syntax check                                             [ OK
]
   ** Verify files to commit                                           [ OK
]
   Please enter the commit log message: Changed user name
   ** Commit change from working copy                                  [ OK
]
   ** Committed revision 4
```

5. Run a command using pmrun using the user name you specified. For example:

```
$ pmrun ls -l /tmp
```

Example 1: Basics

When you use pmrun to run a command, pmmasterd starts up and looks in the Privilege Manager for Unix configuration file for the conditions under which it should accept or reject the request.

The following configuration file fragment allows *Dan* to run programs as root:

```
if(user=="dan")
     {  runuser="root";
     accept;
 }
```

Type this fragment into the /etc/opt/quest/qpm4u/policy/pm.conf file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" with your own user name in quotes.

The syntax of the configuration language is similar to the C programming language:

- Each statement ends with a ; (semicolon)
- = (single equals) assigns values to variables
- == (double equals) compares values for equality
- ( ) (parentheses) enclose the conditional expressions in an if statement
- { } (braces) group statements together

- " " (double quotes) enclose strings
- White space, tab stops, or indentation are ignored

In the example above, the braces { } group the two statements that run if the conditions in the if statement are met. The accept statement causes pmmasterd to accept the request, and asks pmlocald to run whatever command Dan requests as root.

Use the pmcheck program to check the example for errors. pmcheck gives you a line number and brief description for each error found.

Note that pmcheck assumes that the configuration file exists in /etc/opt/quest/qpm4u/policy/pm.conf unless you specify otherwise on the command line with a -f filename argument.

For example, if pmcheck finds a syntax error on line 2 of the configuration file, it prints out a message similar to the following:

```
% pmcheck Version 6.0.0 (003) licensed until Thu Nov 1 06:00:00 2012 Parse error in
"/etc/opt/quest/qpm4u/policy/pm.conf", line 1: syntax error near ';' File
/etc/opt/quest/qpm4u/policy/pm.conf contains 1 error.
```

If pmcheck finds no errors, it displays a message similar to this:

```
% pmcheck
Version 6.0.0 (003) licensed until Thu Nov 1 06:00:00 2012

File /etc/opt/quest/qpm4u/policy/pm.conf contains 0 errors.
```

Try running a few more commands, such as date, hostname, and your favorite shell (such as, csh, sh, or ksh) by preceding the command with pmrun. For example:

```
# pmrun date
```

Example 2: Accept or reject requests

By default, pmmasterd rejects all requests. It only accepts requests if it reaches an accept statement after the appropriate conditions are met in the configuration file. When pmmasterd rejects a request, it does not run the requested program and it sends the user an explanatory message.

pmmasterd can also reject commands explicitly. The following fragment rejects Dan's request to run commands outside of regular office hours:

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

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

```
if(user=="dan") {
     # Explicitly disallow commands run outside of
     #regular office hours
     if(dayname=="Sat" || dayname=="Sun" ||
          !timebetween(800,1700))
          reject;
     runuser="root";
     accept;
 }
```

Once it reaches a `reject` statement, `pmmasterd` reads no further statements; the request ends as soon as it is rejected. Note that no braces { } enclose the reject statement, since it is the only statement that occurs inside the inner `if` statement. Note also the use of the || ("or") and ! ("not") operators in the `if` statement which translates as "if the current day is Saturday or Sunday, or if the current time is not between 8:00 a.m. and 5:00 p.m., then reject the request."

Type this fragment into the `/etc/opt/quest/qpm4u/policy/pm.conf` file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" with your own user name in quotes. Check the configuration file for errors with `pmcheck`. Then try to run commands with `pmrun`. For more information about using `pmcheck`, see Example 1: Basics on page 124.

Try changing the times specified to `timebetween`, to cause requests to be accepted or rejected.

Example 3: Command constraints

This configuration file fragment restricts *Dan* to running only certain programs (`ls`, `hostname`, or `kill`) as `root`.

Type this fragment into the `/etc/opt/quest/qpm4u/policy/pm.conf` file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" with your own user name in quotes.

```
if (user=="dan")
     if(command=="ls" || command=="hostname" ||
          command=="kill") {
     { runuser="root";
          accept;
     }
```

Check the configuration file for errors with `pmcheck`. For more information about using `pmcheck`, see Example 1: Basics on page 124. Try to run one of the programs permitted, then try something that will be rejected, such as:

```
pmrun mail
```

Example 4: Lists

Rather than entering individual commands as in Example 3, you can use list variables as shown below. Note the use of the && ("and") operator in the if statement.

This simple fragment allows users *Dan* and *Robyn* to run certain commands as root. Type this fragment into the /etc/opt/quest/qpm4u/policy/pm.conf file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" and "robyn" with users from your own site.

```
adminusers={"dan",  "robyn"};
 adminprogs={"ls",  "hostname",  "kill"};

 if(user  in  adminusers  &&  command  in  adminprogs)  {
      runuser="root";
      accept;
 }
```

Check the configuration file for errors with pmcheck. Run different commands with pmrun to see which ones are accepted, and which are rejected. Try logging in as one of the users who is not listed in adminusers. Then, try running a command as that user to see if Privilege Manager for Unix rejects the request. List variables are useful in tidying up policy fragments, especially if the information in a list is used more than once.

Example 5: I/O logging, event logging, and replay

The configuration file fragment below permits admin users *Dan* and *Robyn* to run certain commands as root. If the user requests csh or ksh, the input and output from these commands is logged. Privilege Manager for Unix also logs events, whether a request was accepted or rejected, and when a job finishes.

In this example, the input/output is logged to a file in /var/adm with a filename such as pm.dan.ksh.a05998, which you can examine later using pmreplay. The name of the I/O log is a unique temporary filename generated by the mktemp function.

Type this fragment into the /etc/opt/quest/qpm4u/policy/pm.conf file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" and "robyn" with users from your site.

```
adminusers = {"dan", "robyn"};
adminprogs = {"ls", "hostname", "kill", "csh", "ksh", "pmreplay"};

if (user in adminusers){
   runuser="root";
   if (command in {"csh", "ksh"})
      { iolog=mktemp("/var/adm/pm." + user + "."
         + command + ".XXXXXX");
      iolog_opmax=10000;
         print("This request will be logged in:", iolog);
      }
accept;
}
```

Check the configuration file for errors with `pmcheck`. For more information about using `pmcheck`, see Example 1: Basics on page 124.

Try running `csh` or `ksh` with `pmrun`, and typing a few commands in the shell. Exit from the shell, find the I/O log file in `/var/adm`, and replay the session with `pmreplay`.

Privilege Manager for Unix sets the permissions on the I/O log file so that only `root` can read the file. That way, no other user can examine the contents of the log files. You must be logged in as `root` to use `pmreplay` on these files. Of course, you can use `pmrun` to run a `csh` or `ksh` as `root`, and then run `pmreplay`. Or you can add `pmreplay` to the list of `adminprogs`, and then use `pmrun` to run it directly.

Note that `pmreplay` can detect whether a log file has changed. See pmreplay on page 447 for more information on running `pmreplay` interactively and non-interactively.

As `root`, run `pmreplay`, giving the name of the log file printed to the screen as an argument. For example, if the log filename is `/var/adm/pm.dan.ksh.a05998`, enter:

```
pmreplay /var/adm/pm.dan.ksh.a05998
```

You will see something similar to this:

```
================================================================
Log File : ./pm.dan.ksh.a05998
Date : 2008/02/25
Time : 12:00:00
Client : dan@sala.companyname.com
Agent : dan@sala.companyname.com
Command : ksh
Type '?' or 'h' for help
=================================================================
```

Use these commands to navigate through the log file:

Table 17: Log navigation commands

Control	Description
g	Go to start
G	Go to end

Control	Description
p	Pause/resume replay in slide-show mode
q	Quit
r	Redraw from start
s	skip to next time marker
t	Display time stamp
u	undo
v	Dump variables
[Space] bar	Go to next input (usually a single character)
[Enter]	Next new line
[Backspace]	Backup to last position
/<re>[Enter]	Search for a regular expression
Repeat last search	

Make your way through the log file by pressing the [Space] bar (next input character), the [Enter] or [Newline] key, or the s character which shows you what happened each time interval. You can backup through the log file by pressing the [Backspace] key. You can quickly go the start or end of the log file with g or G, respectively.

Display the time of an action at any point in the log file with t, redraw the log file with r, and undo your last action with u. You can also display all the Privilege Manager for Unix variables which were in use at the time the log file was created with v. Use q or Q to quit pmreplay.

You must run the pmreplay command as root because the log files created are readable only by root; however, pmreplay is itself a good candidate for a program to run through Privilege Manager for Unix. Note, in the following example, pmreplay is listed as one of the commands that Privilege Manager for Unix accepts.

Event logging is controlled by eventlog, which specifies the name of the file in which events ("accept", "reject", "finish") are logged. The default is /var/opt/quest/qpm4u/pmevents.db. If you do not want to use the default, see Local logging on page 152 for details.

You can encrypt the contents of the event log. See Event logging on page 153 for details.

To view the event log, use the pmlog command. Although pmlog prints all entries in the file by default, you can restrict it to print only certain entries. For example, to print only those events which occurred after Feb 5, 2012, enter:

```
pmlog -c'date=="2012/2/5"'
```

To print out all the variables stored with each entry, enter:

```
pmlog -v | more
```

The above command line pipes the voluminous output using more for easier viewing. You can also specify the output format and set the output for all event types.

Example 6: More complex policies

The fragment below extends the previous example by rejecting requests from *Dan* if they are made outside regular office hours, defined as 8:00 a.m. to 5:00 p.m., Monday through Friday. A message explaining the rejection is printed to Dan’s screen if this occurs.

Type the following code fragment into the /etc/opt/quest/qpm4u/policy/pm.conf file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" and "robyn" with users from your site (in quotes). Check the configuration file for errors using pmcheck. For more information about using pmcheck, see Example 1: Basics on page 124.

```
adminusers={"dan", "robyn"};
 adminprogs={"ls", "hostname", "kill", "csh", "ksh",
 "pmreplay"};

 if(user in adminusers && command in adminprogs)
      { runuser="root";
           if(command in {"csh", "ksh"}) {
                { iolog=mktemp("/var/adm/pm." + user + "."+ command
                     +".XXXXXX");
                     print("This command will be logged to:", iolog);
                }
                if(user=="dan" &&
                     (!timebetween(800,1700) || dayname in {"Sat", "Sun"}))
                {
                     print("Sorry, you can't use that command outside office
hours.");
                          reject;
                }
      accept;
      }
```

Try running a few commands with pmrun. Change the parameters for timebetween to exclude the current time, and run one of the permitted commands. Privilege Manager for Unix should reject the request and print the message to your screen. You should only be able to run the permitted commands during the specified time period. Try running pmreplay to replay some of the logged csh or ksh sessions.

Example 7: Use variables to store constraints

Similar to Example 6, the fragment below defines a variable to store a set of constraints (in this case, office hours) which may be used more than once in the configuration file. This saves you from typing the constraints each time you need to refer to them.

In the following example, there are two policies which depend on office hours. The first policy rejects *Dan*'s requests if they are made outside office hours. The second policy requires *Robyn* to type in her password if she makes a request outside regular office hours. Note that `officehours` is set to "true" if the time of the request falls between 8:00 a.m. and 5:00 p.m., Monday to Friday. It is "false" if it is not in that time frame.

```
officehours = timebetween(800, 1700) &&
     dayname !in {"Sat", "Sun"};
 adminusers={"dan", "robyn"};
 adminprogs={"ls", "hostname", "kill", "csh", "ksh", "pmreplay"};
 if(user in adminusers && command in adminprogs)
     { runuser="root";
          if(command in {"csh", "ksh"})
               { iolog=mktemp("/var/adm/pm." + user + "."
                         + command + ".XXXXXX");
                    print("The command will be logged in:", iolog);
     }
 # Note how compact the following fragments are compared to
 # example6.conf, referring to the "officehours" variable.
     if(user=="dan" && !officehours)
          { print ("Sorry, you can't do that outside office hours.");
               reject;
          }
               if(user=="robyn" && !officehours)
                    if(!getuserpasswd(user))
                         reject;
               accept;
          }
```

Type this fragment into the `/etc/opt/quest/qpm4u/policy/pm.conf` file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" and "robyn" with users from your site. Check the configuration file for errors with `pmcheck`. Then try to run commands with `pmrun`. For more information about using `pmcheck`, see Example 1: Basics on page 124.

Example 8: Control the run-time environment

This example demonstrates how you can set up a particular job's run-time operating environment with Privilege Manager for Unix. Although the policy fragments shown below are arbitrary, you can use similar fragments to implement your own policies.

Type the following fragment into the `/etc/opt/quest/qpm4u/policy/pm.conf` file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan" and "robyn" with users from your site.

Do not type in the line numbers.

```
1 # Run-time example configuration file
 2      adminusers={"dan", "robyn"};
 3      adminprogs={"ls", "hostname", "kill", "csh", "ksh", "echo"};
 4      if(user in adminusers && command in adminprogs) {
 5          if(!(cwd=="/usr" || glob("/usr/*", cwd))
 6              runcwd="/tmp";
 7          if(argc > 2)
 8              runargv=range(argv, 0, 2);
 9      runuser="root";
 10      rungroup="bin";
 11      if(command!="hostname")
 12          runhost=submithost;
 13      keepenv("TERM", "DISPLAY", "HOME", "TZ", "PWD", "WINDOWID", "COLUMNS",
"LINES");
 14      setenv("PATH", "/usr/ucb:/bin:/usr/bin:/usr/local/bin:/usr/bin/X11:" +
 15      "/usr/X11/bin:/usr/etc:/etc:/usr/local/etc:/usr/sbin");
 16      safeshells={"/bin/sh", "/bin/csh", "/bin/ksh"};
 17      if(getenv("SHELL") in safeshells)
 18          setenv("SHELL", getenv("SHELL"));
 19      else
 20          setenv("SHELL", "/bin/sh");
 21      runumask=022;
 22      runnice=-4;
 23      accept;
 24      }
```

The following describes the results of this example:

- **Lines 5, 6**

  These lines designate in which directory the job will run. Line 5 checks the current working directory: if the `cwd` variable is `/usr` or if it glob-matches "/usr/*", the request will run under that directory. If not, the request will run in `/tmp`.

- **Lines 7, 8**

  In this example, no more than two arguments are allowed to be specified to the requested command. The range function in line 8 returns all arguments and only the first three elements of the `argv` list (element 0, which is the command name; element 1, the first argument; and element 2, the second argument).

- **Line 9**

  This line causes the request to run as `root`.

- **Line 10**

  This line causes the request to run as the `bin` group.

- **Line 11, 12**

  These lines specify that if the command is not `hostname`, run it on the machine from which the request was submitted. If the command is `hostname`, run it on whatever machine the user wishes. (By default, it will run on the machine from which the request was submitted; you can change this using the -h argument to pmrun.)

- **Line 13**

  First, line 13 deletes all environment variables, except those specified in the keepenv list. Since you can use environment variables to exploit security holes in UNIX programs and shell scripts, be careful when specifying the environment variables for a request.

- **Line 14**

  This line sets the PATH variable explicitly to include only safe directories. Note the use of the + operator to concatenate the values assigned to the PATH variable; + splits the values over two lines to avoid ugly end-of-line wrapping.

- **Line 15-19**

  This fragment ensures that the SHELL variable is only set to a safe value. If the existing SHELL variable is already set to one of the values defined as "safe" in safeshells, then that value is used. If not, then the SHELL is set to /bin/sh.

  Note that getenv reads from the env variable; setenv and keepenv write to the runenv variable.

- **Line 20**

  This line sets the command's umask value to 022: data files created by the command will have rw-r--r-- permissions, and directories will have rwxr-xr-x permissions. Since the command will run under the root account, root will own the files.

  Specify a leading zero when typing in umask values so they are interpreted as octal numbers.

- **Line 21**

  The command will run with a nice value of -4, which gives it a high priority relative to other jobs on the system.

- **Line 22**

  After setting up the job's environment, the request is accepted and the job is run.

Check the configuration file for errors with pmcheck. For more information about using pmcheck, see Example 1: Basics on page 124.

Try running your favorite shell, for example:

```
# pmrun csh
```

In the shell, you can then enter env to list the environment variables, pwd to print the working directory in which your request ran, or umask to display the umask value.

Example 9: Switch and case statements

The following example illustrates how you can use the switch and case statements to implement complex policies. In this case, different users act as system administrators on different days of the week.

Type this fragment into the /etc/opt/quest/qpm4u/policy/pm.conf file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan", "robyn" and "cory" with users from your site.

```
adminprogs={"ls", "hostname", "kill", "csh", "ksh", "echo"};
 if(command in adminprogs) {
 switch (dayname) {
      case "Mon": true;
      case "Wed": true;
      case "Fri": adminusers={"dan", "robyn"};
           break;
      case "Tue": true;
      case "Thu": adminusers={"robyn", "cory"};
           break;
      default: adminusers={};
 }
 if (user in adminusers) {
      runuser="root"
      accept;
 }
```

When entering a switch statement, execution immediately jumps to the first case statement that matches the argument to switch (in this case, dayname). Execution proceeds from that point until a break statement or the end of the switch is reached. When a break statement is reached, execution jumps immediately to the end of the switch. If no case matches the argument to switch, execution jumps to the default statement.

Once execution has jumped to a case statement, it is unaffected by subsequent case statements. Only a break causes execution to jump to the end of the switch statement. If you omit a break, execution falls through to the next case statement.

Check the configuration file for errors with pmcheck. For more information about using pmcheck, see Example 1: Basics on page 124.

Log in as one of the adminusers to see if you can run requests with pmrun (it will depend on the current day). See switch on page 308 for further details.

Example 10: Menus

This example shows you how to present the commands a user may access as root in a menu by implementing a menu system with four choices. If the user selects the first menu item, he is asked to correctly type in a password before Privilege Manager for Unix runs the adduser program. If the user selects menu items **b**, **c** or **d**, Privilege Manager for Unix runs the backup, file ownership or line printer administration programs.

If the user’s request is accepted and completes, Privilege Manager for Unix prints messages to the user’s screen specifying the requested command and user under which the command will run. If the user makes an invalid menu choice, Privilege Manager for Unix prints a warning message and rejects the request.

Type the following code fragment into the `/etc/opt/quest/qpm4u/policy/pm.conf` file, or copy it from the examples directory in the Privilege Manager for Unix distribution directory. Replace "dan", "robyn", and "cory" with users from your site.

```
if(command=="adminmenu") {
     print("=========  Admin  Menu  =========");
     print("a)  Add  users");
     print("b)  Start  a  backup");
     print("c)  Change  ownership  of  a  file");
     print("d)  Fix  line  printer  queues");
     choice=input("Please  choose  one:  ");

     switch(choice)  {
          case  "a":
          #  Reject  the  request  if  the  password  "123456"  is  not  entered
          #  correctly.  The  user  is  allowed  only  two  chances  to  type
          #  the  password  correctly.  The  encrypted  version  of  the
          #  password  seen  here  was  generated  using  "pmpasswd".
          #  If  you  store  encrypted  passwords  in  your  config  file,
          #  make  sure  you  turn  off  read  permission  on  the  file  so
          #  that  no  one  can  use  a  password  cracking  program  to
          #  guess  them.
               if(!getstringpasswd("m9xxg7B4.v8Ck",
                         "Type  in  the  adduser  password:  ",  2))
                    reject;
               runcommand="/usr/local/bin/adduser";
               runuser="root";
               break;
          case  "b":
               runcommand="/usr/local/bin/dobackup";
               runuser="backup";
               break;
          case  "c":
               runcommand="/etc/chown";
               runuser="root";
               break;
          case  "d":
               runcommand="/usr/lib/lpadmin";
               runuser="root";
               break;
          default:
               printf("\"%s\"  was  not  a  valid  choice.Sorry.\n",  choice);
          reject;
 }
     print("**  Command  to  be  run  :",  runcommand);
     print("**  User  to  run  command  as  :",  runuser);
     accept;
     }
```

Check the configuration file for errors with `pmcheck`. For more information about using `pmcheck`, see Example 1: Basics on page 124.

To display the menu, enter:

```
# pmrun adminmenu
```

Select the first menu item. When Privilege Manager for Unix asks you for the password, type "123456". Privilege Manager for Unix accepts the request and attempts to run the job.

Since the commands in this example probably do not exist on your system, the job will fail. Try substituting your own commands in each of the menu items, and test the fragment again.

Use the `while` loop

To create more complex statements in the configuration file, you can use a `while` loop construction. For example:

```
while (expression) {
      <script statements>;
 }
```

In the following example, the scripting language searches the argument list of the command for the argument `root`. This is useful for allowing access to the `passwd` command.

```
count=1;
 while( count < argc ) {
      if( argv[count] == "root" )
      reject;
      count=count+1;
 }
```

See while on page 309 for further details.

Use parallel lists

You can use two lists in parallel, with information from element X of one list relating to information from element Y in the other list. In this example, the command name is related to its full pathname. You can incorporate this technique when you require certain users to type in a password that is different for each user.

```
okcommands={"ls", "sort", "pmreplay"};
     okpaths={"/bin/ls", "/bin/sort", "/usr/etc/pmreplay"};
     i=search(okcommands,command);

 if(i==-1) {
          print("Invalid Command");
          reject;
     } else {
          runcommand=okpaths[i];
          accept;
 }
```

If the `search` fails (is set to `-1`), it rejects the request. Otherwise, the `runcommand` variable is set to the permitted path and command, and it accepts the request.

Best practice policy guidelines

One Identity recommends that you keep the following guidelines in mind when building your configuration file. Give careful thought to the environment in which the job will run.

- The directory in which the job will run should be controlled by the `runcwd` variable.

  By default, jobs run in the same directory from which they are submitted.
- The environment variables that you consider "safe":
  - Use the `keepenv` function to keep the "safe" environment variables and remove all others.
  - Variables such as `TERM`, `DISPLAY`, and `TZ` are useful to keep; the job can access and make use of their values.
  - Variables such as `SHELL`, `PATH`, `IFS` or `LD_LIBRARY_PATH` can have unspecified effects if set improperly. To avoid problems, use `keepenv` to delete these variables; use `setenv` to set them to safe values.
- Explicitly set the environment variables:
  - Use the `setenv` function to set these variables.
  - Always set the `PATH` variable explicitly. Running shell scripts or programs with a non-standard `PATH` can allow users to substitute their own -- possibly malevolent -- programs to run in place of the ones that you intended. Well-written shell scripts set `PATH` themselves. Set it explicitly in the Privilege Manager for Unix policy.
- The machine on which the job will run should be controlled by the `runhost` variable.

  By default, jobs run on the machine from which they are submitted. To run a job on a different machine, use the `-h` option of the `pmrun` command. If you are concerned about where the job will run, explicitly set the `runhost` variable. See pmrun on page 450 for details.

- The user ID under which the job will run:
  - Users typically use Privilege Manager for Unix to run jobs as `root`, but may specify any account.
  - The `runuser` variable contains the name of the user under which the job will run.
  - If you do not set `runuser` explicitly, the job will run under the user ID that originally submitted it. This may be advantageous if you are using Privilege Manager for Unix as a substitute for `ssh` to control who can log into a particular machine.
- The groups in which the job will run:
  - The `rungroup` variable stores the name of the job's primary group, while the `rungroups` variable stores a complete list of all groups to which the job belongs.
  - The default is all groups to which the user submitting the job belongs.
- The command that will be run:
  - The `runcommand` variable stores the name of the command that will be run.
  - If it is not a full pathname, Privilege Manager for Unix searches the `PATH` variable for the job to find the command to run (a good reason to explicitly set `PATH` to something safe).
  - You can have Privilege Manager for Unix run a different command from the one asked for by the user, by setting the `runcommand` variable. Example 10: Menus on page 134 displays a menu of administrative programs in response to a user executing a `pmrun adminmenu` command. The user then selects one to run.

  When you set `runcommand`, Privilege Manager for Unix automatically sets the `runargv[0]` variable to the base name of the `runcommand` value. UNIX shells do the same thing when you run a command.
- The arguments for the request:
  - The `argv` list variable stores a list of user requested command names and arguments. `argv[0]` is the command name, `argv[1]` is the first argument, and so on.
  - By changing the `runargv` variable, you can set the arguments to the command. This allows you to limit or add to the arguments requested by the user.

  If the command is a shell script, or if you wish to cause the command to be run through a shell, be careful with the argument list. By adding semicolons into an argument, you can completely change the behavior of a command. For example, if you run this command:

  ```
  csh -c 'ls /tmp'
  ```

  which lists the files in `/tmp`, a malicious user might type:

  ```
  csh -c 'ls /tmp;rm /*'
  ```

Ensure that your programs and/or scripts can handle strange arguments safely.

- The type of logging done for the request:
  - Set the `iolog` variable to a unique pathname; later replay the session using `pmreplay`.
  - A log noting that the request was either accepted, rejected, or completed is stored by default in `/var/opt/quset/qpm4u/pmevents.db`. For more information about logging, see Event logging on page 153.

Multiple configuration files and read-only variables

You can split up the configuration file into separate parts to reduce clutter. Use the `include` statement to hand off control to a subsidiary configuration file. While in the subsidiary configuration file, if an `accept` or `reject` occurs, control never returns to the main file. However, if no `accept` or `reject` occurs, once the end of the subsidiary configuration file has been reached, control returns to the parent file for further processing. Control resumes immediately after the `include` statement.

When handing off control to a subsidiary configuration file whose contents are controlled by a questionable person, it may be desirable to fix certain Privilege Manager for Unix variable values so that they cannot be changed by the subsidiary file. Use the `readonly` statement for this purpose.

For example, you may have an Oracle database administrator, who needs to administer certain Oracle programs. Each of those programs is to run as the "oracle" user. You would like the database administrator to be able to grant or deny access to these programs and this account without your involvement, but you certainly do not want to give this person power over non-Oracle parts of the system.

The following configuration file fragment hands off control to a subsidiary configuration file called `pmoracle.conf`, and ensures that if an `accept` is done within this file, the job being accepted can only run as the oracle user.

```
oraclecmds = {"oradmin", "oraprint", "orainstall"};
 if(command in oraclecmds){
      runuser = "oracle";
      readonly {"runuser"};
      include "/etc/pmoracle.conf";
      reject;
 }
```

The argument passed to `readonly` is a list of variable names (here, we have only specified one variable).

Also, the `reject` statement after the `include` ensures that if the `pmoracle.conf` configuration file does not `accept` or `reject` the job, this fragment will explicitly `reject` it. Of course, if the `pmoracle.conf` file accepts the job, the `reject` in this fragment will never be reached.

You can give the database administrator access to edit the pmoracle.conf file by entering "pmrun pmoracle.conf" if you include the following fragment. It calls the secure pmvi text editor (supplied with Privilege Manager for Unix), which allows the user to edit the file whose name is given on the command line, but will not allow the user to read or write any other file, nor to run any subprocesses from within the editor.

The following example sets:

- the command to be run (/opt/quest/bin/pmvi)
- its arguments ("pmvi /etc/pmoracle.conf")
- the user it will run as ("root")
- and accepts the request

```
if(command == "pmoracle.conf" && user == "dba_login_name")
     {
          runcommand = "/opt/quest/bin/pmvi";
          runargv = split("pmvi /etc/pmoracle.conf");
          runuser = "root";
          accept;
 }
```

Mail

You may use the configuration file to send mail messages when certain actions occur. The following fragment sends mail to root whenever the adduser program runs:

```
if(command=="adduser") {
     system("mail root",
          "pm: adduser was run as root by " + user + "\n");
 }
```

Environmental variables

You can use environment variables to turn on or off special features of Privilege Manager for Unix configuration files. In the following example, the list of Privilege Manager for Unix variables is printed to the user's screen if the DEBUG environment variable is set to "yes". This is useful when debugging a configuration file. Simply set the DEBUG variable to "yes" in your shell, then run pmrun. Privilege Manager for Unix notices the DEBUG variable, and calls the printvars function.

```
if(getenv("DEBUG")=="yes")
   printvars();
```

NIS netgroups

If you have a large site where you add and remove hosts frequently, you may already be using `netgroups` to associate a group name with a set of hosts. The Privilege Manager for Unix `innetgroup` function inquires if a named host is a member of a named netgroup.

For example, you can reject requests originating from any machine that is not in the netgroup `myhosts` as follows:

```
if(!innetgroup("myhosts", host))
   reject;
```

Specify trusted hosts

You can reject all requests that do not originate from your domain; that is, specify only the hosts that you trust to issue requests by using the following:

```
if(submithost !in {"*.quest.com"})
   reject;
```

Configuring firewalls

When the agent and policy server are on different sides of a firewall, Privilege Manager for Unix needs a number of ports to be kept open. By default, Privilege Manager for Unix can use ports in the 600 to 31024 range, but when using a firewall, you may want to limit the ports that can be used. See Restricting port numbers for command responses on page 142 for more information.

This section describes

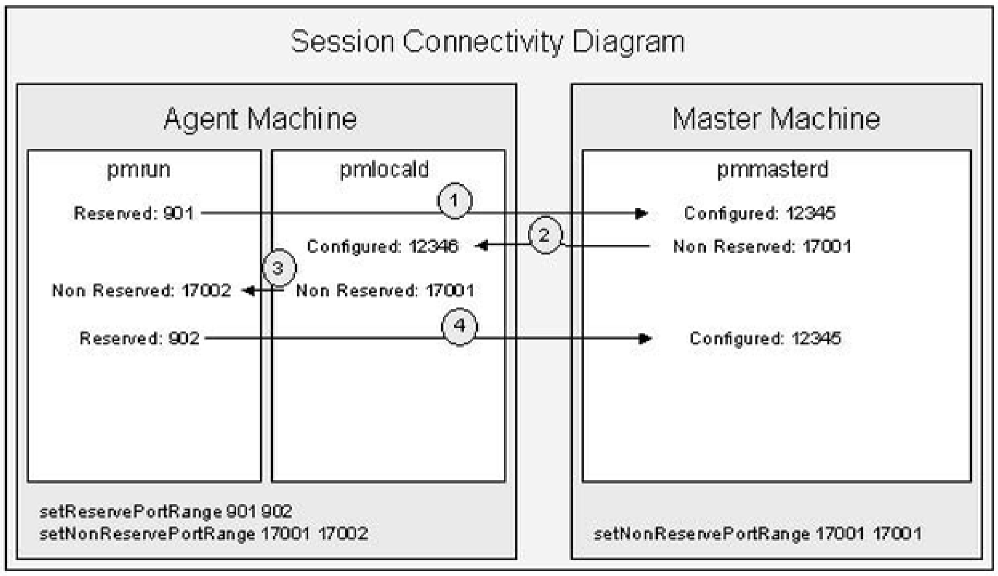
- how Privilege Manager for Unix uses ports from both the reserved and non-reserved port ranges during a session
- how to configure Privilege Manager for Unix over a firewall and, optionally, Network Address Translation (NAT)

Privilege Manager for Unix port usage

For each Privilege Manager for Unix session, the client (`pmrun`) and agent (`pmlocald`) use one port from both the reserved and non-reserved ranges. The policy server (`pmmasterd`) uses one port from its non-reserved range. Each agent can use the same port ranges as they are on separate machines and need only be large enough to support the maximum number of concurrent sessions on that agent. On the other hand, the policy server needs a port range large enough to support all sessions across all agents (minimum of one non-reserved port per session).

This diagram shows the minimum port ranges required for a single Privilege Manager for Unix session:

Figure 7: Privilege Manager for Unix port usage



Connection 4 is used only to send back the exit status if I/O logging is not enabled.

Restricting port numbers for command responses

If commands involve communication through a firewall, you can restrict the TCP/IP port numbers on which responses to `pmrun` commands are returned.

One Identity recommends that you assign a minimum of six ports to Privilege Manager for Unix in the reserved ports range (600 to 1023) and twice that number of ports in the non-reserved ports range (1024 to 65535). The more agents you have, the more ports you need.

To set the reserved port range

1. Add the following line to the `/etc/opt/quest/qpm4u/pm.settings` file:

   ```
   setreserveportrange <lowportnumber> <highportnumber>
   ```

   where `<lowportnumber>` is first port in the range and `<highportnumber>` is the last port in the range.

`<lowportnumber>` and `<highportnumber>` must be port numbers between 600 and 1023. For example:

```
setreserveportrange 600 612
```

To set the non-reserved port range

1. Add the following line to the `/etc/opt/quest/qpm4u/pm.settings` file:

   ```
   setnonreserveportrange <lowportnumber> <highportnumber>
   ```

   `<lowportnumber>` and `<highportnumber>` must be port numbers between 1024 and 65535. For example:

   ```
   setnonreserveportrange 31000 65535
   ```

See PM settings variables on page 286 for more information about modifying the Privilege Manager for Unix configuration settings.

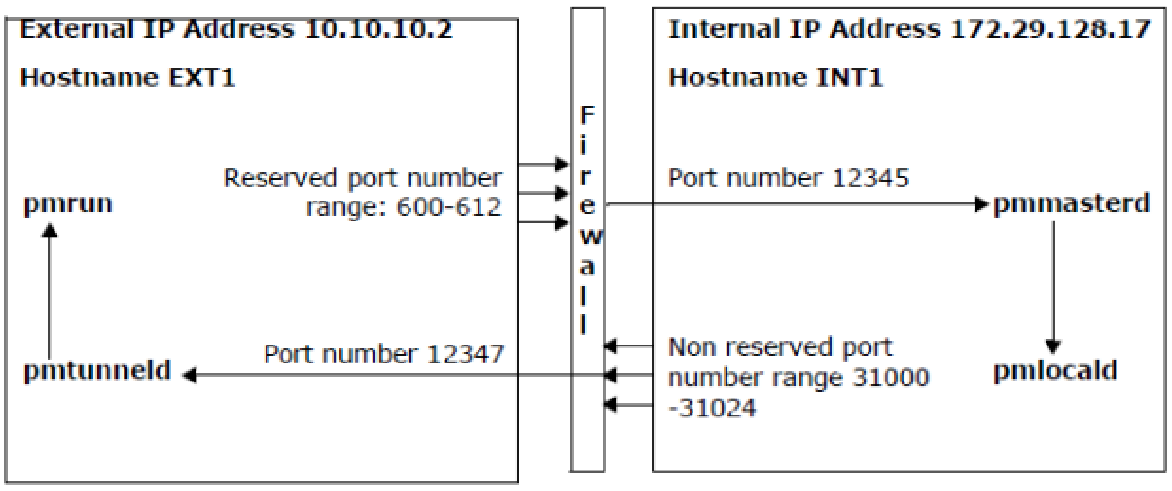
Configuring pmtunneld

`pmtunneld` adds an additional layer of security by acting as a proxy for `pmrun`. Communication sent from `pmlocald` is transmitted using port number 12347, by default, and received by `pmtunneld`. `pmtunneld` then transmits the data to `pmrun`.

In the following example, the firewall is configured to allow the following connections:

- One incoming connection from external host (EXT1) reserved port range (600 - 612) to internal host (INT1) port 12345.
- One outgoing connection from internal host (INT1) non-reserved port range (31000 - 31024) to external host (EXT1) port 12347.

Figure 8: pmtunneld configuration


External IP Address 10.10.10.2
Hostname EXT1
pmrun
Reserved port number range: 600-612
pmtunneld
Port number 12347
Firewall
Internal IP Address 172.29.128.17
Hostname INT1
Port number 12345
pmmasterd
Non reserved port number range 31000 -31024
pmlocald

To configure pmtunneld, in the /etc/opt/quest/qpm4u/pm.settings file, specify the hosts that require pmlocald to use a fixed port when communicating with pmrun and the fixed port that pmlocald uses when communicating with pmrun.

In this example, you configure the external host (EXT1) by adding these lines to the /etc/opt/quest/qpm4u/pm.settings file:

```
tunnelport 12347
pmtunneldenabled yes
```

In this example, you configure the internal host (INT1) by adding these lines to the pm.settings file:

```
tunnelrunhosts EXT1
tunnelport 12347
```

Note that tunnelrunhosts can contain wild cards, such as, *.mydomain.com.

To allow commands to run on the external host, EXT1 in this example, create a firewall rule to allow pmmasterd to connect from the non-reserved port range to the pmlocald port on the external agent.

See PM settings variables on page 286 for more information about modifying the Privilege Manager for Unix configuration settings.

Configuring Network Address Translation (NAT)

To configure Privilege Manager for Unix to allow the use of Network Address Translation (NAT), you must add both the external and internal IP address of the firewall to tunnelrunhosts list in the /etc/opt/quest/qpm4u/pm.settings file.

See PM settings variables on page 286 for more information about modifying the Privilege Manager for Unix configuration settings.

Configuring Kerberos encryption

You can configure Privilege Manager for Unix to use Kerberos encryption to authenticate and to exchange encryption key information

To configure Privilege Manager for Unix to use Kerberos encryption, edit or insert the following line in the /etc/opt/quest/qpm4u/pm.settings file:

```
kerberos yes
```

Also, to use Kerberos with Privilege Manager for Unix, ensure that suitable Service Principal Names (SPNs) are registered. Using the generic host service-type, configure the SPNs like this:

```
host/sun17.quest.com
```

Substitute your own host names.

If the SPN has been registered using the fully qualified DNS name, you can abbreviate the SPNs to the service-type, such as:

```
host
```

Specify the service principal names using the `mprincipal` and `lprincipal` settings in the `pm.settings` file. For example, on an agent with a host name of `sun17.quest.com`, and a SPN registered as `db_serve1.quest.com`, specify:

```
mprincipal host
lprincipal host/db_server1.quest.com
```

You may need to modify these other settings according to your Kerberos configuration:

Table 18: Other Kerberos configuration settings

Kerberos Setting	Description
keytab	Location of the keytab file. Default: `/etc/opt/quest/vas/host.keytab`
krb5rchache	Location of the Kerberos cache. Default: `/var/tmp`
Location of the Kerberos configuration file. Default: `/etc/opt/quest/vas/vas.conf`	

See PM settings variables on page 286 for more information about modifying the Privilege Manager for Unix configuration settings.

Configuring certificates

You can enable configurable certification for use with Privilege Manager for Unix. Configurable certification is a method of proprietary certification based on the system hardware ID, MD5 checksums and DES encryption.

Use the `pmkey` command to generate and install certificates. For example, to generate a new certificate and put it into the specified file, enter:

```
# pmkey -a <filename>
```

To install the newly generated certificate from the specified file, enter:

```
# pmkey -i <filename>
```

Enable configurable certification

To enable configurable certification

1. Ensure that you have configured a Privilege Manager for Unix policy server and a Privilege Manager for Unix client.
2. Add the following statement to the `/etc/opt/quest/qpm4u/pm.settings` file on each host:

   ```
   certificates YES
   ```

3. To generate a key on the Privilege Manager for Unix policy server, enter:

   ```
   # pmkey –a <policy server filename>
   ```

   When prompted, enter a phrase or keyword.
4. To install the key on the Privilege Manager for Unix policy server, run

   ```
   # pmkey -i <policy server filename>
   ```

   You must enter the same filename in both the `-a` and `-i` commands shown above.
5. To generate a key on each Privilege Manager for Unix client, enter:

   ```
   # pmkey –a <client filename>
   ```

   When prompted, enter a phrase or keyword. Note: you must use the same phrase or keyword to generate the client and policy server certificates.
6. To install the key on the Privilege Manager for Unix client, run

   ```
   # pmkey -i <client filename>
   ```

   You must enter the same filename in both the `-a` and `-i` commands shown above.
7. Copy the key file you have created on each of the Privilege Manager for Unix clients to the Privilege Manager for Unix policy server.
8. Copy the key file you have created on the Privilege Manager for Unix policy server to the Privilege Manager for Unix client.

   The keys are located in `/etc/opt/quest/qpm4u/.qpm4u/.keyfiles/<key filename>`.

9. On the Privilege Manager for Unix policy server, enter:

   ```
   # pmkey -i <client filename>
   ```

10. On the Privilege Manager for Unix client, enter:

    ```
    # pmkey -i <policy server filename>
    ```

    Configurable certification is now enabled.

    By default, `pmkey` certifies the pass phrase when installing the keyfile for other hosts. If you do not want `pmkey` to certify the pass phrase when installing the keyfile for other hosts, use `-f` in the `pmkey -i` command, like this:

    ```
    # pmkey -i <keyfile> -f
    ```

Configuring alerts

Alerts enable you to specify commands that raise an alert if entered by a user, and the action you want Privilege Manager for Unix to take.

Use the `alertkeyaction` variable to specify the action Privilege Manager for Unix is to take when an alert is raised. The default action logs the alert and allows the command to continue.

Enter `alertkeysequence` in the policy as a list of regular expressions, like this:

```
alertkeysequence={"^rm.*", "/rm.*", ".*xterm"};
```

Other valid alert actions are:

- log
- reject
- or any valid string

For example:

```
if (user=="root")
{
   alertkeyaction="ignore";
}
   else if (user=="john")
{
   alertkeyaction="alert";
}
   else if (user=="dave")
{
```

```
   alertkeyaction="trace";
}
   else
{
   alertkeyaction="reject";
}
```

If an event raises an alert, Privilege Manager for Unix logs an `AlertRaised` event log. The `alertkeyaction` variable is also included in the log as part of the event.

If the `alertkeyaction` variable is set to `reject`, Privilege Manager for Unix cancels the command, terminates the user's session, and displays a rejection message.

If the `alertkeyaction` variable is not set to `reject`, Privilege Manager for Unix allows the command to run and logs it in the event log. The example shown above shows how you can enter different strings for different users. This enables you to use the `alertkeyaction` variable as a filter to search the event log for these events.

`alertkeyaction` logging is enabled even if `iologging` is disabled. If `iologging` is disabled, a new session is started with `pmmasterd` for each `alertraised` event.

By default, `alertraised` events are not displayed in `pmlog`. To view the `alertraised` event, use the `-l` parameter or the `-d` parameter. For example:

```
# pmlog -l
```

Alert events have the same unique ID as the Privilege Manager for Unix session from which they were generated. This enables you to identify alert events raised during a specific session.

Use `pmcheck` to check a given string against any expression defined in the `alertkeypatterns` list:

```
# pmcheck -a"<string>"<command>
```

For example,

```
# pmcheck -a "rm /etc/opt/quest/qpm4u/pm.settings" ksh
```

Configuring Pluggable Authentication Method (PAM)

Use `authenticate_pam` to define which users you want to authenticate by means of PAM (Pluggable Authentication Method) APIs.

The operating system has configuration files, usually called `/etc/pam.conf`, that specify which security databases to use to authenticate users, such as LDAP, Windows 2000 Active Directory, and various PKI implementations.

The service parameter identifies the name of the PAM service to use to authenticate users. The service parameter can be any valid service name configured in the PAM system configuration and defaults to "login".

For more information on how to configure PAM with Privilege Manager for Unix, consult the documentation for your platform.

Utilizing PAM authentication

Syntax

```
authenticate_pam (user,[<service>])
```

where `<service>` is the PAM service to use, such as `sshd`.

Examples

To utilize PAM authentication, add the following function to your policy file:

```
if ( user=="paul" && basename(command)=="useradd") {
     if (!authenticate_pam(user, "sshd")) { reject; }
     runuser="root";
     accept;
}
```

This function returns 0 to indicate failure and 1 to indicate success.

Related Function

`authenticate_pam_toclient`

Related Topics

authenticate_pam

Authenticate PAM to client

Syntax

```
authenticate_pam_toclient (user,[<service>])
```

where `<service>` is the PAM service to use, such as `sshd`.

Description

`authenticate_pam_toclient` causes `pmmasterd` to send a request to `pmrun` to perform the `authenticate_pam` command on the `pmrun` host.

This function is only available on platforms that have native support for PAM.

Example

To utilize PAM authentication, add the following function to your policy file:

```
if ( user=="paul" && basename(command)=="useradd") {
 if (!authenticate_pam_toclient(user, "sshd")) { reject; }
      runuser="root";
      accept;
 }
```

This function returns 0 to indicate failure and 1 to indicate success.

Related Function

`authenticate_pam`

Related Topics

authenticate_pam_toclient

Administering Log and Keystroke Files

Privilege Manager for Unix allows you to control what is logged, as well as when and where it is logged. To help you set up and use these log files, the topics in this section explore enabling and disabling logging, as well as how to specify the log file locations.

Privilege Manager for Unix includes three different types of logging; the first two are helpful for audit purposes:

- **keystroke logging**, also referred to as I/O logging

  Keystroke logs record the user's keystrokes and the terminal output of any sessions granted by Privilege Manager for Unix.
- **event logging**

  Event logs record the details of all requests to run privileged commands. The details include what command was requested, who made the request, when the request was sent, what host the request was submitted from, and whether the request was accepted or rejected.
- **error logging**

You can configure some aspects of the event and keystroke logging by means of the security policy on the policy servers. What you can configure and how you configure it depends on which type of security policy you are using on your policy server -- pmpolicy or sudo.

Related Topics

Security policy types

Controlling logs

The following variables are used to control the logging of program input and output through Privilege Manager for Unix.

Table 19: Logging variables

Variable	Explanation
iolog	If set to a filename, the `iolog` variable logs all of the information from the `logstdin`, `logstdout`, and `logstderr` variables to the specified filename.
logstderr	If set to `true`, the `logstderr` variable logs any error responses.
logstdin	If set to `true`, the `logstdin` variable logs all information coming in from standard input.
If set to `true`, the `logstdout` variable logs all information being displayed to standard output.	

For details about these logging variables, refer to Global output variables on page 243.

To log the input, output and error I/O streams from a request, set `logstdin`, `logstdout`, and `logstderr` to `true`. Set `iolog` to the name of the log file. After Privilege Manager for Unix completes the request, you can use the `pmreplay` command to replay the session that was logged.

You can limit the amount of data logged for each stream. This avoids filling up the I/O logs with large amounts of output from benign commands, such as when using `cat` or `tail` to display a large file. You can limit the I/O logging to the first `n` bytes of the output. For example, to log only the first 500 bytes of `stdout`, enter:

```
iolog_opmax=500;
```

The following example ensures that whenever you run the `adduser` program through Privilege Manager for Unix, it logs all input and output in the specified file:

```
if(command=="adduser")  {
      iolog="/var/log/iolog/"  +  user  +  mktemp("_XXXXXX");
      logstdin=true;
      logstdout=true;
      logstderr=true;
      runuser="root";
      accept;
 }
```

Local logging

The location of the error logs for the Privilege Manager for Unix components, `pmrun`, `pmlocald`, and `pmmasterd`, is specified using keywords in the `pm.settings` file. Enter the following to specify that you want the error logs written to the `/var/adm` directory:

```
pmlocaldlog /var/adm/pmlocald.log
pmmasterdlog /var/adm/pmmasterd.log
pmrunlog /var/adm/pmrun.log
```

Alternatively, you can enable UNIX syslog error logging in the `pm.settings` file, by specifying:

```
syslog YES
```

Use one of the following keywords to specify which syslog facility to use:

- LOG_KERN
- LOG_USER
- LOG_MAIL
- LOG_DAEMON
- LOG_AUTH (the default)
- LOG_LPR
- LOG_NEWS
- LOG_UUCP
- LOG_CRON
- LOG_LOCAL0 through LOG_LOCAL7

For example, to enable syslog error logging using the LOG_AUTH facility, enter in the `pm.settings` file:

```
syslog YES
facility LOG_AUTH
```

See PM settings variables on page 286 for more information about modifying the Privilege Manager for Unix configuration settings.

Event logging

Event logs are enabled by default for all requests sent to the Privilege Manager for Unix Policy Servers. The default location of the event log file is `/var/opt/quest/qpm4u/pmevents.db`.

When using the pmpolicy type, you can change the location of the event log, or disable event logging for a specific request by modifying the `eventlog` policy variable. For example, to disable event logging for all `pmlist` commands, add the following code to your security policy:

```
if (basename(command) == "pmlist") { eventlog=""; }
```

The following pmpolicy variables affect event log settings:

Table 20: Event logging policy variables

Variable	Data type	Description
eventlog	string	The name of the file in which events (acceptances, rejections, and completions) are logged. (Default is /var/opt/quest/qpm4u/pmevents.db.) This must be a full pathname starting with a / (slash). For example: `eventlog = "/var/logs/pmevents.db";` If the log file name you specify in the policy file cannot be opened, Privilege Manager for Unix automatically logs all events in the default log file. See also eventlog on page 248.
logomit	list	Specifies the names of variables to omit when logging to an event log (no default). Use this to reduce the amount of disk space used by event logs. See also logomit on page 254.
Specify a local variable to add to the event log. (Refer to Operators and expressions on page 186 for more information about export.)		

For example, enter the following to specify that you want to:

- record event log in /var/adm/pmevents.db
- not include the env and runenv variables in the logs

```
eventlog = "/var/adm/pmevents.db";
logomit = {"env","runenv"};
```

Keystroke (I/O) logging

Once your 30-day trial license has expired, One Identity requests that you obtain a Keystroke Logging license to remain in compliance. See Privilege Manager for Unix licensing on page 15 for details.

You can enable keystroke logging using the `iolog` variable. If this variable is not defined or is an empty string, keystroke logging is disabled. Otherwise, specify the full path to the keystroke log using `iolog` variable. See iolog on page 249 for details.

If you use the default profile-based policy, `iolog` is defined in the `profileBasedPolicy.conf` file as:

```
iolog=mktemp("/var/opt/quest/qpm4u/iolog/"
+ profile
+ "/"
+ user
+ "/"
+ basename(runcommand)
+ "_"
+ strftime("%Y%m%d_%H%M")
+ "_XXXXXX");
```

You can enable keystroke logging on a per profile basis by editing the `profile` and `shellprofile` files, and setting the `pf_keystrokelogging` variable to `true` or `false`.

The following variables affect keystroke log settings when using the pmpolicy type:

- iolog
- iolog_encrypt
- iolog_opmax
- iologhost
- logomit
- logstderr
- logstdin
- logstdout
- log_passwords

For details about these variables, refer to the Global output variables on page 243.

Keystroke (I/O) logging policy variables

You can control keystroke (I/O) logging behavior using the following policy variables.

Table 21: Keystroke logging policy variables

Variable	Data type	Description
iolog	string	The name of the file in which input, output, and error output is logged. This must be a full pathname starting with a / (slash). To avoid overwriting existing I/O log files, set the `iolog` variable with a `mktemp` function call.
iolog_encrypt	boolean	Enables encryption of I/O logs: To enable encryption, set: `iolog_encrypt = true;` Log files are encrypted with AES; view them with `pmreplay`.
iolog_errmax	integer	Limits the amount of text logged for `stderr` for each command.
iolog_opmax	integer	Limits the amount of text logged for `stdout` for each command. For example, if `iolog_opmax` is set to 500 and you enter: `cat filename1` it only logs the first 500 bytes of output produced by this command.
log_passwords	boolean	Specifies whether passwords are logged to the keystroke log. The default setting logs passwords. See log_passwords on page 253 for details.
logstderr	boolean	Specifies if error output is logged; default is "true".
logstdin	boolean	Specifies whether input is logged; default is "true".
logstdout	boolean	Specifies whether output is logged; default is "true".

All boolean values default to "true".

Example

```
iolog=mktemp(”/opt/quest/qpm4u/logs/”+”user”+”_”+basename(command)
   +”_XXXXXX”);
iolog_encrypt = true;
iolog_opmax = 500;
iolog_errmax = 200;
logstderr = false;
logstdin = true;
logstdout = true;
log_passwords = false;
```

For details about the keystroke logging variables, refer to Global output variables on page 243.

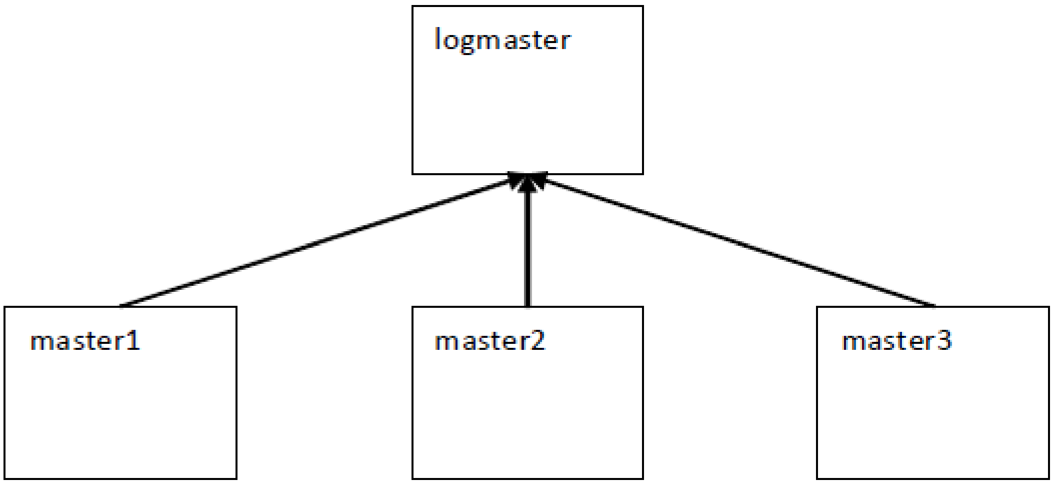
Central logging with Privilege Manager for Unix

Privilege Manager for Unix can configure central logging for I/O and event logs using the `iologhost` and `eventloghost` policy variables.

`pmmasterd` uses port number 12345 by default to communicate with the log server.

A host that is configured as a centralized log server must have the client's keyword added to the `pm.settings` file to specify which policy servers may forward their I/O and event log information to this log server.

Figure 9: Configuring central logging for I/O and event Logs



In this example, master1, master2, master3, and logmaster are all Privilege Manager for Unix policy servers (pmmasterd).

logmaster is configured as the centralized log host for I/O and event logs for master1, master2 and master3. To send I/O and event log information to logmaster, the policy must include the following statements:

```
iologhost = "logmaster";
eventloghost = "logmaster";
```

If for any reason (such as a system outage) the logs cannot be forwarded to the central logging host (logmaster in the above example), log files are stored locally on the authenticating policy server (master1, master2, or master3 in the above example). The location of the log files is specified by the tmplogdir policy variable, which defaults to var/opt/quest/qpm4u/iolog/queue.

The pm.settings file for logmaster must include the clients keyword. For example:

```
clients master1 master2 master3
```

Related Topics

PM settings variables

tmplogdir

Controlling log size with Privilege Manager for Unix

An effective strategy for controlling the size of the log file in Privilege Manager for Unix is to limit the amount of information sent to the logs. Instead of logging keystrokes for every command, you might construct a policy that only captures keystrokes for sensitive commands.

You can use policy variables to limit the information sent to the log files.

Table 22: Size-controlling logging variables

Variable	Data type	Description
iolog_encrypt	boolean	Enables I/O logs encryption; default is "true". Log files are encrypted with AES; view them with pmreplay.
iolog_errmax	integer	Limits the amount of text logged for stderr for each command.

Variable	Data type	Description
iolog_opmax	integer	Limits the amount of text logged for `stdout` for each command. For example, if `iolog_opmax` is set to 500 and you enter the following command: `cat filename1` it only logs the first 500 bytes of output produced by this command.
logomit	list	Specifies the names of variables to omit when logging to an event log (no default). Use this to reduce the amount of disk space used by event logs.
logstderr	boolean	Specifies if error output is logged; default is "true".
logstdin	boolean	Specifies whether input is logged; default is "true".
logstdout	boolean	Specifies whether output is logged; default is "true".

Viewing the log files using a web browser

If you are running Privilege Manager for Unix, you can view events using Management Console for Unix, which provides an intuitive web-based console for managing UNIX hosts.

Refer to the *One Identity Management Console for Unix Administration Guide* for details about using the mangement console.

Viewing the log files using command line tools

If you are not running Privilege Manager for Unix with Management Console for Unix, or if you prefer to use command line tools, you can list events and replay log files directly from the primary policy server using the `pmlogsearch`, `pmreplay`, and `pmremlog` commands.

`pmlogsearch`

`pmlogsearch` is a simple search utility based on common criteria. Run `pmlogsearch` on the primary server to query the logs on all servers in the policy group. `pmlogsearch` provides a summary report on events and keystroke logs matching at least one criteria. `pmlog` provides a more detailed report on events than `pmlogsearch`.

Hostnames may appear in the event logs and keystroke log files in either fully qualified format (`myhost.mycompany.com`) or in short name format (`myhost`), depending on how hostnames are resolved and the use of the short name setting in the `pm.settings` file. To ensure that either format is matched, use the short host name format with an asterisk wildcard (`myhost*`) when specifying a hostname search criteria.

See pmlogsearch on page 427 for more information about the syntax and usage of the `pmlogsearch` command.

`pmlogsearch` performs a search across all policy servers in the policy group and returns a list of events (and associated keystroke log file names) for requests matching the specified criteria. You specify search criteria using the following options (you must specify at least one search option):

Table 23: Search criteria options

Command	Description
--after "YYYY/MM/DD hh:mm:ss"	Search for sessions initiated after the specified date and time.
--before "YYYY/MM/DD hh:mm:ss"	Search for sessions initiated before the specified date and time.
--host hostname	Search for sessions that run on the specified host.
--result accept|reject	Return only events with the indicated result.
--text keyword	Search for sessions containing the specified text.
--user username	Search for sessions by the specified requesting user.

The following `pmlogsearch` options support the use of wildcards, such as ***** and **?**:

- --host
- --user

To match one or more characters, you can use wild card characters (such as ? and *) with the `--host`, `--text`, and `--user` options; but you must enclose arguments with wild cards in quotes to prevent the shell from interpreting the wild cards.

If there is a keystroke log associated with the event, it displays the log host and pathname along with the rest of the event information.

The following example lists two events with keystroke (IO) logs:

```
      # pmlogsearch --user sally
Search matches 2 events
2013/03/16 10:40:02 : Accept : sally@qpmsrv1.example.com
   Request: sally@qpmsrv1.example.com : id
   Executed: root@qpmsrv1.example.com : id
   IO Log: qpmsrv1.example.com:/opt/quest/qpm4u/iologs/demo/sally/id_20120316_1040_
ESpL6L
```

```
2013/03/16 09:56:22 : Accept : sally@qpmsrv2.example.com
   Request: sally@qpmsrv2.example.com : id
   Executed: root@qpmsrv2.example.com : id
   IO Log: qpmsrv2.example.com:/opt/quest/qpm4u/iologs/demo/sally/id_20120316_0956_
mrVu4I
```

pmreplay

You can use the `pmreplay` command to replay a keystroke log file if it resides on the local policy server.

To replay the log, run:

```
# pmreplay <path_to_keystroke_log>
```

For example, the following command replays the first `ls -l /etc` log from the previous example:

```
# pmreplay /opt/quest/qpm4u/iologs/demo/sally/id_20120316_1040_ESpL6L
```

pmremlog

If the keystroke log resides on a remote policy server, you can use the `pmremlog` command with the `–h <remote_host>` and `–p pmreplay` options to remotely replay a keystroke log file. You specify the path argument to the remote `pmreplay` after the `--` flag.

For example, enter the following command all on one line:

```
# pmremlog -h qpmsrv2 -p pmreplay -- /opt/quest/qpm4u/iologs/demo/sally/id_20120316_
0956_mrVu4I
```

Host names may appear in the event logs and keystroke log files in either fully qualified format (`myhost.mycompany.com`) or in short-name format (`myhost`), depending on how host names are resolved and the use of the `shortnames` setting in the `pm.settings` file. To ensure that either format is matched, when you specify a host name search criteria, use the short-host name format with an asterisk wild card (For example, `myhost*`).

Listing event logs

You can list the events that are logged when you run a command, whether accepted or rejected by the policy server.

Keystroke logs are related to events. When you run a command, , such as `pmrun whoami`, the policy server either accepts or rejects the command based on the policy. When the policy server accepts the command, it creates an event and a corresponding keystroke log. If it rejects the event, it does not create a keystroke log. In order to view a keystroke log, you must first list events to find a particular keystroke log.

One Identity recommends that you use Management Console for Unix for viewing event logs and replaying keystroke logs. The mangement console provides comprehensive reporting tools and an intuitive user interface for easy navigation of the event and keystroke log data. However, you can also use command line utilities to display a list of events.

The `pmlog` command displays event log entries, such as events by date and time, host, user, run user, command, and result.

To display a list of events from the command line on the policy server

1. From the command line, enter:

   ```
   # pmlog --after "2011/05/06 00:00:00" --user "tuser"
   ```

   `pmlog` provides direct and flexible access to the event logs on the local policy server and is capable of complex queries.

   If you run a command, you might see output similar to the following which indicates the policy server has successfully accepted or rejected commands:

   ```
   Accept 2011/05/11 13:20:04 tuser@ myhost.example.com -> root@
   myhost.example.com
      whoami
      Command finished with exit status 0
   Accept 2011/05/11 14:05:58 tuser@ myhost.example.com -> root@
   myhost.example.com
      whoami
      Command finished with exit status 0
   Reject 2011/05/11 14:06:17 tuser@ myhost.example.com
      Fakecmd
   ```

   The following `pmlog` options support the use of wildcards, such as ***** and **?**:

   - --user
   - --runuser
   - --reqhost
   - --runhost
   - --masterhost

   You can also use the `pmremlog` command on the primary policy server to run `pmlog` on secondary policy servers. For example:

   ```
   # pmremlog –h polsrv2 –p pmlog -- --user myuser --command sh
   ```

Related Topics

pmlog

pmremlog

Backing up and archiving event and keystroke logs

Use the pmlogadm program to perform backup or archive operations on a policy server's event log database. Because Privilege Manager for Unix stores keystroke logs in individual flat files on the policy server, you may use standard Unix commands to back up or archive them. Make sure the keystroke log files are not associated with active sessions prior to backup or archive.

Disabling and enabling services

While pmlogadm can perform the backup and archive operations on a live event log database, for best results we recommend that you follow these steps prior to performing a backup or archive.

1. Stop the pmserviced and pmlogsrvd services.
   This example shows how to disable services on Redhat Linux systems:

   ```
   # service pmserviced stop
   Stopping pmserviced service:     done
   # service pmlogsrvd stop
   Stopping pmlogsrvd service:     done
   ```

2. Ensure there are no running pmmasterd processes:

   ```
   # ps -ef | grep pmmasterd
   ```

   A running pmmasterd process indicates that there may be an active Privilege Manager for Unix session.

This procedure also allows you to safely backup or archive any keystroke log files. Once the backup or archive operation has completed, remember to restart the pmserviced and pmlogsrvd services.

This example shows how to restart the services on Redhat Linux systems:

```
# service pmlogsrvd start
Starting pmlogsrvd service:     done
# service pmserviced start
Starting pmserviced service:     done
```

Backing up event logs

The pmlogadm backup command creates a clean backup copy of your event log database.

This example performs a backup of the current event log database, placing the copy in the /backup directory:

```
# pmlogadm backup /var/opt/quest/qpm4u/pmevents.db /backup
5 / 208 pages complete
10 / 208 pages complete
...
205 / 208 pages complete
208 / 208 pages complete
```

Backing up keystroke logs

Privilege Manager for Unix stores the keystroke logs in individual files and do not require any special commands for processing.

This example uses the `unix cp` command to recursively copy the keystroke logs to the `/backup` directory:

```
# cp -r /var/opt/quest/qpm4u/iolog /backup
```

Archiving event logs

The `pmlogadm` archive command creates an archive of old event logs and removes the old event logs from the current database. The following example archives logs for all events that occurred before April 1, 2014 from the current event log database, creating an archive database in the `/archive/2014Q1` directory.

If you omit the `--no-zip` option, `pmlogadm` also creates a tar-gzip'ed archive of the database files.

```
# pmlogadm archive /var/opt/quest/qpm4u/pmevents.db 2014Q1 \
  --dest-dir /archive --no-zip --before "2014-04-01 00:00:00"
Archive Job Summary
     Source Log : /var/opt/quest/qpm4u/pmevents.db
   Archive Name : 2014Q1
Destination Dir : /archive
    Zip Archive : No
   Cut off time : 2014/04/01 00:00:00

No pmlogsrvd pid file found, assuming service is not running.
X events will be archived.
Adding events to the archive.
Verifying archive.
Archive verification completed successfully. Removing events from source log.
Archive task complete.
```

Archiving keystroke logs

You can use the `pmlog` command with some carefully chosen options to get a list of keystroke logs associated with the event logs you archive. In this example, you process the list generated by `pmlog`, with the Unix `xargs` and `mv` commands to move the keystroke logs into the `/archive/2014Q1/iolog` directory.

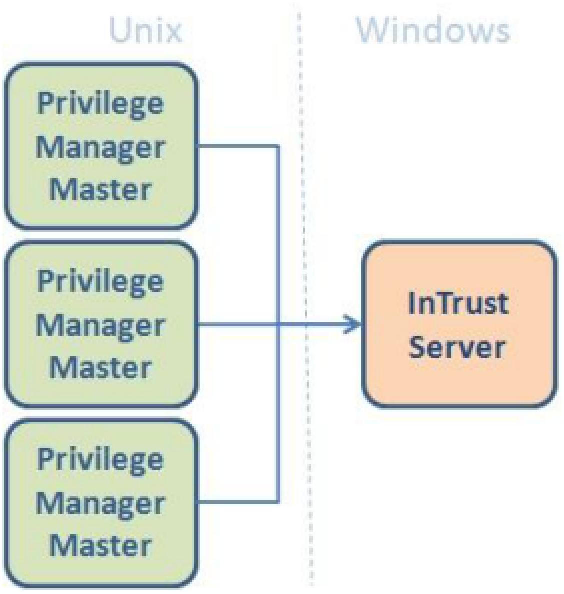
```
# mkdir /archive/2014Q1/iolog
# pmlog -f /archive/2014Q1/archive.db \
   -c "defined iolog && length(iolog) != 0" -p iolog \
   | xargs -i{} mv {} /archive/2014Q1/iolog
```

The usage of the xargs command may differ depending on your platform.

InTrust Plug-in for Privilege Manager for Unix

Quest® InTrust for Active Directory provides a centralized auditing point allowing you to collect and report on the audit data from Privilege Manager for Unix as well as many other data sources you may have in your IT infrastructure.

Figure 10: Audting with InTrust Plug-in



InTrust for Active Directory auditing capabilities allow you to collect and report on the audit data from your Privilege Manager for Unix Security system. Featuring a fully automated workflow, InTrust for Active Directory helps you:

- Gather the Privilege Manager for Unix event logs from the policy servers running on several different platforms
- Consolidate, store, and analyze the gathered data
- Create reports on various aspects of your Privilege Manager for Unix security system operation

InTrust for Active Directory provides reports on the following Privilege Manager for Unix System areas:

- All events
- Elevated privilege events
- All events grouped result
- Out of band events
- Rejected events

InTrust Plug-in requirements

InTrust for Active Directory supports Privilege Manager for Unix version 5.5 and above.

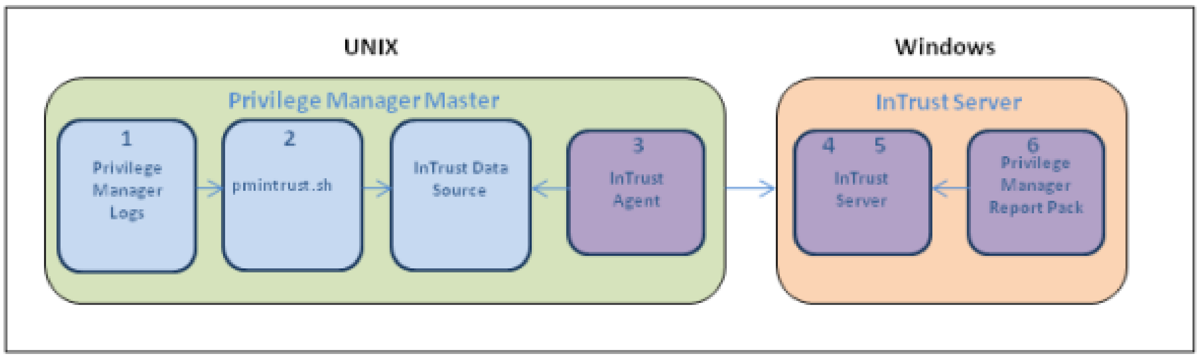
You can collect data from Privilege Manager for Unix hosts running on any of the UNIX platforms supported by InTrust.

To use the MSI installer for the InTrust Reporting Pack, your InTrust Server must use the WindowsSQL Server 2005 as its back-end database.

Installing InTrust Plug-in components

To configure InTrust for Privilege Manager for Unix you must install and configure several components separately. The diagram below shows the major components for the InTrust for Active Directory Plug-in.

Figure 11: InTrust Plug-in components



To install and configure the InTrust for Active Directory Plug-in components

1. Install Privilege Manager for Unix and identify which logs you wish to audit.
2. Install and configure the `pmintrust.sh` script to run as the `root` user to extract the relevant data.

One Identity recommends that you set up a daily cron job to run "pmrun pmintrust.sh" as the pmpolicy service user.

3. Install an InTrust Agent on the Privilege Manager for Unix Policy Server.
4. Configure the InTrust Server: Finding, Gathering, and Storing.
5. Gather Data.
6. Configure the InTrust Server: Reporting.

InTrust Plug-in installation prerequisites

Before you install the InTrust for Active Directory components:

- Install and register an InTrust agent on the Privilege Manager for Unix policy server machine for the collection of syslog messages.

  For more information on this process, refer to the *InTrust Preparing for Auditing and Monitoring Linux* document.

Configuring the policy server for the InTrust Plug-in

Run the `pmintrust.sh` script as the `root` user.

You might need to edit `pmintrust.sh` to ensure it can find all relevant event log files.

The script outputs event log data in a format that the InTrust Agent can handle. When the script runs, it creates a separate file for InTrust called `/tmp/pm_evlog.intrust` containing a plain text version of the events stored in the event log files.

To configure the policy server for the InTrust Plugin

1. Extract the `pmintrust.tgz` archive, located in the utilities directory of the Privilege Manager for Unix distribution media, to the `/tmp` directory.

```
# gzip -dc pmintrust.tgz | tar xvf - -C /tmp
pmintrust/
pmintrust/pmpolicy.crontab
pmintrust/root.crontab
pmintrust/pmintrust.profile
pmintrust/pmintrust.sh
```

2. Copy the pmintrust.sh script to the /opt/quest/sbin directory of your policy server.

   ```
   # cp /tmp/pmintrust/pmintrust.sh /opt/quest/sbin
   ```

3. If necessary, edit the pmintrust.sh script and modify the EVDIRS and EVGLOB variables so that the script can locate the necessary event log files. For example, if your policy defines the eventlog variable as:

   ```
   eventlog="/var/log/eventlogs/"+year+"/"+month+"/"+day+"/"+user+"_events.db";
   ```

   Change the EVDIRS and EVGLOB variables in the pmintrust.sh script to:

   ```
   EVDIRS=`find /var/log/eventlogs -type d`
   EVGLOB="*_events.db"
   ```

4. Configure the system to run the pmintrust.sh script as the root user.

   One Identity recommends that you add a crontab entry as the pmpolicy service user, and configure the cronjob to run pmrun with root user privileges.

   The crontab entry is a file called pmpolicy.crontab in the pmintrust.tgz archive.

   a. The following crontab entry runs pmrun pmintrust.sh at 10:50 pm everyday:

      `50 22 * * * /opt/quest/bin/pmrun /opt/quest/sbin/pmintrust.sh`

      To add the crontab, login (or su) to the pmpolicy service account and run the following command:

      `$ crontab /tmp/pmintrust/pmpolicy.crontab`

      Alternatively, you can configure the script to run directly as the root user by creating a root cron job, and skip part b) of this step.

      There is a root.cronjob file in the pmintrust.tgz archive.

   b. If you are using the default profile-based policy, add the pmintrust.profile to your policy to allow the pmpolicy service account to run the pmintrust.sh script as the root user.

      To checkout, add, and commit the changes to the policy, run the following pmpolicy command:

      ```
      # /opt/quest/sbin/pmpolicy checkout -d /tmp
      # cp /tmp/pmintrust/pmintrust.profile /tmp/policy_pmpolicy/profiles/
      # chown pmpolicy:pmpolicy /tmp/policy_
      pmpolicy/profiles/pmintrust.profile
      # /opt/quest/sbin/pmpolicy add -p profiles/pmintrust.profile -d /tmp
      # /opt/quest/sbin/pmpolicy commit -d /tmp -l ”add pmintrust profile”
      ```

5. Run a new command with Privilege Manager for Unix to verify the change, such as:

   ```
   # pmrun id
   ```

6. Allow the cronjob to run at the scheduled time, then verify the InTrust event log file, `/tmp/pm_evlog.intrust`, was created and contains your test event.

Installing the InTrust Knowledge Pack

To install the InTrust Knowledge Pack

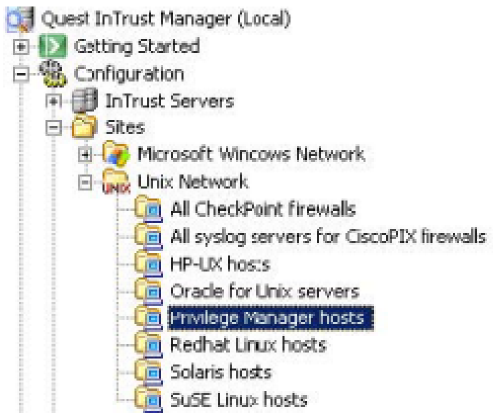
1. Using a InTrust for Active Directory Administration account, login to your InTrust for Active Directory server.
2. Extract the `Privilege_Manager_InTrust_<version>.zip` file to a temporary folder, such as, `d:\temp`.
3. Open a command prompt and change to the following directory:

```
<INTRUST_HOME>\Server\ADC\SupportTools\
```

4. Import each of the XML files using the InTrustPDOImport.exe command, as following:

```
# InTrustPDOImport.exe -import D:\temp\PM_DataSource.xml
# InTrustPDOImport.exe -import D:\temp\PM_GatheringJob.xml
# InTrustPDOImport.exe -import D:\temp\PM_GatheringJob_igtc.xml
# InTrustPDOImport.exe -import D:\temp\PM_GatheringPolicy.xml
# InTrustPDOImport.exe -import D:\temp\PM_GatheringTask.xml
# InTrustPDOImport.exe -import D:\temp\PM_Site.xml
```

5. Verify the Privilege Manager for Unix objects are in the InTrust Manager, under **Sites**:



InTrust Knowledge Pack objects

Table 24: InTrust Knowledge Pack objects

Object type	Objects
Gathering policy	'Privilege Manager for Unix: Event Log Monitoring'
Job	'Gather Privilege Manager for Unix Events'
Task	'Privilege Manager for Unix daily collection of events'
Site	'Privilege Manager for Unix hosts'
Report	'Privilege Manager for Unix All Events' 'Privilege Manager for Unix All Events By Result' 'Privilege Manager for Unix Elevated Privilege Events' 'Privilege Manager for Unix Policy Server By Result' 'Privilege Manager for Unix Policy Server Events' 'Privilege Manager for Unix Rejected Events' 'Privilege Manager for Unix Out Of Band Events'
'Privilege Manager for Unix Event Log'	

Installing the InTrust Reporting Pack

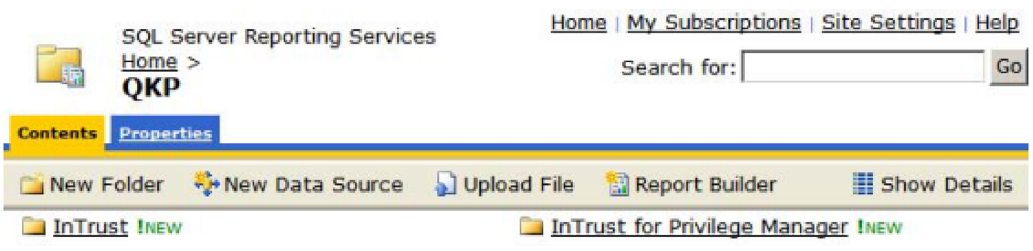
To install the InTrust Reporting Pack

1. Using an InTrust Administration account, log in to your InTrust server.
2. Run the MSI file extracted in the previous section from `Privilege_Manager_InTrust_<version>.zip`

   ```
   # d:\temp\QPM4U.1.0.0.006.msi
   ```

   To use the MSI installer for the InTrust Reporting Pack, your InTrust Server must use the WindowsSQL Server 2005 as its back-end database.
3. Follow the instructions in the on-screen Wizard.
4. Using a web browser, navigate to your InTrust reports and verify that you now have an InTrust for Privilege Manager for Unix section, for example:

```
http://<Intrust Server>/Reports
```



Configuring the InTrust data collection

To install the InTrust data collection

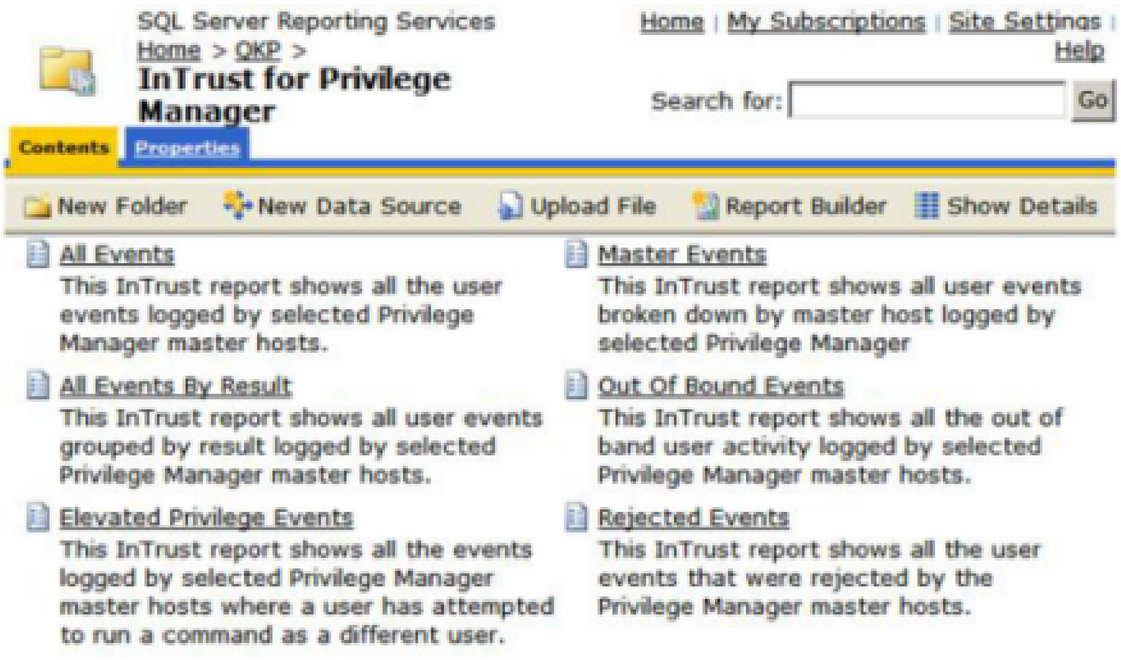
1. Using an InTrust Administration account, log in to your InTrust server.
2. From the menu, navigate to: **Configuration | Sites | Unix Network | Privilege Manager for Unix hosts**.
3. Right click, then select **Properties**.
4. Select the **Objects** tab, click **Add | Computer**, then enter the name of your Privilege Manager for Unix policy server InTrust agent.
5. Click **Apply**, then **OK**.
6. From the menu, navigate to: **Workflow | Tasks | Privilege Manager for Unix daily collection of events**.
7. Right click, then select **Run**.
8. From the menu, navigate to: **Workflow | Sessions** and view the status of your running task which should complete within a couple of minutes, depending on the size of your InTrust event log.
9. Verify that the task completes successfully without errors.

Viewing InTrust reports

To view InTrust reports

1. Using a web browser, navigate to your InTrust reports and verify that you now have an **InTrust for Privilege Manager for Unix** section.

   http://<Intrust Server>/Reports


SQL Server Reporting Services
Home > QKP >
InTrust for Privilege Manager

Home | My Subscriptions | Site Settings | Help
Search for: Go
Contents Properties
New Folder New Data Source Upload File Report Builder Show Details

All Events
This InTrust report shows all the user events logged by selected Privilege Manager master hosts.

Master Events
This InTrust report shows all user events broken down by master host logged by selected Privilege Manager

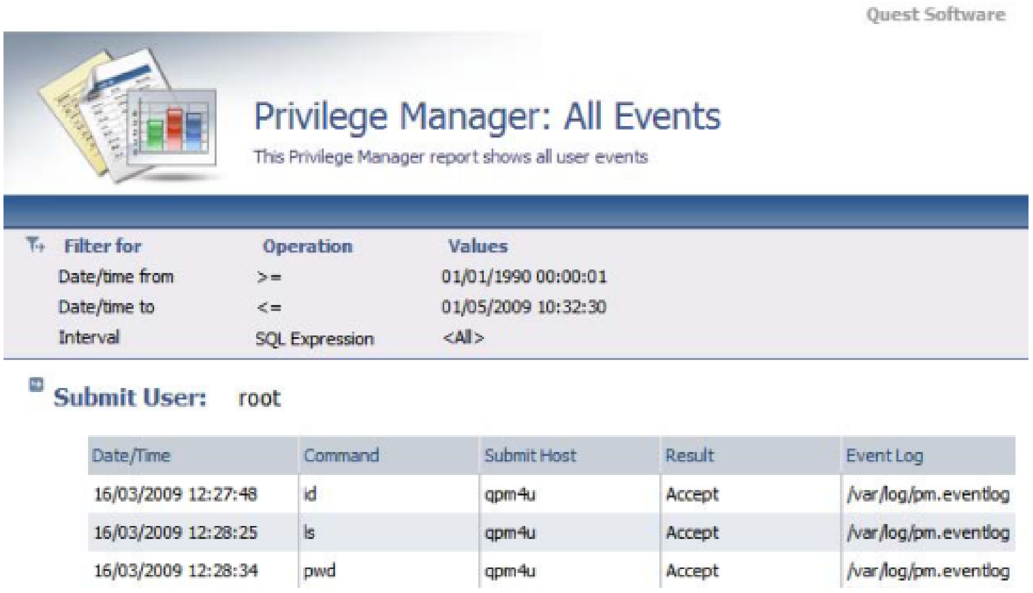
All Events By Result
This InTrust report shows all user events grouped by result logged by selected Privilege Manager master hosts.

Out Of Bound Events
This InTrust report shows all the out of band user activity logged by selected Privilege Manager master hosts.

Elevated Privilege Events
This InTrust report shows all the events logged by selected Privilege Manager master hosts where a user has attempted to run a command as a different user.

Rejected Events
This InTrust report shows all the user events that were rejected by the Privilege Manager master hosts.

2. Select the report type that you want to generate, based on the data currently held in InTrust.


Quest Software

Privilege Manager: All Events
This Privilege Manager report shows all user events

Filter for	Operation	Values
Date/time from	>=	01/01/1990 00:00:01
Date/time to	<=	01/05/2009 10:32:30
Interval	SQL Expression	<All>

Submit User: root

Date/Time	Command	Submit Host	Result	Event Log
16/03/2009 12:27:48	id	qpm4u	Accept	/var/log/pm.eventlog
16/03/2009 12:28:25	ls	qpm4u	Accept	/var/log/pm.eventlog
16/03/2009 12:28:34	pwd	qpm4u	Accept	/var/log/pm.eventlog

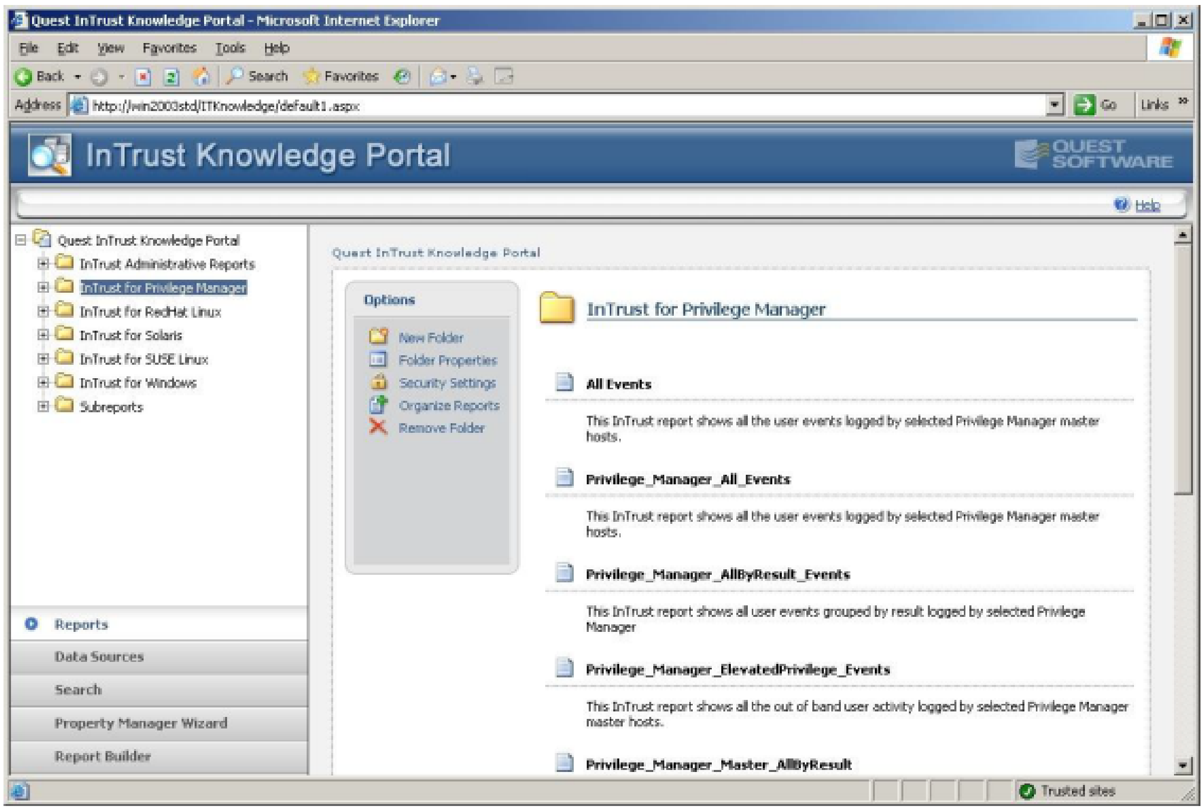
Generating reports

InTrust provides all of its reporting services through the InTrust Knowledge Portal which is based on Microsoft SQL Server Reporting Services. This provides functionality to generate reports dynamically from the InTrust data store and display them though a simple browser based utility.

The Knowledge Portal allows you to create reports manually, however there are a number of pre-compiled reports that gather the following Privilege Manager for Unix event log data:

- All events
- Elevated privilege events
- All events grouped result
- Out of band events
- Rejected events

The reports are provided in a `.msi` installer which installs and configures the required Knowledge Portal components. To view the reports, simply load the Knowledge Portal using **Start | Programs | Quest Software | Quest InTrust Knowledge Portal | Quest InTrust Knowledge Portal**, then select **InTrust for Privilege Manager for Unix** from the report list.



All Events

This InTrust report shows all the user events logged by selected Privilege Manager master hosts.

Privilege_Manager_All_Events

This InTrust report shows all the user events logged by selected Privilege Manager master hosts.

Privilege_Manager_AllByResult_Events

This InTrust report shows all user events grouped by result logged by selected Privilege Manager

Privilege_Manager_ElevatedPrivilege_Events

This InTrust report shows all the out of band user activity logged by selected Privilege Manager master hosts.

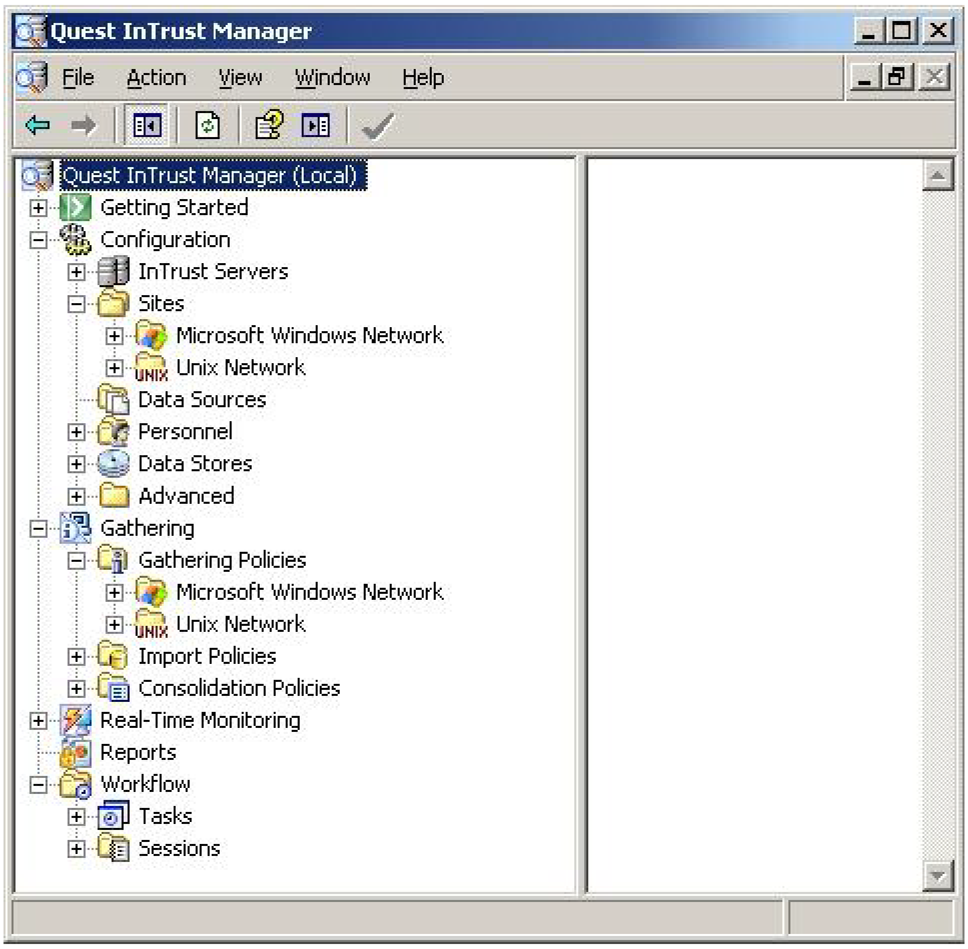
Privilege_Manager_Master_AllByResult

For more information, please refer to the InTrust for Active Directory documentation.

Gathering InTrust data

The general concept behind the InTrust server is that you configure a number of objects individually to perform a specific part of the data gathering process. These objects are then combined to form a work flow system. These are the objects you need to configure to complete a simple data gathering work flow:

- **Configuration | Sites**: Contains a list of Privilege Manager for Unix policy servers from which the gathering process gathers data.
- **Configuration | Data Sources**: Stores details about the data source format.
- **Gathering | Gathering Policies**: Specifies which data source to use.
- **Workflow | Tasks**: A task contains a list of jobs, each of which specifies the frequency at which to gather data according to a particular gathering policy.
- **Configuration | Data Stores**: Database or InTrust Repository that stores the imported data.



You can either manually create these objects or import them from the Privilege Manager for Unix Knowledge Pack.

To import these objects

1. Run the `InTrustPDOImport` import utility:

   ```
   InTrustPDOImport.exe -import <object>
   ```

The import utility is located by default in:

`<install location>\Quest Software\InTrust\Server\ADC\SupportTools`

2. Once you have imported the objects, add the list of Privilege Manager for Unix policy servers to the site object.

   For more information about importing objects, refer to the *InTrust Creating Custom Data Collection* documentation.

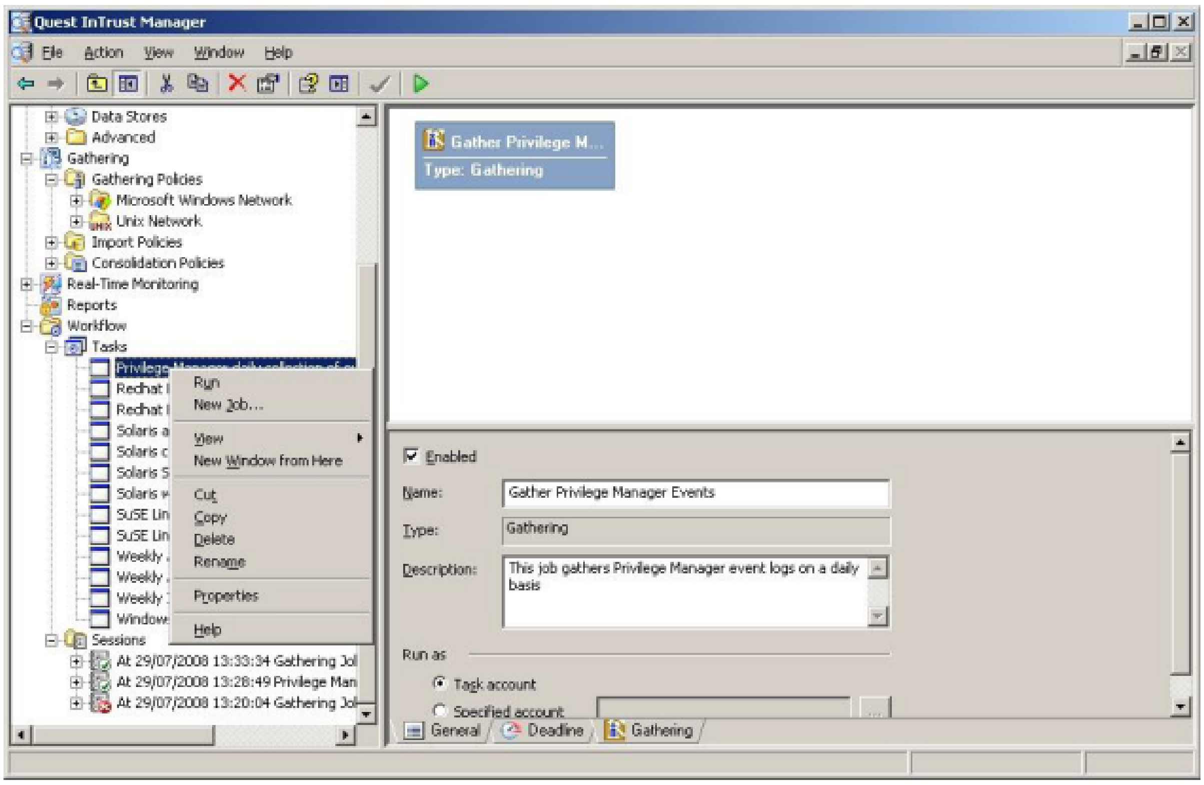
   Once configured, the InTrust server objects can gather the data.

   By default the Privilege Manager for Unix gathering task provided in the knowledge pack retrieves event log data on a daily basis. However, you can customize this setting in the Gathering Policy.

One Identity recommends that you verify the gathering process by running the task manually.

To run the gathering process manually

1. In the Quest InTrust Manager, navigate to **Workflow | Tasks**.
2. Right-click the Privilege Manager for Unix task and select **Run**.



Gather Privilege M...
Type: Gathering

Enabled

Name: Gather Privilege Manager Events

Type: Gathering

Description: This job gathers Privilege Manager event logs on a daily basis

Run as

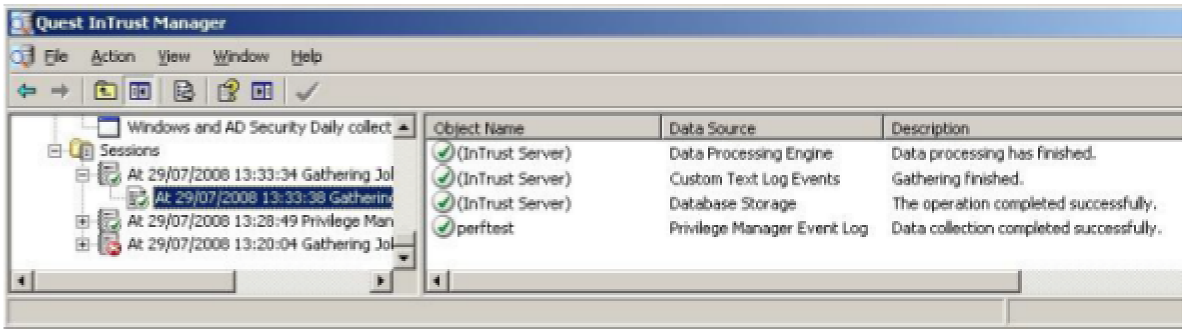
Task account

Specified account

At 29/07/2008 13:33:34 Gathering Jol
At 29/07/2008 13:28:49 Privilege Man
At 29/07/2008 13:20:04 Gathering Jol

The details of a gathering job are recorded in **Workflow | Sessions**, accessible by means of the tree view.

The example below shows the result of a successful job.


Quest InTrust Manager
File Action View Window Help
Windows and AD Security Daily collect
Sessions
At 29/07/2008 13:33:34 Gathering Jol
At 29/07/2008 13:33:38 Gatherinc
At 29/07/2008 13:28:49 Privilege Man
At 29/07/2008 13:20:04 Gathering Jol

Object Name	Data Source	Description
✓ (InTrust Server)	Data Processing Engine	Data processing has finished.
✓ (InTrust Server)	Custom Text Log Events	Gathering finished.
✓ (InTrust Server)	Database Storage	The operation completed successfully.
✓ perftest	Privilege Manager Event Log	Data collection completed successfully.

12

Troubleshooting

To help you troubleshoot, One Identity recommends the following resolutions to some of the common problems you might encounter as you deploy and use Privilege Manager for Unix.

Displaying profile-based policy debug information

To view debug information for profile-based policy, set the value for the `pf_tracelevel` variable either globally in `global_profile.conf`, or in an individual profile.

To set the pf_tracelevel variable in the profile

1. Enable the `pf_tracelevel` option. For example:

```
# Variable: pf_tracelevel: Enables tracing/debugging output at different
levels:
# 1:show reason for reject, 2: verbose output, 3: show debug trace
pf_tracelevel=2;
```

2. To view the trace output, run a command with `pmrun`, like this:

```
$ pmrun id
*******************************************************************
** One Identity Privilege Manager for Unix Version 6.0.0 (006) **
** This request is being authorized on master :<HostName>
** User "luser" has submitted a request from host "<HostName>"
** to run the command "id"
*******************************************************************
   User : luser
   Host : <HostName>
   Command : id
* Check profile:profiles/admin.profile
** Profile:admin does not match user
** Profile:admin does not match UNIX group
** Profile:admin does not match AD group list
* Check profile:profiles/demo.profile
** Validate command:id
** Profile:demo cmd[0] matches command:id Request accepted by the "demo"
profile

All interactions with this command will be recorded in the file:
   /var/opt/quest/qpm4u/iolog/demo/luser/id_20121023_1038_qu3zcf

Executing "id" as user "root" ...

*****************************************************************************
**

uid=0(root) gid=0(root) groups=0(root)
```

Enabling program-level tracing

Technical Support may ask you to create a trace file when you run a program by using the -z option. The -z option enables tracing on a specific program or currently running process.

To display program-level tracing

1. Run a program with the -z option, like this:

   ```
   # <CommandName> -z on
   ```

   The -z option creates a <CommandName>.ini file which then creates a <CommandName>.trc file when you run the command. The .trc file contains the debug information. Both the .ini and the .trc files are created in the /tmp directory.
2. Once you have finished getting the trace output you need, run the program with the -z off option so the log will not continue to grow.

Load balancing and policy updates

pmloadcheck is both a command and a background daemon (run with the –i flag). When run as a command, it checks, updates, and reports on the status of the policy server. You can use pmloadcheck from a policy server or PM Agent.

When run as a daemon process, it keeps track of the status of the policy servers for failover and load-balancing purposes. On policy servers, pmloadcheck is responsible for keeping the production policy file up to date.

See pmloadcheck on page 417 for more information about the syntax and usage of this command.

Policy servers are failing

The primary and secondary policy servers must be able to communicate with each other and the remote hosts must be able to communicate with the policy servers in the policy group.

For example, if you run the `pmloadcheck` command on a policy server or PM Agent to determine that it can communicate with other policy servers in the policy group, you may get output similar to the following:

```
++ Checking host:myhost.example.com (10.10.181.87) ... [FAIL]
```

There are several possible reasons for failure:

- Policy server host is down
- Network outage
- Service not running on policy server host

These are some ways to verify that the Privilege Manager for Unix service is running properly on the policy server host:

1. To verify the policy server configuration, run

   ```
   # pmsrvinfo
   ```

2. To verify that the service is running, enter

   ```
   # ps -ef | grep pmserviced
   ```

3. To verify that the pmmasterd port is in a listening state on the primary policy server, enter

   ```
   # netstat -na | grep 12345
   ```

4. To verify the service is enabled, look for the following in the Privilege Manager for Unix configuration file (`/etc/opt/quest/qpm4u/pm.settings`)

   ```
   pmmasterdEnabled YES
   ```

5. To restart the service (on a Linux host), enter

   ```
   # /etc/init.d/pmserviced restart
   ```

-Or-

```
pmserviced -s
```

6. Check for other communication issues, such as with your firewall, name resolution, dead network interface, and so forth.

Appendix A

Privilege Manager for Unix Policy File Components

This appendix provides detailed information about the components that you use to construct the pmpolicyPrivilege Manager for Unix security policy file.

Lexical and syntactic productions

One Identity uses the following language standards to define the grammar of the policy scripting language used in Privilege Manager for Unix.

Table 25: Lexical productions

Production	Description
<identifier>	::= [A-Za-z][A-Za-z0-9_]*
<number>	::= [0-9]+
<octalnumber>	::= 0[0-7]+
<hexnumber>	::= 0x[a-fA-F0-9]+
<realnumber>	::= <number> '.' <number>
<string>	::= \" <non-double-quote | backslashed-double-quote >* \" | ' <non-single-quote | backslashed-single-quote >* '
<non-double-quote>	::= [^\"]
<backslashed-double-quote>	::= \\\"

Production	Description
<non-single-quote>	::= [^']
<backslashed-single-quote>	::= \\'
<comment>	::= <shell-style-comment> | <c-style-comment> | <cplusplus-style-comment>
<shell-style-comment>	::= '#' [^\n]*
<c-style-comment>	::= /* [^*/]* */
<cplusplus-style-comment>	::= // [^\n]*

Table 26: Syntactic productions

Production	Description
Policy	::= { Statement | Procedure }
Procedure	::= ('procedure' | 'function') <identifier> '(' [Parameters] ')' BlockStatement
Parameters	::= Parameter {',' Parameter }
Parameter	::= <identifier> ['=' Expression]
Statements	::= Statement { Statement }
Statement	::= IfStatement | ForStatement | DoWhileStatement | WhileStatement | SwitchStatement | BreakStatement | ContinueStatement | ReturnStatement | AcceptStatement | RejectStatement | IncludeStatement | ReadOnlyStatement | ReadOnlyExceptStatement | ExpressionStatement | BlockStatement
IfStatement	::= 'if' '(' Expression ')' Statement ['else' Statement]
WhileStatement	::= 'while' '(' Expression ')' Statement
DoWhileStatement	::= 'do' BlockStatement 'while' '(' Expression ')' ';'
ForStatement	::= 'for' '(' Expressions ';' Expression ';' [Expression] ')' Statement | 'for' '(' <identifier> 'in' Expression ')' Statement
SwitchStatement	::= 'switch' '(' Expression ')' '{' [Cases][Default] '}'
Cases	::= Case { Case }
Case	::= 'case' Expression ':' Statements

Production	Description
Default	::= 'default' Statements
BreakStatement	::= 'break' ';'
ContinueStatement	::= 'continue' ';'
ReturnStatement	::= 'return' [Expression] ';'
IncludeStatement	::= 'include' Expression ';'
AcceptStatement	::= 'accept' ';'
RejectStatement	::= 'reject' [Expression] ';'
ReadOnlyStatement	::= 'readonly' Expression ';'
ReadOnlyExceptStatement	::= 'readonlyexcept' Expression ';'
ExpressionStatement	::= Expression ';'
BlockStatement	::= '{' Statements '}'
Expressions	::= Expression {',' Expressions }
Expression	::= AssignmentExpression | ConditionalExpression
AssignmentExpression	::= PrimaryExpression { AssignmentOp Expression }
AssignmentOp	::= '=' | '+=' | '-=' | '*=' | '/='
ConditionalExpression	::= LogicalOrExpression ['?' Expression ':' Expression]
LogicalOrExpression	::= LogicalAndExpression { '||' LogicalAndExpression }
LogicalAndExpression	::= BitwiseOrExpression { '&&' BitwiseOrExpression }
BitwiseOrExpression	::= BitwiseAndExpression { '|' BitwiseAndExpression }
BitwiseAndExpression	::= EqualityExpression { '&' EqualityExpression }
EqualityExpression	::= RelationalExpression { EqualityOp Relation-alExpression }
EqualityOp	::= '==' | '!=
RelationalExpression	::= AdditiveExpression { RelationalOp AdditiveExpression }
RelationalOp	::= '<' | '>' | '<=' | '>=' | 'in'
AdditiveExpression	::= MultiplicativeExpression { AdditiveOp Multi-plicativeExpression }
AdditiveOp	::= '+' | '-'
MultiplicativeExpression	::= PrimaryExpression { MultiplicativeOp PrimaryEx-

Production	Description
	pression }
MultiplicativeOp	::= '*' | '/' | '%'
PrimaryExpression	::= PrefixAssignmentExpression | DesignatorExpression | LiteralExpression | '-' Expression | '!' Expression | 'typeof' Expression | 'defined' <identifier> | '(' Expression ')'
PrefixAssignmentExpression	::= PrefixOp <identifier>
PrefixOp	::= '++' | '--'
DesignatorExpression	::= <identifier> | <identifier> PostfixOp | <identifier> Arguments | <identifier> ListAccess { ListAccess }
PostfixOp	::= '++' | '--'
Arguments	::= '(' [Expressions] ')'
ListAccess	::= '[' Expression ']'
LiteralExpression	::= <string> | <number> | <hexnumber> | <octal-number> | <realnumber> | ListLiteral
ListLiteral	::= '{' [Expressions] '}'

Data types

The following data types are available for use in the policy scripting language.

Table 27: Data types

Type	Description	Example
array	A multi-dimensional array that can contain any mixture of types.	`Users={"fred", "jen", "sally"}; Ids={1, 9, 10}; Usermap={ Users, Ids}; print(umap[0][2] + " -> " + umap[1][2]);`
boolean	The values true and false.	x = true;
double	A number with a fractional component.	`x=2.5; y=4.3; print(x+y); #prints 6.8`
int	The type integer includes the set of integers (…, -2, -1, 0, 1, 2, …). The constants `true` and `false` are defined to have the values 1 and 0, respectively.	`count=0; x=y=1;` You can specify an octal number by preceding it with a leading zero. For example, when specifying a `umask` value `runumask=022`

Type	Description	Example
	Specify hexadecimal numbers with the prefix 0x.	
ldapid	Special type to support LDAP functions.	
ldapsearchresult	Special type to support LDAP functions.	
list	An ordered group of strings separated by commas and surrounded by curly braces. List elements are accessed by postfixing them with square brackets [ ] containing the index of the desired element. Indices start at 0.	`mylist = {"string zero", "string one", "string two"}; print( {"a", "b", "c"}[1] ); # prints "b"`
string	A sequence of zero or more characters within single or double quotes.	`Mystr="this is a string"; Str1="user: " + user;`
undefined	A variable is assigned a type when it is assigned a value of that type. A variable that is referenced but has not been assigned a value is set to the type undefined.	`if (typeof(myvar) == "undefined") { myvar=user;}`

Operators and expressions

Operators specify what is done to variables, constants, and expressions.

Expressions combine variables and constants to produce new values. Expressions which use the operators !, ||, &&, ==, !=, <, >, <=, >=, in, !in and () return a boolean value of `true` or `false`.

Unless otherwise specified, these operators are valid for all types of variables.

Table 28: Variable operators

Operator	Description	Example
=	*variable = expression* The **assignment** operator assigns a copy of the *expression* on the right side to the *variable* on the left side.	`count=0; x=y=1; str="this is a string"; users={"fred", "john"}; list1=users; list [1]="johnr";`
+=	*variable += expression*	`count=1; count +=10; print (count); #prints 11`

Operator	Description	Example
	The **addition self-assignment** operator adds the value of the *expression* to the value of the *variable* and stores the result in the *variable*. Valid for integer, double and string data types.	
-=	*variable -= expression* The **subtraction self-assignment** operator subtracts the value of the *expression* from the value of the *variable* and stores the result in the *variable*. Valid for integer and double data types.	`Count=10; Count-=2; print (Count); #prints 8`
*=	*variable *= expression* The **multiplication self-assignment** operator multiplies the value of the *expression* by the value of the *variable* and stores the result in the *variable*. Valid for integer and double data types.	`tot =10; tot *= 10; print (tot); #prints 100`
/=	*variable /= expression* The **division self-assignment** operator divides the value of the *variable* by the value of the *expression* and stores the result in the *variable*. Valid for integer and double data types.	`tot=10; tot /=2; print(tot); #prints 5`
var++	*variable ++* The **postfix auto increment** operator returns the value of the *variable* and adds 1 to the *variable*. Valid for integer and double data types.	`count=0; userlist [count++]="john";`
++var	*++variable* The **prefix auto increment** operator adds 1 to the *variable* and returns the result. Valid for integer and double data types.	`++count=-1; userlist [++count]="john";`
var--	*variable --* The **postfix auto increment** operator returns the value of the *variable* and subtracts 1 from the *variable*. Valid for integer and double data types.	`for(i=10; i>0; i--) {…}`
--var	*--variable* The **prefix auto increment** operator	`i=9; do { userlist[--i] = value; } while (i>0);`

Operator	Description	Example
	subtracts 1 from the variable and returns the result. Valid for integer and double data types.	
!	*!expression* **Negation** operator negates the value of the *expression* and returns the result.	`while (!found) {…} no = !true; if (!(a&&b)) reject; #request is rejected if a AND b #are not true`
||	*expression || expression* **Logical or** operator resolves to `true` if either *expression* resolves to `true`.	`if ((user in list1) || (user in list2)) {accept;}`
&&	*expression && expression* **Logical or** operator resolves to `true` if both *expressions* resolve to `true`.	`if ((defined myuser) && (myuser == "root")) {accept;}`
|	*expression | expression* **Bitwise or** operator resolves to `true`.	`if (word | 0x4) {…}`
&	*expression & expression* **Bitwise and** operator resolves to `true`.	`if (word & 0x4) {…}`
==	*expression == expression* Resolves to `true` if the *expressions* are identical.	`if (user == "root") {…} if (x==1){…} if (list1 == {"one"}) {…}`
!=	*Expression != expression* **Logical or** operator resolves to `true` if the *expressions* are not identical.	`if (found != true) {…} if (user != "root") {…} if (list1 != {"root"}) {…}`
()	*(expression)* Forces a particular order of evaluation.	`if ((a||b) && c) { accept; } if (a || (b && c)) { reject; }`
?:	*Conditional expression ? t_expression : f_ expression* The *conditional expression* is evaluated. If it resolves to `true`, then it evaluates to `t_ expression`, else it evaluates to `f_ expression`.	`runuser = (user == "cory") ? "root" : "sys"; # is equivalent to: # if (user=="cory") { # runuser = "root";} # else { # runuser = "sys";}`
in	*string in expression* Resolves to `true` if the *string* is a member of the list. It performs a glob-style check on each member of the list, so each list element can be a glob expression. The	`list={"root", "admin"}; print ("root" in userlist); #prints 1`

Operator	Description	Example
	string cannot be a glob expression.	
!in	*string !in expression* Resolves to `true` if the *string* is not a member of the list. It performs a glob-style check on each member of the list, so each list element can be a glob expression. The *string* cannot be a glob expression.	`list={"root", "admin"}; print ("john" ! in userlist); #prints 1`
+ - * / %	*expression operator expression* **Mathematical** operators return the result of evaluating the arithmetic *expression*. The normal mathematical rules for order of evaluation apply. All operands must be integers or doubles. The exception is the + operator which will concatenate strings and lists.	`a = 5 + 4 * 2; #a == 13 b = 5 * 4 / 2; #b == 10 c = 5 % 4; #c == 1 d = "string1" + "string2"; #d = "string1 string2" e={"one"}+{"two"}; #e = {"one", "two"};`
< > <= >=	*expression operator expression* **Relational** operators resolve to `true` if the relationship is `true`.	`4 > 7 // evaluates false 4 >= 4 // evaluates true 4 < 1 // evaluates false "foo" == "bar" // false "foo" > "bar" // true, because foo follows bar alphabetically`
export	*export <varname>* Adds a local variable to the event log and I/O log. Can be specified multiple times.	
[]	*list[number]* Returns the value of an element in a list or array.	`list1={"user0", "user1", "user2"}; print(List[2]); #prints user2 list0={"user0", 0}; list1={"user1",1}; maplist={list0, list1}; print (maplist[0][0], maplist[0] [1]); #prints user0 0`
typeof	*typeof expression* Returns a string representation of the type of an *expression*.	`print(typeof x); #undefined x=1; print(typeof x); #integer x="1"; print(typeof x); #string x={"1"};print(typeof x); #array`
defined	*defined variable* Resolves to true if the variable has been declared with a value.	print(defined x); #prints 0 x=1; print(defined x); #prints 1

Appendix B

Privilege Manager for Unix Variables

This appendix provides detailed information about the variables that may be present in event log entries:

- Variable names
- Variable scope
- Global input variables
- Global output variables
- Global event log variables
- PM settings variables

See also Profile variables on page 66 for additional information about policy profile variables.

Variable names

Privilege Manager for Unix uses a number of predefined global variables and user-defined variables within the pmpolicy scripting language.

Here is some general information about user-defined variables:

- A user-defined variable is declared the first time it is assigned a value. If a variable is referenced before it has been assigned a value, it has the special type of "undefined".
- A variable name can be any length.
- You can use any number of user-defined variables.
- The first character of a variable name must be a letter or an underscore (_).
- Variable names are case-sensitive; thus, the names "checkhost" and "CHECKHOST" refer to different variables.
- Keywords are case-sensitive; you must enter them in lower case.

- Loose typing is applied when variables of different types are used. Thus, if you use mixed types with an operator, such as, an integer and a string with a "+" operator, the parser will attempt to convert the result to a string.

Variable scope

All variables are global in scope unless declared from within a function or procedure.

If a variable is first declared in a function or procedure, it has local scope within that particular function or procedure and is deleted once the function or procedure returns.

Example

```
gvar1="global";

 procedure p1() {
      gvar1="changed in f1";            #gvar1 has global scope
      pvar1="local_to_p1";              #pvar1 is local to procedure p1()
p2();
 }

 procedure p2() {
      gvar1="changed in f2"; # gvar1 is still global
      print((defined pvar1? pvar1 : "undefined"));
                                         # this line prints "undefined"
since
                                         # pvar1 is now out of scope
 }
```

Global input variables

The following predefined global variables are initialized from the submit-user’s environment. You can use these variables in the decision making process in the policy file but you cannot change their value.

Table 29: Global input variables

Variable	Data type	Description
alertkeymatch	sting	The pattern matched by `pmlocald`.

Variable	Data type	Description
argc	integer	Number of arguments in the request.
argv	list	List of arguments in the request.
bkgd	boolean	Reflects the "-b" background argument of a pmrun call.
client_parent_pid	integer	Process ID of the client's parent process.
client_parent_uid	integer	User ID associated with the client's parent process.
client_parent_procname	string	Process name of a client's parent process.
clienthost	string	Originating login host.
command	string	Pathname of the request.
cwd	string	Current working directory.
date	string	Current date.
day	integer	Current day of month as integer.
dayname	string	Current day of the week.
domainname	string	The Active Directory domain name for the submit user if Authentication Services is configured.
env	list	List of submit user’s environment variables.
false	integer	Constant value.
FEATURE_LDAP	integer	Read-only constant used with `feature_enabled()` function.
FEATURE_VAS	integer	Read-only constant used with `feature_enabled()` function.
gid	integer	Group ID of the submitting user’s primary group on sudo host.
group	string	Submit user’s primary group.
groups	list	Submit user’s secondary groups.
host	string	Host destined to run the request.
hour	integer	Current hour.
masterhost	sting	Host on which the master process is running.
masterversion	string	Privilege Manager for Unix version of `masterhost`.

Variable	Data type	Description
minute	integer	Current minute.
month	integer	Current month.
nice	integer	nice value of the submit user's login.
nodename	string	Hostname of pmrun agent.
optarg	integer	Contains the parameter for the last argument or empty string.
opterr	integer	Determines whether to display errors from the getopt functions.
optind	integer	Contains the current argument list index. Use with getopt functions.
optopt	string	Contains the letter of the last option that had an issue. Use with getopt functions.
optreset	boolean	Restarts the getopt functions from the beginning.
optstrictparameters	boolean	Lets getopt_long() recognize non-compliant argument parameter forms.
pid	integer	Process ID of the master process.
pmclient_type	integer	The type of client that sent the request.
pmclient_type_pmrun	integer	Read-only constant for pmrun type clients.
pmclient_type_sudo	integer	Read-only constant for sudo type clients.
pmshell	integer	Identifies a Privilege Manager for Unix shell program.
pmshell_builtin	integer	A constant value that identifies a shell builtin command.
pmshell_cmd	integer	Identifies a command run from a Privilege Manager for Unix shell program.
pmshell_cmdtype	integer	Identifies type of a shell subcommand.
pmshell_exe	integer	A constant value that identifies a normal executable command.
pmshell_interpreter	integer	Identifies the program directive of a shell script.
pmshell_prog	string	Name of the Privilege Manager for Unix shell program.
pmshell_script	integer	A constant value that identifies a shell script.
pmshell_uniqueid	string	uniqueid of the Privilege Manager for Unix shell

Variable	Data type	Description
		program.
pmversion	string	Privilege Manager for Unix version string of client.
ptyflags	string	Identifies ptyflags of the request.
requestlocal	integer	Indicates if the request is local.
requestuser	string	User that the submit user wants to run the request.
rlimit_as	string	Controls the maximum memory that is available to a process.
rlimit_core	string	Controls the maximum size of a core file.
rlimit_cpu	string	Controls the maximum size CPU time of a process.
rlimit_data	string	Controls the maximum size of data segment of a process.
rlimit_fsize	string	Controls the maximum size of a file.
rlimit_locks	string	Control the maximum number of file locks for a process.
rlimit_memlock	string	Controls the maximum number of bytes of virtual memory that can be locked.
rlimit_nofile	string	Controls the maximum number of files a user may have open at a given time.
rlimit_nproc	string	Controls the maximum number of processes a user may run at a given time.
rlimit_rss	string	Controls the maximum size of the resident set (number of virtual pages resident at a given time) of a process.
rlimit_stack	string	Controls the maximum size of the process stack.
samaccount	string	The sAMAccountName for the submit user if Authentication Services is configured.
selinux	integer	Identifies whether a client is running an SELinux environment.
status	integer	Exit status of the most recent system command.
submithost	string	Name of the submit host.
submithostip	string	IP address of the submit host.
thishost	string	The value of the thishost setting in pm.settings on the client.

Variable	Data type	Description
time	string	Current time of request.
true	integer	Read-only constant with a value of 1.
ttyname	string	`ttyname` of the submit request.
tzname	string	Name of the time zone on the server at the time the event was read from the event log by pmlog.
uid	integer	User ID of the submitting user on host.
umask	integer	`umask` of the submit user.
unameclient	list	`Uname` output on host.
unamemaster	list	`Uname`output on policy server host.
uniqueid	string	Uniquely identifies a request in the event log.
use_rundir	string	Contains the value "!~!" and represents the `runuser`'s home directory on the `runhost`.
use_rungroup	string	Contains the value "!g!" and represents the `runuser`'s primary group on the `runhost`.
use_rungroups	string	Contains the value "!G!" and represents the `runuser`'s secondary group list on the `runhost`.
use_runshell	string	Contains the value "!!!" and represents the `runuser`'s login shell on the `runhost`.
user	string	Submit user.
year	integer	Year of the request (YY).

alertkeymatch

Description

Type **string** READONLY

`alertkeymatch` contains the pattern matched by `pmlocald`. This variable is not available for use in the policy file, it is only available in the event log. To view the event log, use the `pmlog -l` command.

Example

```
#view all alerts recorded in the audit log that match the pattern "passwd"
pmlog –l -c 'alertkeymatch == "passwd"'
```

Related Topics

alertdate

alertkeysequence

alertkeyaction

alerttime

argc

Description

Type **integer** READONLY

`argc` contains the number of arguments supplied for the original command. This includes the command name itself. For example, if the original command is `pmrun ls –al`, then `argc` is set to 2.

Example

```
# if any arguments are passed to a vi editor program, like vi
 # then verify the path is not in a list of forbidden directories
 if ((basename(command) in vi_program_list) && (argc > 1))
 {
      count=0;
      while (count < length(forbid_dir_list))
      {
           if (glob(forbid_dir_list[count], dirname(argv[1])))
           {
                reject "You are not allowed to edit a file in this 
directory";
             }
           count=count+1;
      }
 }
```

Related Topics

argv

argv

Description

Type **list** READONLY

argv is a list of the arguments supplied for the original command, including the command itself. For example, if the original command is pmrun ls –al, then argv is set to {"ls","-al"}.

Example

```
# if any arguments are passed to an editor program, like vi
 # then verify the path is not in a list of forbidden directories
 if ((basename(command) in vi_program_list) && (argc > 1))
 {
      count=0;
      while (count < length(forbid_dir_list))
      {
           if (glob(forbid_dir_list[count], dirname(argv[1])))
           {
                reject "You are not allowed to edit a file in this
directory";
           }
           count=count+1;
      }
 }
```

Related Topics

argc

bkgd

Description

Type **boolean** READONLY

bkgd reflects the "-b" background argument of a pmrun call. If the user requested the background mode, it is set to 1.

To change whether the call runs in the background, set the runbkgd variable.

client_parent_pid

Description

Type **integer** READONLY

Process ID of client's parent process.

Example

```
# only allow requests submitted from a login shell
 # (parent process name starts with a dash)
 if (client_parent_procname[0] == "-") {
      printf("process info -- name:[%s], pid[%d], uid[%d]\n"
           client_parent_procname, client_parent_pid, client_parent_uid);
      reject "only requests from login shells are allowed";
 }
```

Related Topics

client_parent_uid

client_parent_procname

client_parent_uid

Description

Type **integer** READONLY

User ID associated with the client's parent process.

Example

```
# only allow requests submitted from a login shell
 # (parent process name starts with a dash)
 if (client_parent_procname[0] == "-") {
      printf("process info -- name:[%s], pid[%d], uid[%d]\n"
           client_parent_procname, client_parent_pid, client_parent_uid);
      reject "only requests from login shells are allowed";
 }
```

Related Topics

client_parent_pid

client_parent_procname

client_parent_procname

Description

Type **string** READONLY

Process name of a client's parent process.

Example

```
# only allow requests submitted from a login shell
 # (parent process name starts with a dash)
 if (client_parent_procname[0] == "-") {
      printf("process info -- name:[%s], pid[%d], uid[%d]\n"
           client_parent_procname, client_parent_pid, client_parent_uid);
      reject "only requests from login shells are allowed";
 }
```

Related Topics

client_parent_pid

client_parent_uid

clienthost

Description

Type **string** READONLY

`clienthost` contains the host name/IP address of the requesting host. For a typical `pmrun` command, this will be identical to the `submithost` variable. For a Privilege Manager for Unix shell running as a login shell (for example, `pmksh`, `pmcsh`, `pmsh`, `pmloginshell`, and `pmbash`), this will contain the host name from which the user is logging in, which may not be a Privilege Manager for Unix host. For example, if the user logs in by means of a `telnet` session from a Windows PC, then the `clienthost` variable will contain the host name of the Windows PC. Always use short names when checking the `clienthost` variable, as some login programs may truncate the full host name.

Example

```
# reject commands being issued from unknown workstations
workstations = {"sun34","sun35","sun36"};
if (!(clienthost in workstations))
   reject;
```

Related Topics

submithost

submithostip

runhost

eventloghost

runclienthost

command

Description

Type **string** READONLY

The name of the command being run.

The `command` variable generally contains the full path name of the command being run. Use the `basename()` function to get the command name without the full path.

Example

```
admincommands = {"hostname","kill","shutdown"};
 if (basename(command) in admincommands)
 {
      runuser = "root";
      accept;
 }
```

Related Topics

runcommand

cwd

Description

Type **string** READONLY

`cwd` contains the pathname of the submit user's current working directory.

Example

```
# if command is executed from any directory other than under /usr,
# change the working directory to /tmp
if (cwd != "/usr" && !glob("/usr/*", cwd))
   runcwd = "/tmp";
```

Related Topics

runcwd

date

Description

Type **string** READONLY

`date` contains the date the request was submitted in the form: YYYY/MM/DD.

Example

```
if (pmshell)
{
     # prints the date and time the shell was opened
     print( command + " started " + date + " "+ time );
     accept;
}
```

Related Topics

dayname

minute

hour

day

month

year

time

day

Description

Type **integer** READONLY

day contains the day the request was submitted formatted as an integer in the range: 1–31.

Example

```
if (command == "dailyadmin")
{
     if (day == 1)
     {
          # first day of the month
          runcommand = ""
     }
}
```

Related Topics

dayname

minute

hour

date

month

year

time

dayname

Description

Type **string** READONLY

`dayname` contains the abbreviated name ("Mon", "Tue, "Wed", "Thu", "Fri", "Sat" or "Sun") of the day the request was submitted.

Example

```
switch (dayname)
 {
      case "Mon":
      case "Wed":
      case "Fri":
           adminusers = {"dan","robyn"};
           break;
      case "Tue":
      case "Thu":
           adminusers = {"robyn","cory"};
           break;
      default:
           adminusers = {};
 }
 if (user in adminusers)
 {
      runuser = "root";
      accept;
 }
```

Related Topics

minute

hour

day

month

year

time

date

domainname

Description

Type **string** READONLY

The Active Directory domain name for the submit user if Authentication Services is configured and the client is able to determine the domain name. Otherwise this variable is set to an empty string.

Example

```
# reject if the user is not in the uxwheel AD group
if (vas_user_is_member(samaccount, "uxwheel", domainname) == false)
    reject "user is not in uxwheel group";
```

Related Topics

samaccount

env

Description

Type **list** READONLY

env contains the list of environment variables configured in the environment where the submit user submitted the request.

Example

```
index=search(env, "APPL_HOME");
 if (index > -1)
 {
      aval=env[index];
      if (dirname(aval ) != "/usr")
      {
           printf("You are not permitted to run this application
from:%s\n",
                dirname(aval));
      }
 }
```

Related Topics

runenv

false

Description

Type **integer** READONLY

`false` contains the constant value 0.

Example

```
adminusers = {"dan","robyn","cory"};
if ((user in adminusers) == false)
   reject;
```

Related Topics

true

FEATURE_LDAP

Description

Type **integer** READONLY

Read-only constant used with the `feature_enabled()` function to determine whether LDAP features are available on a particular policy server.

Example

```
if (!feature_enabled(FEATURE_LDAP)
print("LDAP support is not available on this policy server");
```

Related Topics

FEATURE_VAS

FEATURE_VAS

Description

Type **integer** READONLY

Read-only constant used with the `feature_enabled()` function to determine whether Authentication Services features are available on a particular policy server.

Example

```
if  (!feature_enabled(FEATURE_VAS)
 print("Authentication  Services  support  is  not  available  on  this  policy
server");
```

Related Topics

FEATURE_LDAP

gid

Description

Type **integer** READONLY

gid contains the Group ID of the submitting user's primary group on the client host.

Example

```
adminusers = {"dan","robyn","cory"};
printf ("Request received from user id:%d %d\n", uid, gid);
```

Related Topics

uid

group

rungroup

group

Description

Type **string** READONLY

group contains the name of user's primary group.

Example

```
if (group == "admin")
   adminusers = append(adminusers,user);
```

Related Topics

groups

rungroup

rungroups

groups

Description

Type **string** READONLY

`groups` contains the list all groups in which the user is a member.

Example

```
# If a user belongs to a particular group, reject the command
 if ( "restrictedUsers" in groups )
 {
      reject;
 }
```

Related Topics

group

rungroup

rungroups

host

Description

Type **string** READONLY

`host` identifies the host name where the user has requested to run the command. The value is set to the host name selected by the `pmrun -h <hostname>` option, and defaults to `nodename`. You may expand it to a fully qualified name, if `shortnames` are not used.

Example

```
# If the requested host is not in the allowed_hosts list, reject the command
allowed_hosts = {“hosta.test.com”, “hostb.test.com”, “hostc.test.com”};
if ( host !in allowed_hosts )
{
   reject “Commands on host “ + host + “ are not allowed. \n”;
}
```

Related Topics

runhost

hour

Description

Type **integer** READONLY

`hour` contains the hour the request was submitted (0 – 23).

Example

```
if (hour == 12)
{
    // require the users password from 12:00 to 12:59
    if(!(userpasswd())
    reject;
}
accept;
```

Related Topics

dayname

minute

day

month

year

time

date

masterhost

Description

Type **string** READONLY

`masterhost` contains the host name of the host running `pmmasterd`.

Example

```
printf("Privilege Manager for Unix is authorizing your request on host:
%s\n",masterhost);
 accept;
```

masterversion

Description

Type **string** READONLY

`masterversion` contains the description of Privilege Manager for Unix policy server host.

Example

```
printf("Privilege Manager for Unix %s is authorizing your request on
host %s\n",
     masterversion, masterhost);
 accept;
```

minute

Description

Type **integer** READONLY

`minute` contains the minute the request was submitted (0-59).

Example

```
# display all commands run at 12:00 pm
pmlog -c '(hour==12) && (minute==0)'
```

Related Topics

dayname

hour

day

month

year

time

date

month

Description

Type **integer** READONLY

`month` contains the month number the request was submitted (0-11).

Example

```
if ( month == 11) && ( day == 25 )
 {
      printf ("Happy Christmas");
 }
```

Related Topics

dayname

minute

hour

day

year

time

date

nice

Description

Type **integer** READONLY

`nice` contains the value of the submit user session's `nice` value, that controls the execution priority. For more information, see the `nice` man pages.

Example

```
if ( nice == 019 )
 {
      printf("Warning: you have a very low scheduling priority");
 }
```

Related Topics

runnice

nodename

Description

Type **string** READONLY

`nodename` contains the host name of the client host.

Example

```
printf("Client on host %s \n", nodename
```

Related Topics

submithost

optarg

Description

Type **string** READONLY

`optarg` contains the parameter for the last argument or, if the option takes no argument, an empty string . Use with `getopt` functions.

opterr

Description

Type **boolean** READONLY

`opterr` determines whether to show errors from `getopt` functions.

optind

Description

Type **integer** READONLY

`optind` contains the current argument list index. Use with `getopt` functions.

optopt

Description

Type **string** READONLY

`optopt` contains the letter of the last option that had an issue. Use with `getopt` functions.

optreset

Description

Type **boolean** READONLY

When set to True, `optreset` restarts the `getopt` functions from the beginning. The next time a user calls a `getopt` function, `optind` will be set to 1.

optstrictparameters

Description

Type **boolean** READONLY

The `getopt_long()` function provides specific argument parameters. Arguments with optional parameters are accepted only when entered in the format `--argument=parameter`. For `getopt_long()` to recognize non-compliant forms, such as `--argument parameter`, set `optstrictparameters` to False.

pid

Description

Type **integer** READONLY

`pid` contains the process ID number of the `pmmasterd` process.

Example

```
printf("The pmmasterd process id is :%i", pid);
```

pmclient_type

Description

Type **integer** READONLY

The client type (pmrun or sudo) of the Privilege Manager for Unix request.

Example

```
# reject if pmclient_type is "sudo"
if (pmclient_type == pmclient_type_sudo) {
     reject;
} else if (pmclient_type == pmclient_type_pmrun) {
   ok = true;
}
```

Related Topics

pmclient_type_pmrun

pmclient_type_sudo

pmclient_type_pmrun

Description

Type **integer** READONLY

Read-only constant for pmrun type clients. You can compare `pmclient_type_pmrun` to `pmclient_type` to determine if the request was sent from a Privilege Manager for Unix client including the `pmrun` command, the pmshells (`pmksh`, `pmsh`, `pmcsh`, `pmbash`), and the `pmshellwrapper`.

Example

```
# reject if pmclient_type is "sudo"
if (pmclient_type == pmclient_type_sudo) {
     reject;
} else if (pmclient_type == pmclient_type_pmrun) {
     ok = true;
}
```

Related Topics

pmclient_type

pmclient_type_sudo

pmclient_type_sudo

Description

Type **integer** READONLY

Read-only constant for sudo type clients. You can compare pmclient_type_sudo to pmclient_type to determine if the request was sent from a Sudo Plugin client.

Example

```
# reject if pmclient_type is "sudo"
 if (pmclient_type == pmclient_type_sudo) {
      reject;
 } else if (pmclient_type == pmclient_type_pmrun) {
      ok = true;
 }
```

Related Topics

pmclient_type

pmclient_type_pmrun

pmshell

Description

Type **integer** READONLY

pmshell initializes to true if a Privilege Manager for Unix shell program (such as pmksh, pmsh, pmcsh, pmloginshell, and pmbash) is running; otherwise, the variable is undefined.

Example

```
if (defined pmshell)
{
   printf ("Now running: %s\n", pmshell_prog);
   pmshell_restricted = 1;
   pmshell_checkbuiltins = 1;
```

```
    pmshell_reject = "You are not allowed to run this command";
    pmshell_allow = {"ls","grep","cat"};
    pmshell_forbid = append(pmshell_forbid, "passwd");
    pmshell_forbid = append(pmshell_forbid, "kill");
}
else
{
    printf("Not running a command within %s\n", pmshell_prog);
    accept;
}
```

Related Topics

pmshell_restricted

pmshell_checkbuiltins

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_reject

pmshell_restricted

pmshell_builtin

Description

Type **integer** READONLY

`pmshell_builtin` is a constant value that identifies a shell `builtin` command. Use it to compare with the value of the `pmshell_cmdtype` variable.

Example

```
if (defined pmshell_cmd){
     if ((user in safe_shell_list) && (pmshell_cmdtype == pmshell_
builtin))
     {
          #allow all built-ins for selected users accept;
     }
 }
```

Related Topics

pmshell

pmshell_restricted

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_cmd

Description

Type **integer** READONLY

`pmshell_cmd` is only defined if the command is a Privilege Manager for Unix shell program (in which case it is set to `false`) or the command is a shell subcommand running from a Privilege Manager for Unix shell program (in which case it is set to `true`).

This variable is only applicable to the `pmsh`, `pmksh`, `pmcsh`, and `pmbash` programs.

Example

```
if (defined pmshell_cmd){
     if (user !in safe_shell_list)
     {
          #check builtins
          pmshell_checkbuiltins=true;
     }
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_cmdtype

Description

Type **integer** READONLY

`pmshell_cmdtype` is only defined if the command is a shell subcommand running from a Privilege Manager for Unix shell.

This variable is only applicable to the `pmsh`, `pmcsh`, `pmksh`, and `pmbash` programs.

It is set to one of these constant values: `pmshell_builtin`, `pmshell_script`, or `pmshell_exe`.

Example

```
if  (defined  pmshell_cmd){
     if  (user  !in  safe_shell_list)
     {
          #check  builtins
          pmshell_checkbuiltins=true;
     }
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_exe

Description

Type **integer** READONLY

`pmshell_exe` contains a constant value that identifies a normal executable command. Use it to compare with the value of the `pmshell_cmdtype` variable.

Example

```
if (defined pmshell_cmd){
   if (pmshell_cmdtype == pmshell_exe)
   {
      if (basename(runcommand) in shell_sub_list) {
         accept;
      }
   }
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_interpreter

Description

Type **integer** READONLY

`pmshell_interpreter` is only defined if the command is running from within a Privilege Manager for Unix shell program. If the shell subcommand is an interpreted script (that is, the first line of the file contains a directive in the format #!<*path*>) then this variable contains the pathname of the interpreter identified by this directive. Use this variable to detect and reject a user from running an unrestricted shell script from within a restricted shell program.

Example

```
if (defined pmshell)
{
   printf("Starting %s shell\n", pmshell_prog);
   accept;
}
if ((defined pmshell_cmd) && (pmshell_cmd == true))
{
```

```
# if running a restricted shell, then don't allow the user to run a shell
# script unless it's a Privilege Manager for Unix shell
if (pmshell_restricted && (pmshell_cmdtype == pmshell_script))
{
   if (dirname(pmshell_interpreter) != "/opt/quest/bin")
   {
      reject "Restricted shell only permits you to run a shell in the
                                    /opt/quest/bin directory";
   }
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_prog

Description

Type **string** READONLY

`pmshell_prog` is only defined if a Privilege Manager for Unix shell program is running. If a shell is running, it is set to the name of the shell program (`pmsh`, `pmcsh`, `pmksh`, `pmloginshell`, or `pmbash`).

Example

```
if (defined pmshell)
{
     printf("Starting %s shell\n", pmshell_prog);
     accept;
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_cmd

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_script

Description

Type **integer** READONLY

`pmshell_script` is a constant value that identifies a shell script. Use it for comparison with the value of the `pmshell_cmdtype` variable.

Example

```
if (defined pmshell_cmd && (pmshell_cmdtype == pmshell_script))
{
   #forbid any shell scripts unless interpreter is a program in /opt/quest/bin
   if (dirname (pmshell_interpreter) != "/opt/quest/bin"))
   {
      reject "You cannot run this script";
   }
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_uniqueid

Description

Type **string** READONLY

`pmshell_uniqueid` is only defined if the command is a shell subcommand running from a Privilege Manager for Unix shell (`pmsh`, `pmcsh`, `pmksh`, and `pmbash`). It contains the `uniqueid` of the session running the shell program. It allows the individual commands running within the shell to be identified as part of the same shell session when viewing the audit log entries.

Example

```
#shell script example to print out all shell commands for each shell run on
#15 january 2009

#constraint to select pmshell programs running on selected date
constraint="(date=\"2009/01/15\") && (pmshell==1) && (pmshell_cmd==0))"
```

```
#format to display user and shell program name
userformat="sprintf(\"User:%s, shell:%s\", user, pmshell_prog)"

#format to display shell subcommand name and time
shellformat="sprintf(\" Time:%s, ShellCommand:%s\n", time, runcommand)"

#find the unique IDs for all shell sessions
allids=`/bin/sh -c "pmlog -p 'sprintf(\"%s\", uniqueid)' -c '${constraint}'"`

#for each shell session, print out the username and shell program name,
#and display each shell command run from the shell, with the time it was
#executed for one in $allids
do
   cmd="pmlog -p '${userformat}' -c 'uniqueid==\"${one}\"'"
   /bin/sh -c "${cmd}"
   cmd="pmlog -p '${shellformat}' -c 'pmshell_uniqueid==\"${one}\"'"
   /bin/sh -c "$cmd"
done
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmversion

Description

Type **string** READONLY

`pmversion` contains the Privilege Manager for Unix version and build number.

Example

```
print("The current Privilege Manager for Unix version is %s", pmversion);
```

ptyflags

Description

Type **string** READONLY

`ptyflags` contains a `bitmask` indicating the `ptyflags` set from the submit user's environment. If set, the following bits indicate:

```
Bit 0: stdin is open
Bit 1: stdout is open
Bit 2: stderr is open
Bit 3: command was run in pipe mode
Bit 4: stdin is from a socket
Bit 5: command to be run using nohup
```

Example

```
PTY_IN=0x1;
if (ptyflags & PTY_IN)
{
   #only authenticate if stdin is open and password can be entered
   if (!authenticate_pam(user, "sshd"))
   {
      reject "Failed to authenticate user";
   }
}
else
{
   reject "Cannot authenticate the user"; }
```

Related Topics

runptyflags

requestlocal

Description

Type **integer** READONLY

Indicates if the request is local. requestlocal is true if no request was made to run on a remote host using pmrun -h.

Example

```
# reject requests to run on a remote host
if (requestLocal == false)
   reject "remote requests are not allowed";
```

requestuser

Description

Type **string** READONLY

requestuser is initialized to the selected user name if you select the pmrun –u option. It is a request to set the runuser for the session to the selected user name. The administrator can decide whether to honor the request in the policy file. By default, this variable is set to the value of the user variable.

Example

```
if ((user in adminusers) && (requestuser in adminusers_allowed))
 {
     runuser = requestuser;
 }
```

rlimit_as

Description

Type **string** READ ONLY

The `rlimit_as` variable controls the maximum memory that is available to a process.

Related Topics

runrlimit_as

rlimit_core

Description

Type **string** READ ONLY

The `rlimit_core` variable controls the maximum size of a core file.

Related Topics

runrlimit_core

rlimit_cpu

Description

Type **string** READ ONLY

The `rlimit_cpu` variable controls the maximum size CPU time of a process.

Related Topics

runrlimit_cpu

rlimit_data

Description

Type **string** READ ONLY

The `rlimit_data` variable controls the maximum size of data segment of a process.

Related Topics

runrlimit_data

rlimit_fsize

Description

Type **string** READ ONLY

The `rlimit_fsize` variable controls the maximum size of a file.

Related Topics

runrlimit_fsize

rlimit_locks

Description

Type **string** READ ONLY

The `rlimit_locks` variable control the maximum number of file locks for a process.

Related Topics

runrlimit_locks

rlimit_memlock

Description

Type **string** READ ONLY

The `rlimit_memlock` variable controls the maximum number of bytes of virtual memory that can be locked.

Related Topics

runrlimit_memlock

rlimit_nofile

Description

Type **string** READ ONLY

The `rlimit_nofile` variable controls the maximum number of files a user may have open at a given time.

Related Topics

runrlimit_nofile

rlimit_nproc

Description

Type **string** READ ONLY

The `rlimit_nproc` variable controls the maximum number of processes a user may run at a given time.

Related Topics

runrlimit_nproc

rlimit_rss

Description

Type **string** READ ONLY

The `rlimit_rss` variable controls the maximum size of the resident set (number of virtual pages resident at a given time) of a process.

Related Topics

runrlimit_rss

rlimit_stack

Description

Type **string** READ ONLY

The `rlimit_stack` variable controls the maximum size of the process stack.

Related Topics

runrlimit_stack

samaccount

Description

Type **string** READONLY

The user's sAMAccountName for the submit user if Authentication Services is configured and the client is able to determine the sAMAccountName. Otherwise this variable is set to an empty string.

Example

```
# reject if the user is not in the uxwheel AD group
if (vas_user_is_member(samaccount, "uxwheel", domainname) == false)
   reject "user is not in uxwheel group";
```

Related Topics

domainname

selinux

Description

Type **boolean** READONLY

`selinux` detects whether the client running pmrun or sudo is within an SELinux environment.

If SELinux is enabled on the client or policy host machine, it is True. If disabled, it is False.

status

Description

Type **integer** READONLY

`status` contains the exit status of the most recent command run by the system function.

Example

```
Cprofile=system("find /home/custom_appl -name customprofile.txt");
 if (status == 0)
 {
      welcome_msg=readfile(Cprofile);
      print(welcome_msg);
 }
```

submithost

Description

Type **string** READONLY

`submithost` contains the name of the host where the request was submitted.

Example

```
if ( submithost == "sun.34.com" )
 {
      reject;
 }
```

Related Topics

host

runhost

submithostip

Description

Type **string** READONLY

`submithostip` contains the IP address of the host where a request was submitted.

Example

```
if (  submithost  ==  "10.10.180.123")
 {
     reject;
 }
```

Related Topics

submithost

thishost

Description

Type **string** READONLY

The value of the `thishost` setting in the `pm.settings` file on the client. If you do not specify the `thishost` setting or if the client cannot resolve `thishost` to an IP address configured on the client, the variable remains undefined.

Example

```
# print a warning if thishost is not defined
if (!defined thishost)
   printf("WARNING: the thishost variable is not defined. \
   Please check the pm.settings file on host %s.\n", submithost);
```

Related Topics

host

runhost

submithost

time

Description

Type **string** READONLY

`time` contains the time the request was submitted in the form HH:MM:SS.

Example

```
printf("Command Started At Time: %s", time)
```

Related Topics

dayname

minute

hour

day

month

year

date

true

Description

Type **integer** READONLY

`true` is a read-only constant with a value of `1`.

Example

```
if  (iolog_encrypt  ==  true  )
 {
      iolog  =  mktemp("/var/adm/pm.enc."+user+"."+command+".XXXXXX");
 }
```

Related Topics

false

ttyname

Description

Type **string** READONLY

ttyname contains the name of the TTY device from which the user submitted a request.

Example

```
if ( ttyname == "dev/pts/1")
{
     printf("Command not authorized using tty device dev/pts/1");
     reject;
}
```

tzname

Description

Type **string** READONLY

Description

The time zone variable, tzname, contains the name of the time zone on the server at the time the event was read from the event log by pmlog. The time zone may be overridden using the TZ environment variable when running pmlog.

Note that tzname is accessible from pmlog but not in the policy script evaluation.

Example

```
# pmlog -p `sprintf("%s %s %s, %s, %s", date, time, tzname, event, uniqueid)’
2013-03-14 10:51:59 MDT, Accept, 0b1c7ff3447ac074b4795be2dcd59f6429c8624b
2013-03-14 10:51:59 MDT, Accept, a6cfad1ba6eb64bf9a17d5295b2bb29daa7fbb33
2013-03-14 10:51:59 MDT, Accept, fa742929679bc6c88eadd25ff85d75361f1d28b2
2013-03-14 10:51:59 MDT, Accept, 97ffdb433819c5feab6ec26b528f60dfb18c3d34
2013-03-15 07:02:47 MDT, Accept, d84ac9052265912eb13d32f80584d1ae097e4ce5
2013-03-19 09:41:59 MDT, Accept, b228110f32525c2092d2a46d0327e55f2dfc1d39
```

The actual values may vary by platform. In this sample output, the value of `tzname` is "MDT".

The following example shows the use of the `TZ` variable acting on the output:

```
TZ=Europe/Paris pmlog -p `sprintf( "%s %s %s, %s", date, time, tzname, event
)’
2013-03-14 17:51:59 CET, Accept, 0b1c7ff3447ac074b4795be2dcd59f6429c8624b
2013-03-14 17:51:59 CET, Accept, a6cfad1ba6eb64bf9a17d5295b2bb29daa7fbb33
2013-03-14 17:51:59 CET, Accept, fa742929679bc6c88eadd25ff85d75361f1d28b2
2013-03-14 17:51:59 CET, Accept, 97ffdb433819c5feab6ec26b528f60dfb18c3d34
2013-03-15 14:02:47 CET, Accept, d84ac9052265912eb13d32f80584d1ae097e4ce5
2013-03-19 16:41:59 CET, Accept, b228110f32525c2092d2a46d0327e55f2dfc1d39
```

Related Topics

date

time

uid

Description

Type **integer** READONLY

`uid` contains the user ID of the submitting user on the sudo host.

Example

```
printf("Req uest received from user id: %d %d\n", uid,gid);
```

Related Topics

gid

group

rungroup

umask

Description

Type **integer** READONLY

umask contains the value of the submit user's umask value. See the umask man page for details.

Example

```
if (umask == 077)
 {
     printf("Do not create files with permissions 0777\n");
     runumask =0666;
 }
```

Related Topics

runumask

unameclient

Description

Type **list** READONLY

unameclient contains the system uname information from the client host. This information corresponds to the list returned by uname. For example:

- operating system name
- nodename
- operating system release level

- operating system version
- machine hardware name

unamemaster

Description

Type **list** READONLY

`unamemaster` contains the system `uname` information from the policy serverclient host. This information corresponds to the list returned by `uname`. For example:

- operating system name
- nodename
- operating system release level
- operating system version
- machine hardware name

uniqueid

Description

Type **string** READONLY

`uniqueid` is a 12-character string identifying a session. This is guaranteed to be unique on one policy server machine.

Example

```
printf("Command is running as id = %s", uniqueid);
```

use_rundir

Description

Type **string** READONLY

`use_rundir` is a read-only variable containing the value "!~!". You can use it as a placeholder in the context of any runtime variable to represent the `runuser`'s home directory, as defined on the runhost. pmlocald replaces any instances of this value found in any runtime variable with the runuser's home directory on the runhost.

Example

```
allowedrequestusers={"root", "admin", "oradmin"};
//if requestuser is in allowed list, set runuser to requestuser
   and set groups to match primary group on the runhost,
//and change directory to runuser's home dir
if (requestuser in allowedrequestusers)
{
     runuser=requestuser;
     rungroup=use_rungroup;
     rungroups= {use_rungroup};
     runcwd = use_rundir;
     accept;
}
```

use_rungroup

Description

Type **string** READONLY

use_rungroup is a read-only variable containing the value "!g!". Use it as a placeholder in the context of any runtime variable to represent the runuser's primary group on the runhost. pmlocald replaces any instances of this value found in any runtime variable with the runuser's primary groupname on the runhost.

Example

```
allowedrequestusers={"root", "admin", "oradmin"};
//if requestuser is in allowed list, set runuser to requestuser
   and set groups to match runuser's primary group only,
//and change directory to runuser's home dir
if (requestuser in allowedrequestusers)
{
   runuser=requestuser;
```

```
  rungroup=use_rungroup;
  rungroups= {use_rungroup};
  runcwd = use_rundir;
  accept;
}
```

use_rungroups

Description

Type **sting** READONLY

use_rungroups is a read-only variable containing the value "!G!". Use it as a placeholder in the context of any runtime variable to represent the runuser's group list on the runhost. pmlocald replaces any instances of this value found in any runtime variable with the runuser's group list on the runhost.

Example

```
allowedrequestusers={"root", "admin", "oradmin"};
 //if requestuser is in allowed list, set runuser to requestuser
    and set groups to match those on the runhost, adding any
 //other run groups required, and change directory to runuser's home dir
 if (requestuser in allowedrequestusers)
 {
      runuser=requestuser;
      rungroup=use_rungroup;
      rungroups= {use_rungroups, "oraclegroup"};
      runcwd = use_rundir;
      accept;
 }
```

use_runshell

Description

Type **string** READONLY

use_runshell is a read-only variable containing the value "!!!". Use it as a placeholder in the context of any runtime variable to represent the runuser's login shell on the runhost. pmlocald replaces any instances of this value found in any runtime variable with the runuser's login shell on the runhost.

Example

```
allowedrequestusers={"root", "admin", "oradmin"};
 allowedscripts={"appscript1"};
 //Run a script as the runuser's login shell.
 //If requestuser is in allowed list, set runuser to requestuser, set
 //environment to match runuser's environement, add some necessary
 //environment vars for this script, and run the script as the runuser's
shell.

 if ((runcommand in allowedscripts) && (requestuser in
allowedrequestusers))
 {
      runuser=requestuser;
      rungroup=use_rungroup;
      rungroups= {use_rungroups, "appgroup"};
      runcwd = use_rundir;

      //use the runuser's environment
      profile_use_runuser=true;

      //add an application environment var to runuser's env, based on
runuser's
      //home dir
      str=sprintf("%s/appdir", use_rundir);
      setenv("APP_LOCAL_DIR", str);

      //Set the runcommand to use the runuser's shell to run the script
      runcommand = use_runshell;
      runargv=replace(runargv, 1, length(runargv));
      runargv[0]=use_runshell;
      runargv=append(runargv, "-c");
      runargv=append(runargv, "/appdir/appscript");
      accept;
 }
```

user

Description

Type **string** READONLY

user contaibnts the submit user's login name.

Example

```
If ( (user == "matt") && (command == "passwd") )
{
     printf("matt is not allowed to alter passwords");
     reject;
}
```

Related Topics

runuser

year

Description

Type **integer** READONLY

year contains the year in which the request was submitted in the format YY.

Example

```
if ( (year == "08") || (year == "12") )
{
     if ( (month == "01") && (day == "29") )
     {
          printf("This year is a leap year, something has gone wrong");
          reject;
     }
}
```

Related Topics

dayname

minute

hour

day

month

date

time

Global output variables

The following predefined global variables are initialized from the submit user's environment. They can be affected by the policy file and are used by `pmlocald` to set up the runtime environment for the `runcommand`.

Table 30: Global output variables

Variable	Data Type	Description
alertkeyaction	string	Action to be taken when `alertkeysequence` is matched.
alertkeysequence	list	List of patterns to match in a session.
disable_exec	integer	Specifies whether to prevent the `runcommand` process from executing new processes.
eventlog	string	Pathname of the audit log.
eventloghost	string	Host name list for remote event logging.
execfailedmsg	string	Message to display if `runcommand` cannot run.
iolog	string	Pathname of the keystroke log.
iolog_encrypt	integer	Specifies whether to encrypt the keystroke log.
iolog_errmax	integer	Max bytes to log for a `stderr` message.
iolog_opmax	integer	Max chars to log for a `stdout` message.
iologhost	string	Host name list for remote keystroke logging.
log_passwords	integer	Specifies whether to exclude passwords from the keystroke log.
logomit	list	Variables to omit from the audit and keystroke logs.
logstderr	integer	Specifies whether to keystroke log `stderr` messages.

Variable	Data Type	Description
logstdin	integer	Specifies whether to keystroke log `stdin` messages.
logstdout	integer	Specifies whether to keystroke log `stdout` messages.
notfoundmsg	string	Message to display if the `runcommand` is not found on the run host.
passprompts	list	Detects presence of password prompts.
pmshell_allow	list	Commands to allow in a Privilege Manager for Unix shell without further authorization.
pmshell_allowpipe	list	Commands to allow in a Privilege Manager for Unix shell without further authorization if input is from a pipe.
pmshell_check-builtins	integer	Specifies whether to authorize shell built-in commands in a Privilege Manager for Unix shell.
pmshell_forbid	list	Commands to forbid in a Privilege Manager for Unix shell without further authorization.
pmshell_readonly	list	Variables to mark as read-only in a Privilege Manager for Unix shell.
pmshell_reject	string	Reject message to display when a forbidden command runs in a Privilege Manager for Unix shell.
pmshell_restric-ted	integer	Specifies whether to run a Privilege Manager for Unix shell in restricted mode.
preserve_clienthost	integer	Specifies whether to use the originating login host name in preference to the submit host.
profile_keepenv	list	A list of values specified by the `keepenv()` call.
profile_setenv	list	A list of values specified by the `setenv()` call.
profile_unsetenv	list	A list of values specified by the `unsetenv()` call.
profile_use_runuser	string	Specifies whether to use the `runuser`'s environment rather than the submit user's environment
rejectmsg	string	Message to display when a session is rejected.
runargv	list	List of arguments for the request.
runbkgd	boolean	The run version of bkgd. When set to True, lets the user stop the pmrun call and move it to the background.
runchroot	string	Requests the command to run with a specified `root` directory.
runcksum	string	Identifies a checksum to use to verify against the

Variable	Data Type	Description
		runcommand.
runclienthost	string	A modifiable copy of the clienhost input variable.
runcommand	string	Full pathname of the request.
runconfirmuser	string	Specifies whether the agent should request the runuser to authenticate before executing the runcommand.
runcwd	string	Working directory to set for the request.
runenablerlimits	boolean	Lets you use runrlimit variables on the run host.
runenv	list	List of environment variables to set for the request.
rungroup	string	Primary group to set for the request.
rungroups	list	List of secondary groups to set for the request.
runhost	string	Host on which to run the request.
runnice	integer	Nice value to apply for the request.
runpaths	list	A list of permitted paths for commands.
runptyflags	string	Pty flags to apply for the request.
runrlimit_as	string	Controls the maximum memory that is available to a process.
runrlimit_core	string	Controls the maximum size of a core file.
runrlimit_cpu	string	Controls the maximum size CPU time of a process.
runrlimit_data	string	Controls the maximum size of data segment of a process.
runrlimit_fsize	string	Controls the maximum size of a file.
runrlimit_locks	string	Control the maximum number of file locks for a process.
runrlimit_memlock	string	Controls the maximum number of bytes of virtual memory that can be locked.
runrlimit_nofile	string	Controls the maximum number of files a user may have open at a given time.
runrlimit_nproc	string	Controls the maximum number of processes a user may run at a given time.
runrlimit_rss	string	Controls the maximum size of the resident set (number of virtual pages resident at a given time) of a process.
runrlimit_stack	string	Controls the maximum size of the process stack.
runtimeout	integer	Specifies the number of seconds of idle time before ending

Variable	Data Type	Description
		the session.
runumask	integer	Umask value to apply for the request.
runuser	string	User to run the request.
runutmpuser	string	Utmp user to use when logging to utmp.
subprocuser	string	User name to run subprocesses of the policy server master daemon.
tmplogdir	string	Directory used for temporary storage of I/O log files if a remote log host is specified in iologhost.

alertkeyaction

Description

Type **string** READ/WRITE

alertkeyaction contains the action to be taken if a command matches a pattern configured in alertkeysequence. The alertkeyaction can be defined as "reject", "log" or any custom string. The default value is "log".

Example

```
switch (user) {
     case "root" : alertkeyaction = "ignore"; break;
     default : alertkeyaction = "log"; break;
 }
```

Related Topics

alertdate

alertkeysequence

alertkeymatch

alerttime

alertkeysequence

Description

Type **list** READ/WRITE

alertkeysequence contains a list of regular expressions, against which pmlocald checks the standard input commands entered by the user during a session. If a match is found, then an alert is raised in the event log.

Example

```
Switch (user) {
     case "root": alertkeysequence={"passwd"};
          alertkeyaction="log";
          break;
     default : alertkeysequence={"passwd", "shutdown"};
          alertkeyaction="reject";
          break;
 }
```

Related Topics

alertdate

alertkeymatch

alertkeyaction

alerttime

disable_exec

Description

Type **integer** READ/WRITE

Use disable_exec to prevent the runcommand process from executing new UNIX processes. For example, you can prevent a vi session from executing shell commands. This variable is only supported if the underlying operating system supports the noexec feature; that is, Linux, Solaris, HP-UX, and AIX. If set to true(1), Privilege Manager for Unix sets the LD_PRELOAD environment variable, which causes the runcommand to be loaded with a Privilege Manager for Unix library that overrides the system exec functions, and thus prevents the runcommand from using exec to create a new process.

Example

```
if (basename(runcommand) in editor_program_list)
{
     disable_exec=true;
}
```

eventlog

Description

Type **string** READ/WRITE

`eventlog` contains the full pathname of the file in which audit events are logged. The default pathname is `/var/opt/quest/qpm4u/pmevents.db`.

Example

```
adminusers = {"dan","robyn","cory"}
if (user in adminusers)
   eventlog = "/var/log/pm+admin_eventlog_" + user + ".log";
else
   eventlog = "/var/opt/quest/qpm4u/pmevents.db";
```

Related Topics

eventloghost

event

Event logging

eventloghost

Description

Type **string** READ/WRITE

`eventloghost` is a string that defines the host that acts as a centralized event log server.

Example

```
eventloghost="sol32.test.com";
```

Related Topics

eventlog

event

execfailedmsg

Description

Type **string** READ/WRITE

If `execfailedmsg` is defined, this string sets the error message that displays if `pmlocald` fails to run `runcommand` for any reason other than the file not being found.

Example

```
if  (user  !=  "root")
 {
      execfailedmsg  =  "This  command  is  not  available  to  you  at  this
time";
 }
```

Related Topics

notfoundmsg

runcommand

iolog

Description

Type **string** READ/WRITE

iolog is the full path name of the keystroke log file in which input, output, and error output is logged.

Example

```
if (command in {"csh","ksh"})
{
     iolog_encrypt = true;
     log_passwords = false;
     iolog_errmax = 10000;
     iolog_opmax = 10000;
     iolog = mktemp("/var/adm/shells/pm." + user + "." + basename
(runcommand) + ".XXXXXX");
     accept;
}
     else
{
     iolog=mktemp("/var/adm/pm." + user + "." + basename(runcommand) +
".XXXXXX");
}
```

Related Topics

iologhost

iolog_opmax

iolog_errmax

iolog_encrypt

log_passwords

tmplogdir

Keystroke (I/O) logging policy variables

iolog_encrypt

Description

Type **integer** READ/WRITE

Set iolog_encrypt to true to encrypt the contents of the keystroke log. The pmreplay program decrypts the log before displaying it. The default value is false.

Example

```
if (command in {"csh","ksh"})
{
     iolog_encrypt = true;
     log_passwords = false;
     iolog_errmax = 10000;
     iolog_opmax = 10000;
     iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
     accept;
}
```

Related Topics

iologhost

iolog_opmax

iolog_errmax

log_passwords

iolog

Keystroke (I/O) logging policy variables

iolog_errmax

Description

Type **integer** READ/WRITE

`iolog_errmax` limits the number of bytes logged to the keystroke log for each line of `stderr` produced during the session.

Example

```
if (command in {"csh","ksh"})
{
   iolog_encrypt = true;
   log_passwords = false;
   iolog_errmax = 10000;
   iolog_opmax = 10000;
   iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
   accept;
}
```

Related Topics

iolog

iologhost

iolog_opmax

iolog_encrypt

log_passwords

Keystroke (I/O) logging policy variables

iolog_opmax

Description

Type **integer** READ/WRITE

`iolog_opmax` limits the size in bytes of each `stdout` keystroke log entry produced during the session.

Example

```
if (command in {"csh","ksh"})
 {
     iolog_encrypt = true;
     log_passwords = false;
     iolog_errmax = 10000;
     iolog_opmax = 10000;
     iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
     accept;
 }
```

Related Topics

iolog

iologhost

iolog_errmax

iolog_encrypt

log_passwords

Keystroke (I/O) logging policy variables

iologhost

Description

Type **string** READ/WRITE

`iologhost` is a string that defines the host that acts as a centralized I/O log server.

Example

```
iologhost="sol34.test.com";
```

Related Topics

iolog

iolog_opmax

iolog_errmax

iolog_encrypt

log_passwords

tmplogdir

log_passwords

Description

Type **integer** READ/WRITE

Set `log_passwords` to `false` to disable the keystroke logging of any password entry commands detected during the session. The default value is `true`.

Example

```
if (command in {"csh","ksh"})
{
     iolog_encrypt = true;
     log_passwords = false;
     iolog_errmax = 10000;
     iolog_opmax = 10000;
     loggroup = "admin";
     logstderr = true;
     logstdout = false;
     logstdin = true;
     iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
accept;
}
```

Related Topics

Keystroke (I/O) logging policy variables

logomit

Description

Type **list** READ/WRITE

`logomit` specifies a list of variable names to omit when logging to the keystroke and event log which can be useful if space is at a premium. For example, the administrator could choose to log only the `runenv` variable, and omit the submit `env` variable. The default is an empty list.

Example

```
logomit={ "nice" };
```

Related Topics

iolog

eventlog

Event logging

logstderr

Description

Type **integer** READ/WRITE

Set `logstderr` to `true` to enable keystroke logging of `stderr` output produced during the session. The default value is `true`.

Example

```
if (command in {"csh","ksh"})
 {
     iolog_encrypt = true;
     log_passwords = false;
     iolog_errmax = 10000;
     iolog_opmax = 10000;
     loggroup = "admin"; logstderr = true; logstdout = false;
          logstdin = true;
          iolog = mktemp("/var/adm/pm." + user + "." + command +
".XXXXXX");
     accept;
 }
```

Related Topics

logstdin

logstdout

Keystroke (I/O) logging policy variables

logstdin

Description

Type **integer** READ/WRITE

Set `logstdin` to `true` to enable keystroke logging of `stdin` input produced during the session. The default value is `true`.

Example

```
if  (command  in  {"csh","ksh"})
 {
      iolog_encrypt  =  true;
      log_passwords  =  false;
      iolog_errmax  =  10000;
      iolog_opmax  =  10000;
      loggroup  =  "admin";
      logstderr  =  true;
      logstdout  =  false;
      logstdin  =  true;
      iolog  =  mktemp("/var/adm/pm."  +  user  +  "."  +  command  +  ".XXXXXX");
      accept;
 }
```

Related Topics

logstderr

logstdout

Keystroke (I/O) logging policy variables

logstdout

Description

Type **integer** READ/WRITE

Set `logstdout` to `true` to enable keystroke logging of `stdout` output produced during the session. The default value is `true`.

Example

```
if (command in {"csh","ksh"})
{
   iolog_encrypt = true;
   log_passwords = false;
   iolog_errmax = 10000;
   iolog_opmax = 10000;
```

```
    loggroup = "admin";
    logstderr = true;
    logstdout = false;
    logstdin = true;
    iolog = mktemp("/var/adm/pm." + user + "." + command + ".XXXXXX");
    accept;
}
```

Related Topics

logstderr

logstdin

Keystroke (I/O) logging policy variables

notfoundmsg

Description

Type **string** READ/WRITE

`notfoundmsg` is set to the message that displays if the selected `runcommand` is not available on the target host.

Example

```
notfoundmsg = "Command \"" + runcommand + "\" not available.";
```

passprompts

Description

Type **list** READ/WRITE

`passprompts` contains a list of strings that should be interpreted as password prompts when attempting to exclude passwords from `iolog`.

Example

```
passprompts={"Password=", "Enter password"};
```

pmshell_allow

Description

Type **list** READ/WRITE

`pmshell_allow` contains a list of regular expressions identifying Privilege Manager for Unix shell subcommands that are pre-authorized. The list may contain regular expressions.

This variable is applicable to `pmsh`, `pmcsh`, `pmksh`, and `pmbash`.

On startup, the Privilege Manager for Unix shell programs load this list. Any shell subcommand entered by the user that matches one of these expressions is pre-authorized, that is, it will be allowed to run locally without any further authorization by `pmmasterd`, and will not be logged as an event. By default, the list is empty.

Example

```
pmshell_allow = {"ls","grep"};
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_forbid

pmshell_restricted

pmshell_allowpipe

Description

Type **list** READ/WRITE

`pmshell_allowpipe` identifies the list of Privilege Manager for Unix shell subcommands that are pre-authorized if the input to the command is from a pipe. The list may contain regular expressions.

This variable is applicable to `pmsh`, `pmcsh`, `pmksh`, and `pmbash`.

On startup, the Privilege Manager for Unix shells load this list. For any shell subcommand entered by the user that takes its input from a pipe, if the command matches one of these expressions, it will be allowed to run locally without any further authorization by the `pmmasterd`, and will not be logged as an event. By default, the list is empty.

For example, if this list contains the string "more", the "more" command will be pre-authorized in the context of the command `ls | more` but will require authorization in the context of the command `more /tmp/file`.

Example

```
pmshell_allow = {"grep","cat", "more"};
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_forbid

pmshell_restricted

pmshell_checkbuiltins

Description

Type **integer** READ/WRITE

If pmshell_checkbuiltins is set to true, the Privilege Manager for Unix shell program will check all shell builtin commands as if they were not built-ins. That is, it will match each one against the forbidden list, then the allowed list, and if no match is found, then the command will be authorized with pmmasterd. To see a full list of the builtin commands for a particular shell program, run the shell program with the -? option. The default value for this variable is false.

This variable is applicable to the pmsh, pmksh, and pmcsh programs.

Example

```
if (defined pmshell_cmd){
     if (user !in safe_shell_list)
     {
          #check builtins
          pmshell_checkbuiltins=true;
     }
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_forbid

Description

Type **list** READ/WRITE

pmshell_forbid contains a list of regular expressions loaded on startup by the Privilege Manager for Unix shell programs: pmsh, pmcsh, pmksh, and pmbash. The list may contain regular expressions.

Any command entered by the user during the shell session, that matches one of these expressions, will be forbidden without any further authorization by the pmmasterd, and will not be logged as an event. All shell subcommands are matched with this list before checking the allowed list. By default, the variable contains an empty list.

Example

```
pmshell_forbid = {"kill","passwd"};
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_restricted

pmshell_readonly

Description

Type **list** READONLY

`pmshell_readonly` is only defined if the command is a shell subcommand running from within a Privilege Manager for Unix shell program (`pmsh`, `pmcsh`, and `pmksh`). You can set this variable to a list of environment variables to mark as `readonly` in the shell. It defaults to an empty list.

Example

```
if (defined pmshell)
{
     #set some application specific readonly variables for the shell
     pmshell_readonly={"PATH", "SHELL", "APPL_HOME"};
}
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_reject

Description

Type **string** READ/WRITE

The `pmshell_reject` string is displayed by the Privilege Manager for Unix shell programs (`pmsh`, `pmcsh`, `pmksh`, and `pmbash`) for any shell subcommands rejected because they are listed in `pmshell_forbid`. The default is "Request Rejected".

Example

```
pmshell_reject = "Your request has been rejected by the shell";
```

Related Topics

pmshell

pmshell_restricted

pmshell_checkbuiltins

pmshell_cmd

pmshell_prog

pmshell_allow

pmshell_forbid

pmshell_restricted

pmshell_restricted

Description

Type **integer** READ/WRITE

If `pmshell_restricted` is set to `true`, then the Privilege Manager for Unix shell program is run as a restricted shell. This means that the user cannot:

- change directory
- change the PATH, SHELL, or ENV variables
- run any command that is not found in the PATH
- run any command identified by full pathname
- Overwrite any existing files using output redirection (such as, `echo "" > /etc/passwd`)

These restrictions are applied without any further authorization by the policy server. The default for this variable is `false`.

This variable is applicable to the `pmsh`, `pmcsh`, `pmksh`, and `pmbash` programs.

Example

```
if (user != "root")
{
      pmshell_restricted = true;
}
```

Related Topics

pmshell

pmshell_checkbuiltins

pmshell_cmd

pmshell_prog

pmshell_reject

pmshell_allow

pmshell_forbid

preserve_clienthost

Description

Type **integer** READ/WRITE

The `clienthost` variable normally matches the host name on which the `pmrun` client was run. To preserve the host name of the login host instead, set the `preserve_clienthost` variable to `true`.

Example

```
print("User has logged in from host:%s\n", clienthost);
```

profile_keepenv

Description

Type **list** READ/WRITE

A list of values specified by the `keepenv()` call. `profile_keepenv` tracks the values set from the latest `keepenv()` function call. Do not modify this variable directly; the `keepenv()` function updates this list.

Example

```
# add "HOME" to the keepenv list if not already in the list
if ("HOME" !in profile_keepenv)
   keepenv(append(profile_keepenv,"HOME"));
```

Related Topics

profile_setenv

profile_unsetenv

profile_setenv

Description

Type **list** READ/WRITE

A list of values specified by the `setenv()` call. `profile_setenv` tracks the values set from the latest `setenv()` function call. Do not modify this variable directly; the `setenv()` function updates this list.

Example

```
# setenv "HOME" to "/root" if not already in the list
if (search(profile_setenv,"HOME=*") == -1)
   setenv("HOME","/root");
```

Related Topics

profile_keepenv

profile_unsetenv

profile_unsetenv

Description

Type **list** READ/WRITE

A list of values specified by the `unsetenv()` call. `profile_unsetenv` tracks the values set from the latest `unsetenv()` function call. Do not modify this variable directly; the `unsetenv()` function updates this list.

Example

```
# unsetenv "HOME" if not already in the list
if ("HOME" !in profile_unsetenv)
   unsetenv("HOME");
```

Related Topics

profile_keepenv

profile_setenv

profile_use_runuser

Description

Type **string** READ/WRITE

The environment in which the `runcommand` runs is normally initialized from the submitting user's environment (that is, the `env` variable). To direct the `pmlocald` daemon to initialize the runtime environment using the `runuser`'s environment on the agent instead, set `profile_use_runuser` to `true`. The default value is `false`.

Note that `profile_use_runuser` causes the `runuser`'s dotfiles to run without an associated tty. The dotfiles (that is, `.profile`) should test for the existence of a `tty` (if `` `tty -s` ``) before a command runs that relies on a `tty` (for example, "who am i", "tset", "stty").

Example

```
if (defined pmshell)
 {
     profile_use_runuser=true;
 }
```

rejectmsg

Description

Type **string** READ/WRITE

`rejectmsg` contains the message that displays when a request is rejected.

Example

```
rejectmsg= "You are not permitted to run this command";
```

runargv

Description

Type **list** READ/WRITE

runargv specifies the complete argument list for the session. This variable is initialized from the value of the incoming argv variable.

Example

```
# Setting the runargv in the policy file can be used to add additional
 # command line arguments to programs
 if (command == "runTest")
 {
      runargv=replace(runargv,1,length(runargv));
      runargv=append(runargv, "-u", user };
 }
```

Related Topics

argv

runbkgd

Description

Type **boolean** WRITABLE

runbkgd determines whether a command is run in the background. If set to True, the command will ignore the SIGHUP (hangup) signal. This variable is initialized from the value of the incoming variable bkgd.

This variable does not affect commands run via sudo.

runchroot

Description

Type **string** READ/WRITE

runchroot emulates the behavior of the system chroot command; that is, it runs a command with a specified root directory. Ordinarily, file names are looked up starting at the root of the directory structure, ('/'). Setting runchroot to a different value changes the root directory, a directory that must exist.

Example

```
if  (basename(runcommand)  ==  "customapplication")
 {
      runchroot="/home/customapplicationv";
 }
```

runcksum

Description

Type **string** READ/WRITE

If runcksum is defined, pmlocald verifies the value of this variable against the checksum of the runcommand and rejects the request if it does not match. Set this variable to the value produced by running the pmsum command on the agent with the full pathname of the runcommand.

You can use this method to detect a program that has been changed without authorization, and a program that a user is attempting to run from an unauthorized path.

Example

```
# Generate a checksum value for the program "/usr/bin/passwd" on the
agent:host1
# for use in the policy file on the policy server.
pmsum /usr/bin/passwd

# The pmsum command displays the output:
fbc9cf01 /usr/bin/passwd

# Update the security policy using this checksum:
```

```
if (( basename(runcommand) == "passwd" ) && (host == "host1"))
{
   runcksum="fbc9cf01";
}
```

runclienthost

Description

Type **string** READ/WRITE

`runclienthost` is a modifiable copy of the `clienthost` input variable.

Example

```
# reject commands being issued from unknown workstations
workstations = {"sun34","sun35","sun36"};
if (!(clienthost in workstations))
   reject;
```

Related Topics

clienthost

runcommand

Description

Type **string** READ/WRITE

`runcommand` is a modifiable copy of the `command` input variable. It specifies the pathname of the program that `pmlocald` will run.

Example

```
Setting the runcomand can be a useful way of using a pseudonym for a
command that an
 auditor wants to disguise:
 if ( command == "passcmd")
 {
      runcommand="/usr/bin/passwd"
      runargv[0]="passwd";
      runargv=replace(runargv,1,length(runargv));
 }
```

Related Topics

command

runconfirmuser

Description

Type **string** READ/WRITE

Set `runconfirmuser` to a user name to direct `pmlocald` to request the `runuser` to authenticate as this user before running the `runcommand`. If authentication fails, then `pmlocald` rejects the session.

Example

```
if ( (user in appl_users) && (command in appl_cmds) )
 {
      runconfirmuser=runuser;
 }
```

Related Topics

runuser

runcwd

Description

Type **string** READ/WRITE

runcwd is a modifiable copy of the cwd input variable. Specifies the working directory for pmlocald to use when setting up the runtime environment for the session.

Example

```
if ( command in appl_cmds)
{
      runcwd = "/home/appl_home";
}
```

Related Topics

cwd

runenablerlimits

Description

Type **boolean** WRITABLE

runenablerlimits lets you use runrlimit variables on the run host. To enable the rlimit variables, runenablerlimits must be set to a value of True.

runenv

Description

Type **list** READ/WRITE

runenv is a modifiable copy of the env input variable. It contains a list of environment variables that pmlocald sets up when initializing the runtime environment for the session.

Example

```
if ( (command in appl_cmds) && (runhost == "sun8") )
 {
      runenv={"TERM=xterm","PATH=/usr/bin:/usr/local/bin", "HOME=/home/appl_
home"};
 }
```

Related Topics

env

rungroup

Description

Type **string** READ/WRITE

`rungroup` is a modifiable copy of the `group` input variable. It specifies the primary group for `pmlocald` to use when initializing the runtime environment.

Example

```
if ( (user == "apache") && (command == "admin.cgi") )
 {
      rungroup="root";
 }
```

Related Topics

groups

group

rungroups

rungroups

Description

Type **list** READ/WRITE

rungroups is a modifiable copy of the groups input variable. It specifies the full list of groups for pmlocad to use when initializing the runtime environment.

Example

```
if ( (user == "apache") && (command == "admin.cgi") )
{
     rungroups={"admin","operators"};
}
```

Related Topics

groups

group

rungroup

runhost

Description

Type **string** READ/WRITE

runhost specifies the host on which the runcommand will run.

Example

```
If ( command == "runSimulation" )
{
     runhost="sol34.test.com";
}
```

Related Topics

host

runnice

Description

Type **integer** READ/WRITE

`runnice` specifies the execution priority that `pmlocald` sets when initializing the runtime environment. (For more details, see the UNIX man pages for `nice`.)

Example

```
if ( timebetween(900,1630) )
 {
      runnice=010;
 }
      else
 {
      runnice=020;
 }
```

Related Topics

nice

runpaths

Description

Type **list** READ/WRITE

A list of permitted paths for commands. If configured, the agent rejects a command if it is not run from one of these paths, even if the command is authorized by the policy.

Example

```
# allow commands only from the /bin, /sbin, /usr/bin, and /usr/sbin
directories
runpaths={"/bin", "/sbin", "/usr/bin", "/usr/sbin"};
```

runptyflags

Description

Type **string** READ/WRITE

`runptyflags` is a modifiable copy of the `ptyflags` input variable. Use it to close `stdin` to prevent `stdin` on the runtime environment.

Example

```
if ( basename(runcommand) == "appl_home")
{
     # close stdin and prevent the user from providing any input
     # for a command that is only intended to be run in batch mode.
     runptyflags &= | 0x1;
}
```

Related Topics

ptyflags

runrlimit_as

Description

Type **string** WRITABLE

`runrlimit_as` is a modifiable copy of the `rlimit_as` input variable. It controls the maximum memory that is available to a process.

Related Topics

rlimit_as

runrlimit_core

Description

Type **string** WRITABLE

`runrlimit_core` is a modifiable copy of the `rlimit_core` input variable. It controls the maximum size of a core file.

Related Topics

rlimit_core

runrlimit_cpu

Description

Type **string** WRITABLE

`runrlimit_cpu` is a modifiable copy of the `rlimit_cpu` input variable. It controls the maximum size CPU time of a process.

Related Topics

rlimit_cpu

runrlimit_data

Description

Type **string** WRITABLE

`runrlimit_data` is a modifiable copy of the `rlimit_data` input variable. It controls the maximum size of the data segment of a process.

Related Topics

rlimit_data

runrlimit_fsize

Description

Type **string** WRITABLE

`runrlimit_fsize` is a modifiable copy of the `rlimit_fsize` input variable. It controls the maximum size of the data segment of a file.

Related Topics

rlimit_fsize

runrlimit_locks

Description

Type **string** WRITABLE

`runrlimit_locks` is a modifiable copy of the `rlimit_locks` input variable. It controls the maximum number of file locks for a process.

Related Topics

rlimit_locks

runrlimit_memlock

Description

Type **string** WRITABLE

`runrlimit_memlock` is a modifiable copy of the `rlimit_memlock` input variable. It controls the maximum number of bytes of virtual memory that can be locked.

Related Topics

rlimit_memlock

runrlimit_nofile

Description

Type **string** WRITABLE

`runrlimit_nofile` is a modifiable copy of the `rlimit_nofile` input variable. It controls the maximum number of files a user may have open at a given time.

Related Topics

rlimit_nofile

runrlimit_nproc

Description

Type **string** WRITABLE

`runrlimit_nproc` is a modifiable copy of the `rlimit_nproc` input variable. It controls the maximum number of processes a user may run at a given time.

Related Topics

rlimit_nproc

runrlimit_rss

Description

Type **string** WRITABLE

`runrlimit_rss` is a modifiable copy of the `rlimit_rss` input variable. It controls the maximum size of the resident set (number of virtual pages resident as a given time) of a process.

Related Topics

rlimit_rss

runrlimit_stack

Description

Type **string** WRITABLE

runrlimit_stack is a modifiable copy of the rlimit_stack input variable. It controls the maximum size of the process stack.

Related Topics

rlimit_stack

runtimeout

Description

Type **string** READ/WRITE

runtimeout specifies the number of seconds of idle time allowed before the session is closed.

Example

```
# close the session if the user is idle for 5 minutes
runtimeout=300;
```

runumask

Description

Type **integer** READ/WRITE

runumask is a modifiable copy of the umask input variable. Specifies the umask filter which determines file permissions for files created during execution of the runcommand.

Example

```
trustedusers = {"jamie", "cory", "robyn"};
 if (user in trustedusers )
 {
      runumask=066;
 }
```

Related Topics

umask

runuser

Description

Type **string** READ/WRITE

`runuser` is a modifiable copy of the `user` input variable. Specifies the user name that `pmlocald` uses when initializing the runtime environment for the `runcommand`.

Example

```
if ( (user == "apache") && (command == "admin.cgi") )
 {
      runuser="root";
 }
```

Related Topics

user

runutmpuser

Description

Type **string** READ/WRITE

`runutmpuser` specifies the login name of the user that will be used when updating the UNIX `utmp` and `wtmp` files when the request runs.

Example

```
if ( user == "djv" )
 {
      runutmpuser="dave";
 }
```

subprocuser

Description

Type **string** READ/WRITE

`subprocuser` is the user name used to run any subprocesses of `pmmasterd` such as, when running the system function. The default value is "root".

Example

```
subprocuser="appl_user";
 cfile=system("find /home/applhome -name customprofile.txt");
 if (status == 0)
 {
      print(readfile(cfile));
 }
```

tmplogdir

Description

Type **integer** READ/WRITE

`tmplogdir` is the directory used for temporary storage of I/O log files if a remote log host is specified in `iologhost`. The default value is `/opt/quest/qpm4u/iologs/queue`.

Example

```
iologhost="sol34.test.com";
 tmplogdir="/var/iologs/queue";
     iolog  =  mktemp("/var/adm/pm.enc."+user+"."+command+".XXXXXX");
 }
```

Related Topics

iolog

iologhost

iolog_opmax

iolog_errmax

iolog_encrypt

log_passwords

Global event log variables

The following predefined global variables appear only in the audit (event) log. They are not available for use in the policy file, as they are set by `pmlocald` during the `runcommand` session. Use `pmlog` to view them.

Table 31: Global event log variables

Variable	Data Type	Description
alertdate	string	Date on which the alert was raised.
alerttime	string	Time at which the alert was raised.
event	string	Type of event.
exitdate	string	Date on which the finish event was logged.
exitstatus	string	Exit status of the request
exittime	string	Exit time of the request.

alertdate

Description

Type **string** READONLY

`alertdate` contains the date when a configured alert was matched by `pmlocald`. It is not available for use in the policy file, it is set in the event log. To view the event log, use the `pmlog -l` command.

Example

```
#display all alerts raised with action set to log
pmlog –l -c 'alertkeyaction == "log"'
```

Related Topics

alertkeyaction

alertkeysequence

alertkeymatch

alerttime

alerttime

Description

Type **string** READONLY

`alerttime` contains the time when a configured alert was matched by `pmlocald`. It is not available for use in the policy file, it is set in the event log. To view the event log, use the `pmlog` command.

Example

```
#display all alerts raised after 6pm
pmlog –l –c 'alerttime > "18:00:00"'
```

Related Topics

alertkeysequence

alertkeymatch

alertkeyaction

alerttime

event

Description

Type **string** READONLY

`event` identifies the type of event logged by the policy server process. An event is logged when the policy server accepts or rejects a command. An event is also logged by the agent when a `runcommand` completes execution and an alert is raised.

Possible values are:

- Accept
- Reject
- Finish
- AlertRaised

This value is saved in the event log and can be viewed using `pmlog`.

Example

```
#Display all accepted events from the audit log
pmlog -c 'event == "Accept"'
```

Related Topics

eventlog

eventloghost

exitdate

Description

Type **string** READONLY

exitdate is the date the requested command finished running. This is saved in the event log when the session exits, and can be viewed using pmlog.

Example

```
#Display all events that finished on 15 january 2009
pmlog -c 'exitdate == "2009/01/15"'
```

Related Topics

exitstatus

exittime

exitstatus

Description

Type **string** READONLY

exitstatus contains the exit status of the runcommand. This variable is not available for use in the policy file. It is logged in the "Finish" event by pmlocald when the session ends.

Example

```
#Display all sh commands that failed to complete successfully
 pmlog –c 'runcommand == "sh" && exitstatus != "Command finished with
exit status 0"'
```

Related Topics

exitdate

exittime

exittime

Description

```
Type string READONLY
```

`exittime` is the time the requested command finished running (HH:MM:SS)

Example

```
#display all commands that finished after 6pm
pmlog -c 'exittime > "18:00:00"'
```

Related Topics

exitstatus

exitdate

PM settings variables

This section describes the settings and parameters used by Privilege Manager for Unix. These settings are stored on each host in the `/etc/opt/quest/qpm4u/pm.settings` file which contains a list of settings, one per line, in the form: settingName value1 [value2 [... value*n*]]. See Configuration prerequisites on page 122 to view a sample `pm.settings` file.

You can modify these policy server configuration settings using the configuration script initialized by the `pmsrvconfig` command or you can modify the `pm.settings` file manually. See Configuring the primary policy server for Privilege Manager for Unix on page 28 for details about running the configuration script.

If you manually change the `pm.settings` file, restart the `pmserviced` and/or `pmloadcheck` daemons in order for the changes to take effect.

The following table describes each of the `pm.settings` variables:

Defaults may differ depending on the platform you are configuring and whether you are configuring a policy server or PM Agent. Many of these settings will not have a default value.

The variables are not case sensitive.

Table 32: Variables: pm.settings

Variable	Data type	Description
certificates	boolean (YES/NO)	Specifies whether certificates are enabled. To enable configurable certification, add the following statement to the `/etc/opt/quest/qpm4u/pm.settings` file on each host: `certificates yes`. Default: NO For more information, see Enable configurable certification on page 146.
checksumtype	string	Specifies standard or MD5 checksum types for use with `pmsum` program.
clients	list of hostnames	Identifies hosts for which remote access functions are allowed. Only required if one policy server needs to retrieve remote information from another policy server that does not normally accept requests from it. For more information, see Central logging with Privilege Manager for Unix on page 157.
clientverify	string	Identifies the level of host name verification applied by the policy server host to the submit host name. The verification ensures that the incoming IP address resolves (on the primary policy server) to the same host name as presented by the submit host. Valid values are: • **none**: No verification performed. • **yes**: If a host name is presented for verification by the runclient it will be verified. • **All**: The policy server will only accept a request from a client if the host name is verified. Default: NONE
encryption	string	Identifies the encryption type. You must use the same encryption setting on all hosts in your system. Valid values are:

Variable	Data type	Description
		• AES • DES • TripleDES Default: AES
eventlogqueue	string	Directory used by `pmmasterd` and `pmlogsrvd` where event data is temporarily queued prior to being written to the event log database. Default: `/var/opt/quest/qpm4u/evcache`
EventQueueFlush	integer	Tells pmlogadm how often to reopen the db (in minutes) flushing the data. Default: 0, in which case pmlogsrvd will keep the db open while the service is running.
EventQueueProcessLimit	integer	Specifies the number of cached events that will be processed at a time; this limits the memory use in pmlogadm. Default: 0, in which case pmlogsrvd will not apply a limit.
facility	string	Sets the SYSLOG facility name to use when logging a message to the `syslog` file. Valid values are: • LOG_AUTH • LOG_CRON • LOG_DAEMON • LOG_KERN • LOG_LOCAL0 through LOG_LOCAL7 • LOG_LPR • LOG_MAIL • LOG_NEWS • LOG_USER • LOG_UUCP Default: LOG_AUTH, if the platform defines LOG_AUTH; otherwise the default is 0 (zero).
failovertimeout	integer	Sets the timeout in seconds before a connection attempt to a policy server is abandoned and the

Variable	Data type	Description
		client fails over to the next policy server in the list. This setting also affects the timeout for the client and agent. Default: 10 seconds. If omitted from `pm.settings`, default is 180 seconds.
failsafecommand	string	Sets the command to run in *failsafe* mode; that is, login `pmksh` user as `root`.
fwexternalhosts	list	Identifies a list of hosts to use a different range of source ports, identified by the `openreservedport` and `opennonreserved` port settings.
getpasswordfromrun	boolean (YES/NO)	Determines whether authentication is performed on the policy server or the client when a `getuserpasswd()` or `getgrouppasswd()` function is called from the policy file. If set to `yes`, the authentication is performed on the client. This variable also affects the user information functions: `getfullname()`, `getgroup()`, `getgroups()`, `gethome()`, and `getshell()`. If set to **yes** in the policy server's `pm.settings` file, these functions retrieve user information from the client host. Default: NO
handshake	boolean (YES/NO)	Enables the encryption negotiation handshake. This allows a policy server to support clients running different levels of encryption. Default: NO
kerberos	boolean (YES/NO)	Enables or disables Kerberos. Default: NO For more information, see Configuring Kerberos encryption on page 144.
keytab	string	Sets the path to the Kerberos keytab file. Default: `/etc/opt/quest/vas/host.keytab`
krb5rcache	string	Sets the path to the Kerberos cache. Default: `/var/tmp`
krbconf	string	Sets the path to the Kerberos configuration file.

Variable	Data type	Description
		Default: `/etc/opt/quest/vas/vas.conf`
libldap	string	Specifies the pathname to use for the LDAP library. No default value.
localport	integer	Sets the TCP/IP port to use for `pmlocald`. Default: 12346
lprincipal	string	Sets the service principal name to use for the agent. Default: pmlocald
masterport	integer	Specifies the TCP/IP port to use for `pmmasterd`. Default: 12345
masters	list	Identifies a list of policy server hosts to which a client can submit requests for authorization, and from which an agent can accept authorized requests. This can contain host names or netgroups. No default value.
mprincipal	string	Sets the Kerberos service principal name to use for the policy server. Default: host
nicevalue	integer	Sets the execution priority level for Privilege Manager for Unix processes. Default: 0
opennonreserveportrange	integer integer	Specifies a range of non-reserved ports to use as source ports when connecting to a host in the `fwexternalhosts` list. No default value.
openreserveportrange	integer integer	Specifies a range of reserved ports to use as source ports when connecting to a host in the `fwexternalhosts` list. No default value.
pmclientdenabled	boolean (YES/NO)	Flag that enables the `pmclientd` daemon.
pmclientdopts	string	Sets the options for the `pmclientd` daemon.

Variable	Data type	Description
pmlocaldenabled	boolean (YES/NO)	Flag that enables the `pmlocald` daemon.
pmlocaldlog	string	Sets the path for the agent error log. Default: `/var/adm/pmlocald.log` or `/var/log/pmlocald.log` depending on the platform. For more information, see Local logging on page 152.
pmlocaldopts	string	Sets the options for the `pmlocald` daemon.
pmloggroup	string	Specifies the group ownership for `iolog` and `eventlogs`. Default: pmlog
pmlogsrvlog	string	Identifies the log used by the `pmlogsrvd` daemon.
pmmasterdenabled	boolean (YES/NO)	Flag that enables the `pmmasterd` daemon. Default: YES
pmmasterdlog	string	Sets the path for the master error log. Default: `/var/adm/pmmasterd.log` or `/var/log/pmmasterd.log` depending on the platform. For more information, see Local logging on page 152.
pmmasterdopts	string	Sets the options for the `pmmasterd` daemon. Default: -ar
pmrunlog	string	Sets the path for the client error log. Default: `/var/adm/pmrun.log` or `/var/log/pmrun.log` depending on platform. For more information, see Local logging on page 152.
pmservicedlog	string	Identifies the log used by the `pmserviced` daemon. Default: `/var/log/pmserviced.log`
pmtunneldenabled	boolean (YES/NO)	Flag that enables the `pmtunneld` daemon.
pmtunneldopts	string	Sets the options for the `pmtunneld` daemon.

Variable	Data type	Description
policydir	string	Sets the directory in which to search for policy files Default: `/etc/opt/quest/qpm4u/policy`
policyfile	string	Sets the main policy filename. Default: `pm.conf`
policymode	string	Specifies the type of security policy to use, pmpolicy or Sudo. Default: sudo
reconnectagent	boolean (YES/NO)	Allows backwards compatibility with older agents on a policy server. Settings on policy server and agents must match. Default: NO
reconnectclient	boolean (YES/NO)	Allows backwards compatibility with older clients on a policy server. Settings on policy server and client must match. Default: NO
selecthostrandom	boolean (YES/NO)	Set to `yes` to attempt connections to the list of policy servers in random order. Set to `no` to attempt connections to the list of policy servers in the order listed in `pm.settings`. Default: YES
setnonreserveportrange	integer integer	Specifies a range of non-reserved ports to use as source ports by the client and agent. • Minimum non-reserved port is 1024. • Maximum non-reserved port is 31024. The full range for non-reserved ports is 1024 to 65535. For more information, see Restricting port numbers for command responses on page 142.
setreserveportrange	integer integer	Specifies a range of reserved ports to use as source ports by the client when making a connection to the policy server. • Minimum reserved port is 600. • Maximum reserved port is 1023.

Variable	Data type	Description
		The full range for reserved ports is 600 to 1023. For more information, see Restricting port numbers for command responses on page 142.
setutmp	boolean (YES/NO)	Specifies whether `pmlocald` adds a `utmp` entry for the request. Default: YES
shortnames	boolean (YES/NO)	Enables or disables short names usage. Setting `shortnames` to `yes` allows the use of short (non-fully qualified) host names. If set to `no`, then the Privilege Manager for Unix components will attempt to resolve all host names to a fully qualified host name. Default: YES
syslog	boolean (YES/NO)	Set to `yes` to send error messages to the syslog file as well as to the Privilege Manager for Unix error log. Default: YES For more information, see Local logging on page 152.
thishost	string	Sets the client's host name to use for verification. Specifying a `thishost` setting causes the Privilege Manager components to bind network requests to the specified host name or IP address. If you set `thishost` to the underscore character (_), requests bind to the host's primary host name. No default value.
tunnelport	integer	Sets the TCP/IP port to use for the `pmtunneld` daemon. Default: 12347 For more information, see Configuring pmtunneld on page 143.
tunnelrunhosts	list	Identifies the hosts on the other side of a firewall. No default value. For full details of how to configure your system across a firewall, see Configuring firewalls on

Variable	Data type	Description
		page 141.
validmasters	list	Identifies a list of policy servers that can be identified using the `pmrun -m <master>` option, but that will not be used when you run a normal `pmrun` command. This is useful for testing connections to a policy server before bringing it on line. No default value.

Appendix C

Privilege Manager for Unix Flow Control Statements

You can use the following reserved words to control the flow of logic in the pmpolicy file.

Table 33: Control flow reserved words

Statement	Description
accept, reject	Accept or reject the submitted request.
break	Break out of a `while` or for `loop`.
continue	Skip the rest of the loop body and continue to the next iteration of the loop.
do-while	Perform the loop body multiple times until an expression is `true`, evaluating the expression after running the statement.
for loop	c-style for loop.
for loop	Perform the loop body for each element in a list.
function	Stand-alone subroutine, allowing you to reuse policy.
if-else	Used to determine which statement to run next based on whether an expression is `true` or `false`.
include	Include the named policy file.
procedure / function	Stand-alone subroutine, allowing you to reuse policy.
readonly	Mark a variable as read-only.
readonlyexcept	Mark all variables as read-only except for the specified list.
return	Return from a function or procedure.
switch	Used to determine which statement to run next based on whether an expression matches one of several values.
while	Perform the loop body multiple times until an expression is `true`, evaluating the expression before running the statement.

accept, reject

Syntax

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

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

Description

The `accept` statement accepts the job request submitted by a user. The `reject` statement denies the request. After a command is accepted, nothing else in the configuration script is run. If neither an `accept` nor `reject` statement is reached while parsing the configuration file, the command is rejected by default. A default reject message is displayed to the user if no message is specified with the `reject` statement. If a null string is specified, then the command is rejected silently.

The expanded form of the `accept` and `reject` statements make it possible to accept or reject a command based on the criteria "who", "what", and "where" without using conditional statements.

Examples

```
adminusers = {"dan","robyn"};
adminprogs = {"hostname","kill","csh","ksh"};
if (user in adminusers && command in adminprogs)
{
   runuser = "root";
   if (user == "dan" && !officehours)
   {
      reject "You can't use “ + runcommand + “ outside office hours\n";
#custom msg

   }
   if (user == "robyn" && !officehours)
   {
      if (!getuserpasswd(user))
         reject ; #use default reject msg
   }
   accept;
}
else
{
   reject ""; #reject silently - no msg displayed to the user
}
```

break

Syntax

```
break;
```

Description

The `break` statement exits a loop and terminates cases. Use to force an immediate exit in **case** statements and looping statements such as `for`, `while`, and `do-while` statements.

Example

```
for ( oneuser in userlist )
 {
     if (oneuser == "root")
     {
          break;
     }
     print(oneuser);
 }
```

continue

Syntax

```
continue;
```

Description

Use the `continue` statement in the body of a C-style `for` loop, `while`, or `do-while` statement to skip the rest of the statements in the body of the loop and start again from the top of the loop.

Example

```
for ( oneuser in userlist )
 {
     if (oneuser == "root")
     {
         continue;
     }
         print(oneuser);
 }
```

do-while

Syntax

```
dostatement while ( expression ) ;
```

Description

The `do-while` statement is a looping statement. It repeatedly runs the specified *statement* until the specified *expression* evaluates to `false` (the value 0) or it encounters a `break` statement.

The specified *statement* runs at least once (unlike the `while` statement, which may terminate immediately).

Use a statement block in the form { *statement ...* } to run multiple statements in the loop. One Identity recommends using a statement block for readability.

Examples

This prints the values `1,2,3,4,5`:

```
x=1;
do print(x++); while (x <= 5);
```

This prints the values `1,2,3,4,5` using a statement block:

```
x = 1;
do {
   print(x);
   x++;
} while (x <= 5);
```

This prints the values `1,2,3` because the `break` statement terminates the loop:

```
x=1;
do {
   if (x > 3) break;
   print(x++);
} while (x <= 5);
```

for loop

Syntax

```
for ControlValue = StartValue to StopValue
[step increment] {
          initializer statements ;
          conditional expression ;
          update expression ;
          initializer statements ;
          conditional expression ;
     }
```

Description

The `for` statement is a looping statement. It runs one or more *initializer statements* and then evaluates the *conditional expression*. Use a comma to separate multiple *initializer statements*. If the *conditional expression* evaluates to `true` (any non-zero value), then it runs the specified *statement*. It runs the *update expression* (if present) immediately after it runs the specified *statement*. The `for` statement is terminated if the *conditional expression* evaluates to `false` (the value 0), or it encounters a `break` statement.

Typically, a `for` statement contains one *initializer statement*, a *conditional expression*, and an *update expression* that all operate on the same variable.

Use a statement block in the form { *statement ...* } to run multiple statements. One Identity recommends using a statement block for readability.

Examples

This prints the values `1,2,3,4,5`:

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

This prints the values `1,2,3,4,5`. (Note that this example does not have an *update expression* and it uses a statement block):

```
for  (x  =  1;  x  <=  5;  )  {
     print(x);
     x++;
 }
```

This prints the values `1,2,3` because the `break` statement terminates the loop:

```
for  (x  =  1;  x  <=  5;  x++)  {
     if  (x  >  3)  break;
     print(x);
 }
```

for loop

Syntax

```
for (variable in expression ) statement
```

Description

The `for` statement is a looping statement. The specified *expression* must be an array. It runs the specified *statement* once for each array element, and assigns it to the specified *variable* in turn. The `for` statement terminates when the specified *expression* does not evaluate to an array value, either when each element of the array has been iterated, or it encounters a `break` statement.

Use a statement block in the form { *statement ...* } to run multiple statements. One Identity recommends using a statement block for readability.

Examples

This prints the values 1,2,3,4,5:

```
for (x in {1,2,3,4,5}) print(x);
```

This does not print any value, since the expression does not evaluate to an array:

```
for (x in "foo") print(x);
```

This prints the values 1,2,3 because the break statement terminates the loop:

```
values = {1,2,3,4,5};
 for (x in values) {
      if (x > 3) break;
      print(x);
 }
```

function

Syntax

```
function ( parameter = expression, ... ) { statement ... }
```

Description

See procedure / function on page 304 for a full description of `function`.

if-else

Syntax

```
if ( expression ) statement
```

```
if ( expression ) statement else statement
```

Description

The `if-else` statement is a conditional statement. It runs the specified *statement* if the specified *expression* evaluates to `true` (a non-zero value). If the `else` part is present, it runs the associated *statement* if the *expression* evaluates to `false` (the value 0).

Use a statement block of the form { *statement ...* } to run multiple statements. One Identity recommends using a statement block for readability.

Examples

Accept if the user is contained in the set of trusted users, otherwise continue execution at the next statement:

```
trustedusers = {"jamie","corey","robyn"};
if (user in trustedusers)
   accept;
```

Accept if the user is contained in the set of trusted users, otherwise reject:

```
trustedusers = {"jamie","corey","robyn"};
if (user in trustedusers)
   accept;
else
   reject;
```

Note the use of statement block to handle multiple statements:

```
trustedusers  =  {"jamie","corey","robyn"};
 if  (user  in  trustedusers)  {
      print("accepted");
      accept;
 }  else  {
      print("rejected");
      reject;
 }
```

include

Syntax

```
include file-name
```

Description

The Privilege Manager for Unix configuration language contains the `include` statement, which is used to call out to other configuration files. By splitting your configuration file up into several smaller files, you can eliminate clutter. You can also hand-off control over certain aspects of configuration to different people, by giving them access to the subsidiary configuration files.

If an `accept` or `reject` is done within the included file, control never returns to the original file. On the other hand, if no `accept` or `reject` is done in the included file, execution will proceed to the end of that file, and then resume in the original file immediately after the `include` statement.

If a full pathname is not specified, the value of the `policydir` setting from the `/etc/opt/quest/qpm4u/pm.settings` file will be pre-pended to the filename.

When handing off control to a subsidiary configuration file whose contents are controlled by a questionable person, you might want to "fix" certain Privilege Manager for Unix variable values so that they cannot be changed by the subsidiary file. Use the `readonly` and `readonlyexcept` statements for this purpose.

As an example, you may have an Oracle® database administrator, who you want to be able to administer certain Oracle® programs. Each of those programs is to run as the "oracle" user. You would like the DBA to be able to grant or deny access to these programs and this account without your involvement, but you certainly do not want to give this person power over non-Oracle® parts of the system.

Specify the file to be included as a string expression; it may contain variables. For example, `include "/etc/ + usr + ".conf";`.

The following configuration file fragment hands off control to a subsidiary configuration file called, `/etc/pmorcle.conf`, and ensures that if an `accept` is done within this file, the job being accepted can only run as the oracle user.

Examples

```
oraclecmds = {"oradmin", "oraprint", "orainstall"};
 if(command in oraclecmds)
 {
      runuser = "oracle";
      readonly {"runuser"};
      include "/etc/pmoracle.conf";
      reject;
 }
```

procedure / function

Syntax

```
procedure parameter (argument-list)
{
statement ...
parameter = expression;
}
```

```
function parameter (argument-list)
{
statement ...
parameter = expression;
}
```

Description

A `procedure` is a named block of code that runs a sequence of one or more statements, and which may declare zero or more parameters. Each parameter is a variable that may optionally have a default value. If a parameter is declared with a default value, then all following parameters must also be declared with a default value. A procedure terminates when the final statement is run or when a `return` statement is run.

Variables and parameters declared within the procedure have local scope and are discarded when the procedure terminates. If an identifier is referenced within a procedure, the local scope of the procedure is checked first for a variable or parameter with a matching name. If one cannot be found, then the containing scope is checked for a variable with a matching name. If a matching variable still cannot be found, a new variable is declared, with a scope local to the procedure.

A procedure is invoked by specifying the name of the procedure and providing values for each parameter in a comma-separated argument list contained within parentheses. No argument is required if the matching parameter has a default value; in this case, the parameter will be assigned its specified default value.

A procedure may be declared using the `procedure` or `function` keywords. Historically, a `function` returns a value whereas a `procedure` does not; however, the parser will permit any procedure to return a value regardless of the keyword used. The choice of using the `procedure` or `function` keyword is stylistic. If a procedure ends without a return statement, a variable with the same name as the procedure is treated as the return value.

Examples

Procedure with no parameters:

```
procedure include_defaults() {
   include "/opt/quest/qpm4u/policies/defaults.conf";
}

include_defaults();
```

Procedure with two parameters, one of which has a default value:

```
procedure process_include_file(fname, fdir="") {
   topdir = "/opt/quest/qpm4u/policies";
   fpath = topdir + "/" + (fdir == "" ? "" : fdir + "/") + fname;
   if (fileexists(fpath)) {
      include fpath;
   }
}

process_include_file(user + ".conf");
         # default value of "" is assigned to parameter fdir

process_include_file(user + ".conf", "users");
         # parameter fdir is assigned the value "users"
```

Procedure with a parameter that masks a top-level variable with the same name. This print 1,2,1:

```
x = 1;

procedure foo(x) {
   print(x);
}

print(x);
foo(2);
print(x);
```

readonly

Syntax

```
readonly list
```

Description

Use the `readonly` statement to make a variable read-only. This means that its current value is frozen, so that no configuration file statement can change it. The purpose of this statement is to allow a system administrator to freeze the value of certain variables before calling out to another configuration file using the `include` statement. By safely freezing certain variable values, control over the other configuration file can safely be given to other, less-trusted personnel, knowing that they will not be able to abuse their privilege and gain unauthorized access to parts of the system that they should not be tampering with.

Examples

```
runuser = "jamie";
readonly {"runuser","runhost","runcommand"};
runuser = robyn;
print(runuser);
```

This policy will cause an execution error. Running `pmcheck` displays a message similar to this:

```
**Policy execution error in /etc/opt/quest/qpm4u/policy/pm.conf, line 3 Cannot
assign value to readonly identifier runuser
```

readonlyexcept

Syntax

```
readonlyexcept list
```

Description

The `readonlyexcept` statement is related to the `readonly` statement. The `readonlyexcept` statement makes all variables read-only, except those listed in the statement. The `readonlyexcept` statement has the same syntax as the `readonly` statement.

Examples

```
runhost = "myhost";
runuser = "jamie";
readonlyexcept {"runuser"};
runhost = "newhost"; // fails, runhost still equals "myhost"
runuser = "corey"; // runuser now equals "corey"
```

This policy will cause an execution error. Running `pmcheck` displays a message similar to this:

```
**Policy execution error in /etc/opt/quest/qpm4u/policy/pm.conf, line 3 Cannot
assign value to readonly identifier runuser
```

return

Syntax

```
return [expression];
```

Description

`return` exits the current procedure/function and returns the value of *expression*.

Examples

```
function square (n){
   n2 = n * n;
   return n2;
}

print(square(10)); // prints "100"
```

switch

Syntax

```
switch (string)
{
     case expression1:
          statement1a; [statement1b; …] [break;]
     case expression2:
          statement2a; [statement2b; …] [break;]
     default: statement3a; [statement3b; …] [break;]
}
```

Description

The switch statement tests whether an expression matches one of several values (each of which is specified in a case statement) and branches accordingly. If a case matches the value, execution will begin at that case falling through to subsequent cases until a break statement occurs. The break statement forces an immediate exit from the switch statement; it is optional.

The default statement runs if none of the cases match the value. This statement is optional. If there is no default and none of the cases match the value, nothing happens. Case statements can be in any order, but the default statement, if present, must occur after all of the case statements.

Examples

```
switch (user) {
   case "leslie":
      runuser="sys";
      break;
   case "adrian":
      accept;
   case "cory":
   case "jamie":
      runuser = "root";
      accept;
   default:
      reject;
}
```

```
switch (gidnum){
   case 0: runuser="root"; break;
   default: break;
}
```

See Example 9: Switch and case statements on page 133 for additional usage examples.

while

Syntax

```
while ( expression ) statement
```

Description

The `while` statement is a looping statement. It repeatedly runs the specified *statement* while the specified *expression* evaluates to `true` (any non-zero value). The `while` statement terminates when the specified *expression* evaluates to `false` (the value 0) or it encounters a `break` statement.

The specified *statement* may not run if the specified *expression* initially evaluates to `false` (unlike the `do-while` statement, which always runs its specified *statement* at least once).

Use a statement block in the form `{ statement ... }` to run multiple statements in the loop. One Identity recommends using a statement block for readability.

Examples

This prints the values `1,2,3,4,5`:

```
x = 1;
while (x <= 5) print(x++);
```

This prints the values `1,2,3,4,5` and uses a statement block:

```
x = 1;
 while (x <= 5) {
      print(x);
      x++;
 }
```

This prints the values `1,2,3` because the `break` statement terminates the loop:

```
x=1;
 while (x <= 5) {
      if (x > 3) break;
      print(x++);
 }
```

See Use the while loop on page 136 for more usage examples.

Appendix D

Privilege Manager for Unix Built-in Functions and Procedures

This section documents the syntax and usage of the built-in functions and procedures that are available to use within the policy file. They are listed in the following categories:

- Environment functions
- Hash table functions
- Input and output functions
- LDAP functions
- List functions
- Miscellaneous functions
- Password functions
- Remote access functions
- String functions
- User information functions
- Authentication Services functions

Environment functions

These are the built-in environment functions available to use within the policy file.

Table 34: Environment functions

Name	Description
getenv	Return the value of an environment variable in runenv.
getlistsetting	Return a list of the settings in the current policy server host settings file.
getnumericsetting	Return the integer of the numeric setting in the current policy server

Name	Description
	host settings file.
getstringsetting	Returns the value of a string setting in the current policy server host settings file.
getyesnosetting	Returns the value of a yes/no setting in the current policy server host settings file.
keepenv	Remove all except the specified variables from the runenv.
policygetenv	Set the value of the local variable to the value of the environment variable on the policy server.
policysetenv	Locally set the environment variable on the policy server host.
policyunsetenv	Locally unset an environment variable on the policy server.
setenv	Set a runtime environment variable.
unsetenv	Remove an environment variable from the runtime environment

getenv

Syntax

```
string getenv ( string name [, string value] )
```

Description

getenv returns the value of the specified environment variable from the runenv variable.

Example

```
# print the value of HOME if defined, otherwise print "none"
print(getenv("HOME", "none"));
```

Related Topics

keepenv

setenv

unsetenv

getlistsetting

Syntax

```
list getlistsetting ( string <variable_name>)
```

Description

`getlistsetting` returns a list of the settings in the pmpolicy server host settings file. If the named config is not present in the policy server host setting file, it returns an empty list.

Example

```
# get the master list setting
submitMasterList(getlistsetting("submitmasters"));
```

Related Topics

getstringsetting

getnumericsetting

getyesnosetting

getnumericsetting

Syntax

```
int getnumericsetting ( string <variable_name>)
```

Description

`getnumericsetting` returns the integer of the numeric setting in the pmpolicy server host settings file. If the named config is not present in the policy server host setting file, it returns zero.

Example

```
# get the value for master delay time
delayTime(getnumericsetting("masterdelay"));
```

Related Topics

getstringsetting

getlistsetting

getyesnosetting

getstringsetting

Syntax

```
string getstringsetting ( string variable_name)
```

Description

`getstringsetting` returns the value of a string setting in the pmpolicy server host settings file. If the named config is not present in the policy server host setting file, it returns an empty string.

Example

```
if  (getstringsetting("eventLogQueue")  ==  false  )  {
      reject;
 }
```

Related Topics

getnumericsetting

getlistsetting

getyesnosetting

getyesnosetting

Syntax

```
boolean getyesnosetting ( string <variable_name>)
```

Description

getyesnosetting returns the value of a yes/no setting in the current policy server host settings file. If the named config is not present in the policy server host setting file, it returns false.

Example

```
if (getyesnosetting("sysLogQueue") == false ) {
      reject;
 }
```

Related Topics

getstringsetting

getnumericsetting

getlistsetting

keepenv

Syntax

```
keepenv( string env1 [, string env2, …] )
```

Description

The keepenv procedure modifies the runenv variable to keep only those environment variables whose names are specified. All others are deleted from the runtime environment. This is used to constrain which environment variables a user may keep when running programs through Privilege Manager for Unix or Safeguard for Sudo when using the pmpolicy style policy.

Example

```
# reset the environment to the minimum
keepenv("PATH", "TERM", “HOME”, “USER”);
```

Related Topics

setenv

unsetenv

policygetenv

Syntax

```
string policygetenv ( string name [, string value] )
```

Description

`policygetenv` returns the value of the specified environment variable from the policy server.

Example

```
# print the value of HOME if defined, otherwise print "none"
print(policygetenv("HOME", "none"));
```

Related Topics

keepenv

setenv

policysetenv

Syntax

```
policysetenv ( string variable, string value )
```

Description

The `policysetenv` procedure sets one or more environment variables in the policy server.

Example

```
#set the shell variable
policysetenv("SHELL", "/opt/quest/bin/pmsh");
```

Related Topics

keepenv

unsetenv

policyunsetenv

Syntax

```
unsetenv( string env1 [, env2, …+ )
```

Description

The `policyunsetenv` procedure removes the named environment variable from the policy server.

Example

The following example deletes the `PAGER` and `EDITOR` environment variables from the policy server.

```
policyunsetenv("PAGER", "EDITOR");
```

Related Topics

keepenv

unsetenv

setenv

Syntax

```
setenv ( string name, string value )
```

Description

The `setenv` procedure sets one or more environment variables in the `runenv` variable.

Example

```
#set the shell variable
setenv("SHELL", "/opt/quest/bin/pmsh");
```

Related Topics

keepenv

unsetenv

unsetenv

Syntax

```
unsetenv( string env1 [, env2, …+ )
```

Description

The `unsetenv` procedure removes the named environment variable from the `runenv` variable.

Example

The following example deletes the `PAGER` and `EDITOR` environment variables from the runtime environment.

```
unsetenv("PAGER", "EDITOR");
```

Related Topics

keepenv

setenv

Hash table functions

These are the built-in hash table functions available to use within the policy file.

Table 35: Hash table functions

Name	Description
hashtable_add	Add a new list value to a hash table.
hashtable_create	Create a new hash table.
hashtable_enum	Enumerate entries in a hash table.
hashtable_import	Import a hash table from a file.
hashtable_lookup	Look up a value in a hash table.

hashtable_add

Syntax

```
int hashtable_add ( int hid, string key , list value)
```

Description

hashtable_add adds a new list value to the specified hash table, associated with the specified key.

Returns 0 if the hash table was successfully added, otherwise returns non-zero.

Example

```
hid=hashtable_create();
hashtable_add(hid, "unxadm", {"johnd", "davel", "jamesp"});
hashtable_add(hid, "winadm", {"marym", "stevec", "janel"});
print("Windows Admin Group:" + hashtable_lookup(hid, "winadm"));
```

Related Topics

hashtable_add

hashtable_import

hashtable_lookup

hashtable_create

Syntax

```
int hashtable_create ()
```

Description

hashtable_create creates a new hash table that can be used to store key-value pairs in a format that allows more efficient searching than an array.

Returns an identifier that you can use to add entries to and search the hash table.

Example

```
hid=hashtable_create();
hashtable_add(hid, "unxadm", {"johnd", "davel", "jamesp"});
hashtable_add(hid, "winadm", {"marym", "stevec", "janel"});
print("Windows Admin Group:" + hashtable_lookup(hid, "winadm"));
```

Related Topics

hashtable_import

hashtable_add

hashtable_lookup

hashtable_enum

Syntax

```
string hashtable_enum (int hid, [int reset])
```

Description

`hashtable_enum` returns the next entry in a hash table.

Example

```
hid=hashtable_create();
 hashtable_add(hid, "unxadm", {"johnd", "davel", "jamesp"});
 hashtable_add(hid, "winadm", {"marym", "stevec", "janel"});
 print("Windows Admin Group:" + hashtable_lookup(hid, "winadm"));
 for (x=hashtable_enum (hid,1); x!=""; x=hashtable_enum(hid,0)) {
      printf("Table contains key=%s\n", x);
 }
```

Related Topics

hashtable_import

hashtable_add

hashtable_lookup

hashtable_import

Syntax

```
int hashtable_import ( int hid, string filename )
```

Description

hashtable_import reads a specified file and uses the contents to create a hash table containing hash table entries, one per line, consisting of a single hash key, a colon, and a comma-separated list of hash values. The file may also contain comments delimited by the # character.

If successfully imported, it returns the number of entries in the hash table.

Example

```
#File admgroups.txt contains the formatted text
 unxadm:john,bob,fred,jane
 winadm:mary,chris,henry

 #policy loads this file into a hashtable that identifies the group
permissions,
 hid=hashtable_create();
 count=hashtable_import(hid, "/etc/opt/quest/qpm4u/tables/admgroups.txt");
 printf("Import loaded %d groups\n", count);
 unxadm=hashtable_lookup(hid, "unxadm");
 if (user !in unxadm)
 {
      reject "You are not authorized to run this command";
 }
```

Related Topics

hashtable_create

hashtable_add

hashtable_lookup

hashtable_lookup

Syntax

```
list hashtable_lookup ( int hid, string key)
```

Description

hashtable_lookup searches the specified hash table for the key.

If it finds the key, it returns the associated list, otherwise it returns an empty list.

Example

```
hid=hashtable_create();
hashtable_add(hid, "unxadm", {"johnd", "davel", "jamesp"});
hashtable_add(hid, "winadm", {"marym", "stevec", "janel"});
print("Windows Admin Group:" + hashtable_lookup(hid, "winadm"));
```

Related Topics

hashtable_create

hashtable_import

hashtable_add

Input and output functions

These are the built-in input and output functions available to use within the policy file.

Table 36: Input and output functions

Name	Description
fprintf	Write a string to a file on the policy server.
input	Request input from the user.
inputnoecho	Request input from the user without echoing to the screen.
print	Print a string to `stdout` with newline.
printf	Print a string to `stdout`.
printnnl	Print a string to `stdout` without newline.
printvars	Print the policy variables to `stdout`.
readdir	Return the list of entries in a directory as a string.
readfile	Read from a file on the policy server.
sprintf	Format a string.
syslog	Log a message to the syslog file.

fprintf

Syntax

```
fprintf ( string filename, string format [, string expression...] )
```

Description

The `fprintf` function is similar to `printf` except that the first argument is a filename. It appends the formatted string to the specified file.

For more information about formatting parameters, see the `printf(3)` man page.

Example

This example appends the string "End of file" to the `pmlog` file in the specified format.

```
fprintf("/var/adm/pmlog", "%s\n", "End of file";
```

Related Topics

printf

print

input

Syntax

```
string input( string prompt )
```

Description

`input` prompts the user to enter a single line of input and returns the entered string.

If the user enters a string, use the `atoi` function to convert the string to an integer.

Example

```
menu_selection = input("Enter your selection: ");
 switch(atoi(menu_selection)) {
   …
 }
```

Related Topics

atoi

inputnoecho

inputnoecho

Syntax

```
string inputnoecho( string prompt )
```

Description

`inputnoecho` prompts the user for a single line of input. The input is not displayed on the screen as it is typed.

Example

```
Instr = inputnoecho("Enter Selection: ");
 if (Instr in allowed_strs) {
   ….
 }
```

Related Topics

input

print

Syntax

```
print ( expression exp1 [, expression exp2, ...] )
```

Description

The `print` procedure prints the expression to `stdout` as a single line terminated with a newline character. If there is more than one argument, they are printed with a space delimiter. If there are no arguments, such as `print()`, the print result is a newline only. You can use variables, numbers, strings, lists or expressions as arguments in this function.

Example

```
print("Hello world");
```

Related Topics

fprintf

printf

printnnl

printvars

printf

Syntax

```
printf ( string format [, expression exp1, ...] );
```

Description

The `printf` procedure prints a formatted string to `stdout`.

For more information about formatting parameters, see the `printf(3)` man page.

Example

```
#this prints " 10" with no newline.
printf("%4d", 10);

#this prints "cory" preceded by 16 blank spaces, terminated with a newline.
user="cory";
printf("%-20.20s\n", user);
```

Related Topics

fprintf

print

printnnl

printvars

printnnl

Syntax

```
printnnl ( expression expr1 [, expression expr2, ...] )
```

Description

The `printnnl` procedure is similar to the `print` function except that it does not terminate the output with a newline character.

Related Topics

fprintf

print

printf

printvars

printvars

Syntax

```
printvars( );
```

Description

The `printvars` procedure prints all Privilege Manager for Unix variables to the user’s screen. It is useful for debugging configuration file policies.

Related Topics

fprintf

print

printf

printnnl

readdir

Syntax

```
string readdir ( string path [, string filter] )
```

Description

`readdir` reads the contents of the directory identified by `path`, and returns the list of files as a string. If you supply a `filter`, it applies a glob-style filter and only returns those files that match the `filter` in the string. If you do not supply a fully qualified `path`, it assumes the `path` is relative to the `path` identified by the `policyDir` setting in the `pm.settings` file.

Example

```
#find all *.profile files in the profiles directory and include any found
incfiles=readdir("profiles", "*.profile");
incfile_list=split(incfiles);
for onefile in incfile_list {
   include onefile;
}
```

Related Topics

keepenv

setenv

unsetenv

readfile

Syntax

```
string readfile ( string filename )
```

Description

The `readfile` function reads the contents of the specified file and returns the contents as a single string. Note that any new lines in the file will be present in the string returned by `readfile`. If the file does not exist, it rejects the session and produces a syntax error.

Example

```
#print a welcome msg from a file in /etc/
x=readfile("/etc/custom_welcome.txt");
print (x);
```

Related Topics

input

sprintf

Syntax

```
string sprintf ( string format [, expression expr, ...])
```

Description

The `sprintf` function returns a formatted string.

For more information about formatting parameters, see the `printf(3)` man page.

Example

```
printf("User= %-8.8s Application: %s\n", user, app);
```

Prints the same as:

```
a=sprintf("User= %-8.8s Application: %s", user, app);
print(a);
```

syslog

Syntax

```
syslog ( string format [, expression expr, ...])
```

Description

`syslog` sends a formatted message to `syslog` as a `LOG_INFO` message.

For more information about configuring `syslog` messages, see the `syslog(3)` man page.

Example

```
syslog("Accepted request from %s@%s", user, submithost);
```

LDAP functions

These are the built-in LDAP functions available to use within the pmpolicy file.

Table 37: LDAP functions

Name	Description
ldap_ bind	Bind an LDAP connection to the given credentials.
ldap_count_ entries	Count the number of entries returned by `ldap_search`.
ldap_dn2ufn	Convert a DN to a user-friendly format.
ldap_explode_dn	Return the elements of a DN.
ldap_first_ attribute	Obtain the first attribute in an LDAP entry.
ldap_first_entry	Obtain the first entry returned by `ldap_search`.
ldap_get_attrib- utes	Return all attribute names in an LDAP entry.
ldap_get_dn	Return the DN of an entry.
ldap_get_values	Return a list of the values for an attribute.
ldap_next_ attribute	Return the next attribute in an LDAP entry.
ldap_next_entry	Return the next entry returned by `ldap_search`.
ldap_open	Open a connection to an LDAP server.
ldap_search	Search the LDAP directory.
ldap_unbind	Close the LDAP connection.

ldap_ bind

Syntax

```
int ldap_bind(integer ldapid, string userdn [, string password [, boolean trace]] )
```

Description

`ldap_ bind` binds an LDAP connection to the specified credentials. The LDAP ID must be a valid LDAP connection ID returned by `ldap_open`. You can require an optional password.

If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

If successful, it returns `0`; otherwise it returns non-zero or an undefined variable.

Example

```
rc=ldap_bind(ldapid, "cn=admin", "Secretpassword");
 if ((!defined rc) || (rc != 0))
 {
      reject "Bind to ldap directory failed";
 }
```

ldap_count_entries

Syntax

```
int ldap_count_entries(int ldapid, ldapresult searchresult[, boolean trace] )
```

Description

ldap_count_entries returns the number of LDAP entries found by a previous call to ldap_search.

If the optional trace parameter is set to true, any errors or warnings from the LDAP function are written to stdout.

Example

```
# search for all Users at base level
 searchresults= ldap_search( ldapid, 'ou=Users,dn=ldap,dn=domain,dn=com',
      'onelevel', '(objectClass=*)' );
 if (ldap_count_entries(ldapid, searchresults) == 0)
 {
      reject "Found no users";
 }
```

Related Topics

ldap_dn2ufn

ldap_dn2ufn

Syntax

```
string ldap_dn2ufn(string dnstr[, boolean trace])
```

Description

`ldap_dn2ufn` converts a DN formatted string to a more user friendly format returned as a string.

If the optional `trace` parameter is set to `true`, any errors and warnings from the LDAP function are written to `stdout`.

Example

```
ufn=ldap_dn2ufn("uid=jsmith,ou=Users,dn=directory,dn=ourdomain,dn=com");
print(ufn);

#prints the output:
#jsmith, Users, directory, ourdomain, com
```

Related Topics

ldap_explode_dn

ldap_explode_dn

Syntax

```
list ldap_explode_dn(string dnstr [, boolean noTypes[, boolean trace]] )
```

Description

`ldap_explode_dn` returns a list of strings composed of the elements of the specified DN. If the optional `noTypes` parameter is set to `true`, the types are stripped from the exploded values. The default for `noTypes` is `false`.

If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

Example

```
dnlist=ldap_explode_dn
("uid=jsmith,ou=Users,dn=directory,dn=ourdomain,dn=com");
stripped=ldap_explode_dn
("uid=jsmith,ou=Users,dn=directory,dn=ourdomain,dn=com");
print(dnlist);
print(stripped);

#prints the following output
#{ uid=jsmith ou=Users dn=directory dn=ourdomain dn=com}
#{jsmith, Users, directory, ourdomain, com}
```

Related Topics

ldap_first_attribute

ldap_first_attribute

Syntax

```
string ldap_first_attribute(int ldapid, ldapentry entry[, boolean trace] )
```

Description

`ldap_first_attribute` returns the first attribute name in the `ldapentry` returned by a previous call to `ldap_first_entry` or `ldap_next_entry`.

If not present, returns an empty string. If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

Example

```
str=ldap_first_attribute(ldapid, entry);
while (length(str) > 0) {
    #process attribute
    …
    str=ldap_next_attribute(ldapid, entry);

}
```

Related Topics

ldap_get_attributes

ldap_first_entry

Syntax

```
int ldap_first_entry(int ldapid, ldapresult, result[, boolean trace] )
```

Description

`ldap_first_entry` returns the first entry from the list of results returned by `ldap_search` if present, otherwise returns an empty entry.

If the optional `trace` parameter is set to `true`, any errors and warnings from the LDAP function are written to `stdout`.

Example

```
entry=ldap_first_entry(ldapid,  searchresults);
 while(  entry)  {
       func_process_entry(entry);
       entry=ldap_next_entry(ldapid,  entry);
 }
```

Related Topics

ldap_get_attributes

ldap_get_attributes

Syntax

```
list ldap_get_attributes(int ldapid, ldapentry entry[, boolean trace] )
```

Description

`ldap_get_attributes` returns the full list of attribute names in an `ldapentry` returned by a previous call to `ldap_first_entry` or `ldap_next_entry`.

If none are present, it returns an empty list. If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

Example

```
allattributes=ldap_get_attributes(ldapid, entry);
 if (selected_attribute in allattributes) {
      #process matching attribute
 }
```

Related Topics

ldap_get_dn

ldap_get_dn

Syntax

```
string ldap_get_dn(int ldapid, ldapentry entry[, boolean trace])
```

Description

`ldap_get_dn` returns the DN of the specified entry, as a string. `ldapentry` is a valid entry returned by a previous call to `ldap_first_entry` or `ldap_next_entry`.

If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

Example

```
dnstr=ldap_get_dn(ldapid,entry;
```

Related Topics

ldap_get_values

ldap_get_values

Syntax

```
list ldap_get_values(int ldapid, ldapentry entry, string attr[, boolean trace] )
```

Description

`ldap_get_values` returns a list of values for the specified attribute from the given LDAP entry.

If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

Example

```
values=ldap_get_values(ldapid, entry, "uid");
 if (user !in values) {
      reject "You are not authorized";
 }
```

Related Topics

ldap_next_attribute

ldap_next_attribute

Syntax

```
string ldap_next_attribute(int ldapid, ldapentry entry[, boolean trace])
```

Description

ldap_next_attribute returns the next attribute name in the ldapentry returned by a previous call to ldap_first_entry or ldap_next_entry.

If the optional trace parameter is set to true, any errors or warnings from the LDAP function are written to stdout.

Example

```
str=ldap_first_attribute(ldapid, entry);
 while (length(str) > 0) {
      #process attribute
      …
      str=ldap_next_attribute(ldapid, entry);
 }
```

Related Topics

ldap_next_entry

ldap_next_entry

Syntax

```
int ldap_next_entry(int ldapid, ldapentry entry[, boolean trace] )
```

Description

ldap_next_entry returns the next entry from the series of results returned by ldap_search, if present; otherwise it returns a NULL or empty entry.

If the optional trace parameter is set to true, any errors or warnings from the LDAP function are written to stdout.

Example

```
entry=ldap_first_entry(ldapid, searchresults);
 while( entry) {
      func_process_entry(entry);
      entry=ldap_next_entry(ldapid, entry);
 }
```

Related Topics

ldap_open

ldap_open

Syntax

```
ldapid ldap_open( string host [, int port [, boolean trace]] )
```

Description

`ldap_open` opens a connection to the LDAP server on the specified `host` (identified by hostname or IP address) and `port` number. The default port number is 389. Use the returned LDAP connection ID as the first parameter to the other LDAP functions.

If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

If successful, it returns a valid LDAP connection ID; otherwise it returns an undefined variable.

The `ldap_open` library function has been deprecated in the open LDAP libraries. If supported by the installed LDAP library, the `ldap_open` policy function calls `ldap_initialize` in preference to `ldap_open`. However, `ldap_initialize` does not open the connection - the connection is opened by the first operation attempted, so `ldap_initialize` will succeed even if given an invalid host name. The `ldap_open` policy function displays the loaded LDAP library path if a value of `1` is passed as the *trace* parameter to `ldap_open`. This makes it easier to determine which LDAP library is used.

Example

```
ldap = ldap_open( 'ldap.host' );
 if( !defined ldap ){
      reject "Connection to LDAP server failed" ;
 }
```

ldap_search

Syntax

```
ldapresult ldap_search(int ldapid, string basedn, string scope, string filter [,
list attrList [, int attrOnly[, boolean trace]]] )
```

Description

ldap_search performs a search in the LDAP directory starting at the location identified by basedn. The ldapid is a valid connection ID returned by ldap_open.

The optional attrList parameter is the list of attributes to return in the results. This defaults to an empty list. The filter contains the LDAP search string, in the format described in RFC 4526.

The optional attrOnly parameter is a true or false value. When true, the results contain only the attribute; when false the results return attributes and values. Default setting is true.

Possible search scope:

- "base" - returns only the entry specified at the DN specified by *basedn*.
- "onelevel" - returns all matching entries from the next level down the directory.
- "subtree" - returns all matching entries below the *basedn* in the tree.

If the optional trace parameter is set to true, any errors or warnings from the LDAP function are written to stdout.

Returns a special type ldapresult containing the results of the search in a format that you can pass to the ldap_first_entry and ldap_next_entry functions.

Example

```
#search for all Users at base level
 searchresults= ldap_search( ldapid, "ou=Users,dn=ldap,dn=domain,dn=com",
      'onelevel', '(objectClass=*)' );
 if (ldap_count_results(ldapid, searchresults) == 0)
 {
      reject "Found no users";
 }
```

ldap_unbind

Syntax

```
ldap_unbind (int ldapid[, boolean trace] )
```

Description

`ldap_unbind` closes the LDAP connection and frees all associated resources. The `ldapid` must be a valid LDAP connection returned by `ldap_open`.

If the optional `trace` parameter is set to `true`, any errors or warnings from the LDAP function are written to `stdout`.

Example

```
ldapid = ldap_open( 'ldap.host' );
 if( defined ldapid ){
      rc=ldap_bind(ldapid, "cn=admin", "Secretpassword");
      if ((defined rc) && (rc == 0)){
           rc=func_search_for_user(ldapid);
           ldap_unbind(ldapid);
      }
 }
```

LDAP API example

The pmpolicy language supports the use of LDAP calls to obtain data on the following platforms:

- all versions of Linux on x86 supported by Privilege Manager for Unix
- all versions of Linux on x86-64 supported by Privilege Manager for Unix
- Solaris SPARC® 6 and above
- AIX 5.2 and above
- HP-UX PA-RISC 11 and above

The pmpolicy LDAP functions follow, as closely as possible, the API outlined in RFC 1823 to ensure compatibility and ease of understanding.

The `feature_enabled()` function indicates whether the LDAP functions are available on a particular policy server.

The following example illustrates the use of the LDAP functions.

```
if (!feature_enabled(FEATURE_LDAP) {
   print("LDAP support is not available on this policy server");
} else {
   ld_user = "cn=Directory Manager";
   ld_passwd = "password";
   ld_host = "ldapserver";
   BASEDN="ou=People,dc=skynet,dc=local";
   SCOPE="onelevel";
   FILTER="(objectClass=*)";
   ATTRLIST={};
   ATTRONLY=false;

   print( "LDAP Server: " + ld_host );
   print( "    User DN: " + ld_user );
   print( "   Password: " + ld_passwd );
   print( "" );
   print( "    Base DN: " + BASEDN );
   print( "      Scope: " + SCOPE );
   print( "     Filter: " + FILTER );
   print( "" );

   # Open a connection to the directory server
   ldapid = ldap_open( ld_host );
   if( ldapid < 0 ) {
      print( "ldap_open failed" );
      reject;
   }
   # bind to the directory
   rc = ldap_bind( ldapid, ld_user, ld_passwd );
```

```
if( rc==0 ) {
   # perform the search
   ld_results = ldap_search( ldapid, BASEDN, SCOPE, FILTER, ATTRLIST, ATTRONLY );
   if( ld_results >= 0 ) {
      # how many results have been returned?
      num = ldap_count_entries( ldapid, ld_results );
      str = sprintf( "Num results = %d", num );
      print(str);
      print("");
      print("RESULTS");
      print("");
      if( num>0 ) {
         # Grab the first entry from the results
         lentry = ldap_first_entry( ldapid, ld_results );
         while( lentry ) {
            # print the DN
            dn = ldap_get_dn( ldapid, ld_results );
            print("---- START OF ENTRY (" + dn + ") ----");
            e = ldap_explode_dn( dn );
            print( "               Exploded DN: " + join( e, ', ' ) );
            e = ldap_explode_dn( dn, 1 );
            print( "Exploded DN, no type names: " + join( e, ', ' ) );
            print( "               User Friendly form: " + ldap_dn2ufn( dn ) );
            print("");
            oc = ldap_get_values( ldapid, lentry, "objectClass" );
            if( "inetorgperson" in oc ) {
               gn = ldap_get_values( ldapid, lentry, "givenname" );
               sn = ldap_get_values( ldapid, lentry, "sn" );
               print( "  Found a person, Name = " + gn[0] + " " + sn[0] );
            }

            attrs = ldap_get_attributes( ldapid, lentry );
            print( "Attributes: " + join(attrs, ", ") );
            # Move through each attibute for the entry
            attr = ldap_first_attribute( ldapid, lentry );
            while( attr != '' ) {
               print(" ATTR: " + attr );
                  # Print the values for the given attribute
                  values = ldap_get_values( ldapid, lentry, attr );
                  print( "  VALUES = { " + join(values, ", ") + " }" );

                  # move to the next attibute
                     attr = ldap_next_attribute( ldapid, lentry );
            }
            # move to the next entry
            lentry = ldap_next_entry( ldapid, ld_results );
            print("---- END OF ENTRY (" + dn + ") ---- ");
            print("");
            }
            print("");
         }
         print("-- END OF RESULTS --");
      }
   } else {
      print( "ldap_bind failed" );
      reject;
   }

   rc = ldap_unbind( ldapid );
   str = sprintf( "rc = %d", rc );
   print(str);
}
```

Related Topics

feature_enabled on page 357

List functions

These are the built-in list functions available to use within the pmpolicy file.

Table 38: List functions

Name	Description
append	Append to a list.
insert	Insert a string or list into a list.
join	Concatenate a list into a string.
length	Return the length of a string, list, or array.
lsubst	Substitute part of a string with another string throughout all or part of a list.
range	Select a range of entries in a list.
replace	Replace one or more strings in a list.
search	Search a list for a string.
split	Convert a string into a list.
splitSubst	Convert a string into a list.

append

Syntax

```
list append( list dest, list|string src1 [, list|string src2, ...])
```

Description

append creates a list constructed by appending the specified strings or lists src1, src2, etc. to the end of the list dest and returns a new list.

Example

```
trustedusers = {"jamie", "cory", "robyn"};
a = append(trustedusers, "adrian");
```

sets a to the following list:

```
{"jamie", "cory", "robyn", "adrian"}
```

Related Topics

insert

join

insert

Syntax

```
list insert( list dest, int index, string src1, [, string src2, ...] )
```

Description

insert constructs a list by inserting strings into a list at the specified position. Note that the first element in the list is index: 0. If the index is greater than the length of the specified list (for example, 999), then the strings append to the end of the list.

Returns the newly constructed list.

Example

```
trustedusers={"jamie", "cory", "robyn"};
a=insert(trustedusers, 1, "leslie");
```

sets `a` to the list:

`{"jamie", "leslie", "cory", "robyn"}`

Related Topics

append

join

join

Syntax

```
string join( list X [, string delimiter] )
```

Description

`join` returns a string constructed by concatenating each element of list X. Each element of the string is separated by `delimiter`. The default *delimiter* is a space character.

Example

```
trustedusers={"jamie", "cory", "robyn"};
print(join(trustedusers, "\n"));
```

Prints the following string:

```
jamie
 cory
 robyn
```

Related Topics

append

insert

length

Syntax

```
int length( list|string X )
```

Description

`length` returns the number of elements in the specified list or the number of characters in the specified string.

Example

```
trustedusers={"jamie", "cory", "robyn"};
print(length(trustedusers));
```

lsubst

Syntax

```
string lsubst( list X, string pattern, string replacement )
```

Description

`lsubst` substitutes part of a string with another string throughout all or a specified part of a list X.

Example

```
print(lsubst({"xxxonexxx","xxxonexxx"},"one","two"));

#prints the following list
#"{xxxtwoxxx,xxxtwoxxx}"
```

range

Syntax

```
list range( list X, int begin, int end )
```

Description

The `range` function returns a subset of the elements from list `X`. The subset of elements in the range specified by `begin` and `end`. Any value for `end` greater than the length of the list is the same as `end`.

Example

```
trustedusers={"jamie", "cory", "robyn"};
a=range(trustedusers, 1, 2);
```

The value of `a` is set to: `{"cory", "robyn"}`

replace

Syntax

```
list replace( list X, int start, int end [, string s1, ...])
```

Description

The `replace` function deletes the elements between the `start` and `end` indices of the specified list and inserts the supplied strings in their place. If you do not specify any replacement string values, it replaces those elements with nothing; that is, it returns the list with the specified portion omitted.

Example

```
trustedusers={"jamie", "cory", "robyn"};
a=replace(trustedusers, 1, 1, "sandy");
print(a); // prints "{jamie, sandy, robyn}"
```

search

Syntax

```
int search( list X, string pattern)
```

Description

The `search` function returns the index of the first matching instance of `pattern` in the specified list. If there is no match, it returns -1.

The first element in the list is index:0.

Example

The following example prints the index number for "cory", which is 1:

```
a=search({"jamie","cory","robyn"},"c*"); print(a);
```

Table 39: Search patterns

j*	j followed by any number of characters.
j*e	j followed by any number of characters, ending with an e.
[jJ]*	Upper or lower case j followed by any number of characters.
[a-z]	Any lower case character.
[^a-z]	Any character except lower case characters.
j followed by a single character.	

split

Syntax

```
list split ( string X [, string delimiter] string omit_empty_elements )
```

Description

The `split` function is the opposite of `join`. It constructs a list by concatenating the strings into a list. It separates each element in the list with a delimiting character, which can be any character from the `delimiter` string. The default for `delimiter` is any white space character.

A sequence of two or more contiguous delimiter characters in the parsed string is considered to be a single delimiter. Delimiter characters at the start or end of the string are ignored.

The `omit_empty_elements` argument defaults to true. If specified and is false, the empty elements are not omitted from the resulting list.

Example

The following example returns the list: `{"jamie", "cory", "robyn"}`

```
a = split( "jamie, cory, robyn", ", ")
```

Related Topics

splitSubst

splitSubst

Syntax

```
list splitsubst( string X, string delimiter )
```

Description

The `splitsubst` function splits a string *X* into a list. This function is similar to the `split` function except that the delimiter contains the entire delimiter string.

Example

The following example returns the list: `"john","jane,james"`

```
a = splitsubst( "john,,jane,james", ",," )
```

Related Topics

split

Miscellaneous functions

These are the built-in miscellaneous functions available to use within the pmpolicy file.

Table 40: Miscellaneous functions

Name	Description
atoi	Translate a string representation of an integer to an integer.
authenticate_pam	Authenticate a user on the primary policy server.
authenticate_pam_toclient	Authenticate a user on the client.
basename	Return the filename portion of a path.
comparehosts	Check whether a host string matches a host definition.
datecmp	Compare two date strings.
dirname	Return the directory name portion of a path
feature_enabled	Determine whether a feature is supported on the policy server
fileexists, access	Check whether a file or path exists on the policy server.
getopt	Examine a list of arguments for short options to break up command lines for easier parsing.
getopt_long	Examine a list of arguments for short or long options to break up command lines for easier parsing.
getopt_long	Examine a list of arguments for only long options to break up command lines for easier parsing.
glob	Match a string to a pattern.
ingroup	Check whether a host is in the specified UNIX group on the policy server.
innetgroup	Check whether a user is in the specified NIS netgroup on the policy server.
innetuser, inuser-netgroup	Check whether a user is in the NIS netgroup or specified netgroup on the policy server.
lineno	Return the current line number in the policy file.
mktemp	Create a temporary file. Same as `mktemp system`.

Name	Description
osname	Return a string representation of the operating system.
quote	Quote a string.
rand	Generate a random number.
stat	Obtain information about a file on the policy server.
strftime	Format the current date/time as a string.
system	Run a program on the policy server.
timebetween	Check whether a given time is between two times.
tolower	Convert string to lower case.
toupper	Convert string to upper case.
uname	Return system information on the policy server; output of `uname` system command line.

atoi

Syntax

```
int atoi ( string nptr )
```

Description

`atoi` converts the string representation of a decimal integer to an integer. If the string does not contain a number, it produces a syntax error and rejects the session.

This function returns the converted integer.

Example

```
x=atoi("123");
printf("%d\n", x);
```

Returns: 123

Related Topics

insert

join

authenticate_pam

Syntax

```
int authenticate_pam ( string user [, string service] )
```

Description

The `authenticate_pam` function authenticates a user by means of the PAM (Pluggable Authentication Method) APIs on the policy servers.

For more information on how to configure PAM, consult the documentation for your platform.

The service parameter identifies the name of the PAM service to use to authenticate the user. This can be any valid service name configured in the PAM system configuration. It defaults to the PAM service "login".

This function returns 0 to indicate failure and 1 to indicate success.

Example

```
if (user=="paul" && basename(command)=="useradd")
{
     if (!authenticate_pam(user, "sshd"))
     {
          reject;
     }
     runuser="root";
     accept;
}
```

Related Topics

authenticate_pam_toclient

Utilizing PAM authentication

authenticate_pam_toclient

Syntax

```
int authenticate_pam_toclient ( string user [, string service] )
```

Description

The `authenticate_pam_toclient` function authenticates a user by means of the PAM (Pluggable Authentication Method) APIs on the policy server.

For more information on how to configure PAM, consult the documentation for your platform.

The service parameter identifies the name of the PAM service to use to authenticate the user. This can be any valid service name configured in the PAM system configuration. It defaults to the PAM service "login".

This function returns 0 to indicate failure and 1 to indicate success.

Example

```
if (user=="paul" && basename(command)=="useradd")
 {
     if (!authenticate_pam_toclient(user, "sshd"))
     {
          reject;
     }
     runuser="root";
     accept;
     }
```

Related Topics

authenticate_pam

Authenticate PAM to client

basename

Syntax

```
string basename ( string pathname )
```

Description

`basename` returns the filename portion of a pathname. It does not check that either the filename or path exist.

Example

```
print(basename("/var/adm/pm.log"));
```

Returns: "pm.log"

Related Topics

dirname

comparehosts

Syntax

```
int comparehosts(hoststring, hostpattern)
```

Description

`comparehosts` checks whether a host string (either host name or IP string) matches a host definition, which could be a host name (such as, `host1.a.b.com`), IP address (such as, `10.10.10.1`), netgroup (such as, `@mygroup1`), host pattern (such as, `*.a.b.com`) or IP address (such as, `10.10.10.*`).

This function honors the value of short names defined in `pm.settings` when resolving host names.

Returns 1 if a match is found, 0 if no match is found.

Example

```
if (comparehosts(submithost,"*.a.b.com"))
    {
         ...
    }
```

datecmp

Syntax

```
int datecmp(date1, date2)
```

Description

`datecmp` compares the two dates, which must be in the format `YYYY/MM/DD` or `YY/MM/DD` (in which case 2000 is added to the year).

This function returns these values:

- -1: date1 < date2
- 1: date1 > date2
- 0: date1 = date2

Example

```
if (datecmp(startdate, enddate) >=0)
    {
         reject “startdate must be before enddate”;
    }
```

dirname

Syntax

```
string dirname ( string pathname )
```

Description

`dirname` returns the directory portion of a pathname. It does not check that the filename or path exist.

Example

```
print(dirname("/var/adm/pmlog"));
```

Returns: `"/var/adm"`

Related Topics

basename

feature_enabled

Syntax

```
int feature_enabled (int feature )
```

Description

`feature_enabled` checks whether a particular feature is enabled on the policy server. Use this function to detect support for platform-dependant features; currently these comprise FEATURE_LDAP and FEATURE_VAS (defined as integer constants).

Returns `true` if the feature is enabled, otherwise `false`.

Example

```
if  (feature_enabled(FEATURE_LDAP))
 {
     if  (proc_do_ldap_authentication(user))
     {
          accept;
     }
 }
```

fileexists, access

Syntax

```
int fileexists ( string path )
```

```
int access ( string path )
```

Description

`fileexists` or `access()` determines whether the file `fn` or `path` exists on the policy server. Returns `true` if the path name exists, `false` if not.

Example

```
if (fileexists("/opt/quest/pmc") ) {
      print ("PMC is installed.");
 }
```

```
if (access("/opt/quest/pmc") ) {
      print ("PMC is installed.");
 }
```

Related Topics

access

getopt

Syntax

```
int getopt ( string argv, string optstring))
```

Description

`getopt` breaks up command lines for easier parsing and legal review. It examines a list of arguments for short options, which is a dash followed by a single letter or parameter.

Example

```
while ((option = getopt(args, "vh")) !=""){
print("Matched option",option);
}
```

Related Topics

getopt_long

getopt_long_only

optarg

optind

optopt

optreset

optstrictparameters

getopt_long

Syntax

```
int getopt_long ( string argv, string optstring, string long_options))
```

Description

getopt_long breaks up command lines for easier parsing and legal review. It examines a list of arguments for short or long options.

The function works in posixly correct mode and does not reorder arguments. However, if you unset the POSIXLY_CORRECT environment option, it reorders the argv variable as it scans, placing all nonoptions at the end of the list.

Example

```
while ((option = getopt_long(args, "vh",{"verbose","help"})) !=""){
print("Matched option",option);
}
```

Related Topics

getopt

optarg

opterr

optind

optopt

optreset

optstrictparameters

getopt_long_only

Syntax

```
int getopt_long_only ( string argv, string optstring, string long_options))
```

Description

getopt_long breaks up command lines for easier parsing. It examines a list of arguments for only long options.

The function works in posixly correct mode and does not reorder arguments. However, if you unset the POSIXLY_CORRECT environment option, it reorders the argv variable as it scans, placing all nonoptions at the end of the list.

Example

```
while ((option = getopt_long_only(args, "vh",{"verbose","help"})) !=""){
print("Matched option",option);
}
```

Related Topics

getopt

getopt_long

optarg

opterr

optind

optopt

optreset

optstrictparameters

glob

Syntax

```
int glob ( string pattern, string str )
```

Description

glob matches a string to a pattern. This match is often used for filenames since the patterns are the same ones that the UNIX shell uses for filename matching.

For more information, see the fnmatch(3) man page.

Returns true if the string matches the pattern, otherwise false.

Example

```
#this returns true because the “*” wildcard character matches any number of
any character
glob("a*b", "axyzb")

#this returns true because the “.” Is interpreted as a literal period char.
glob("a.*b", "a.fgb")
```

Table 41: Search patterns

j*	j followed by any number of characters.
j*e	j followed by any number of characters, ending with an e.
[jJ]*	Upper or lower case j followed by any number of characters.
[a-z]	Any lower case character.
[^a-z]	Any character except lower case characters.
j followed by a single character.	

ingroup

Syntax

```
int ingroup ( string user, string group )
```

Description

ingroup returns true if the specified user is in the specified UNIX group on the policy server; otherwise returns false.

Example

```
if (ingroup("cory", "admin") ) {
     accept;
}
```

Related Topics

innetgroup

innetgroup

Syntax

```
int innetgroup ( string netgroup, string host )
```

Description

innetgroup returns true if the specified host is in the specified NIS netgroup on the policy server; otherwise returns false.

Example

```
if ( ! innetgroup("submithosts", submithost)) {
     reject "You are not permitted to submit a command from this host";
}
```

innetuser, inusernetgroup

Syntax

```
int innetuser (string netgroup, string user)
```

```
int inusernetgroup (string netgroupname, string username)
```

Description

`innetuser` or `inusernetgroup` returns `true` if the specified user is in the specified NIS netgroup or other specified group on the policy server; otherwise the function returns `false`.

Example

```
if ( ! innetuser("submitusers", user)) {
     reject "You are not permitted to submit a command from this host";
}
```

```
if ( ! inusernetgroup("submitusers", user)) {
     reject "You are not permitted to submit a command from this host";
}
```

Related Topics

innetgroup

lineno

Syntax

```
int lineno( )
```

Description

`lineno` returns the current line number in the policy file.

Example

```
printf("TRACE: user:%s, cmd:%s, lineno:%d\n", user, command, lineno());
```

mktemp

Syntax

```
string mktemp ( string template )
```

Description

`mktemp` returns a unique filename which is guaranteed not to exist on the policy server. Use the `mktemp` function to create unique temporary filenames.

For more information, see the `mktemp(3)` man page.

Example

```
#generate a unique filename-the XXXXXX chars will be replaced to construct a
unique name
filename=mktemp("/tmp/pmXXXXXX");
print(filename); // prints "/tmp/pmAxK2de"
```

osname

Syntax

```
string osname( )
```

Description

`osname` returns an internal string representation of the operating system on the policy server, such as `aix43-rs6k`, `linux-x86_64`.

Example

```
printf("Policy server is running on OS:%s\n", osname());
```

quote

Syntax

```
string quote( string str [, string esc[, string surrounding_string]] )
```

Description

The `quote` function puts the specified string between quotation marks. It inserts the "\" (backslash) character as required to "quote" any occurrences of the characters in the second argument to indicate that they are taken literally. The string is surrounded by a "surrounding_string" and defaults to the value of `esc`, which is optional and defaults to the value of the specified escape character. UThe `quote` function is useful when parsing arguments into commands which are shell scripts. The default escape character is a single quote.

Example

```
#this function will return: [This won\'t fail.]
quote("This won't fail.", "'");
```

rand

Syntax

```
int rand(int max )
```

Description

`rand` returns a random number less than the specified maximum.

Example

```
# print a random item from a list
print(alist[rand(length(alist)]);
```

stat

Syntax

```
list stat ( string fn )
```

Description

`stat` returns information about a specified file on the policy server.

If the file `fn` exists on the policy server, `stat` returns the following list of values:

- File size in bytes
- File owner as username
- File group as groupname
- File permissions as octal
- File change date in the format: YYYY/MM/DD
- File change time in the format: HH:MM:SS
- File change time in the format: seconds since the epoch
- File access date in the format: YYYY/MM/DD
- File access time in the format: HH:MM:SS
- File access time in the format: seconds since the epoch

- File modification date in the format: YYYY/MM/DD
- File modification time in the format: HH:MM:SS
- File modification time in the format: seconds since the epoch
- File inode number

strftime

Syntax

```
string strftime (string format )
```

Description

`strftime` formats dates and times.

For more information on the standard formats for dates and times, refer to the `strftime (3)` man pages.

Table 42: Standard date and time formats

%d	Day of the month
%H	24 hour format
%I	12 hour format
%j	Day of the year
%m	Month number
%M	Minute
%S	Seconds
%w	Weekday name
Abbreviated month name	

Example

```
strftime("%m/%d/%Y") strftime("%H:%M")
```

Returns the current date and time formatted, as follows:

```
03/17/2012
 13:05
```

system

Syntax

```
string system( string command [, string input] )
```

Description

The `system` function runs the specified `command` on the policy server, taking input from and sending output to the users terminal. `system` can use an optional string parameter to pass an input string to a command instead of prompting the user for input.

`system` sets the `status` variable to the exit status of the command. Typically, the exit status of a command returns `0` if it is successful, and non-zero if it is not successful.

By default, the command runs as `root`, but you can set the `subprocuser` variable to a different user under which to run the command.

For security reasons, One Identity recommends that you set the second parameter to " " (empty quotation marks) for all system calls that do not require user input.

Example

```
#list the contents of the directory /etc – and store the result as a string in
"files".
#The exit status is stored in "status" and should be 0 if ls succeeds.
files=system("/bin/ls /etc");
if (status == 0) { …}
```

```
#perform a NIS lookup for all known hosts and store the result in "hosts"
variable.
hosts=system("ypcat hosts");
if (status==0) {…}
```

```
#send mail to "root" user - the second param contains the contents of the
mail, which
#will be passed to the mail program as standard input.
system("mail root", "mail from QPM4U\n");
```

timebetween

Syntax

```
int timebetween ( int starttime, int endtime )
```

Description

The `timebetween` function returns a `0` or `1` depending on whether the current time is between those specified. Use this function to determine whether a user is submitting a request within valid business hours. Times must be specified using the 24-hour clock. Do not use leading zeroes for time specifications, because this will be interpreted in octal. For example, 12:30 am can be 30 or 2430.

Example

```
If (timebetween(800, 1630)) {
     proc_working_hours_rules();
 } else {
     proc_outside_working_hours_rules();
 }
```

tolower

Syntax

```
string tolower ( string expr )
```

Description

`tolower` converts all upper case characters in the string to lower case. Leaves all other characters unchanged. The `tolower` function is frequently used in search and comparison expressions to make them case-insensitive.

Example

The following example accepts user inputs of "adrian", "Adrian", or "ADRIAN" and returns "adrian".

```
#this returns "adrian"
tolower("Adrian");
```

toupper

Syntax

```
string toupper( string str )
```

Description

`toupper` returns a copy of `str` with all characters converted to uppercase, if possible. Some characters such as !£$%^& or numbers do not have an uppercase equivalent.

Example

```
user = "ADRIAN"
if ( user == toupper("Adrian")) {
   accept; }
if (tolower(input("User:"))=="adrian")
   accept;
```

uname

Syntax

```
list uname ()
```

Description

The `uname` function returns a list containing the following `uname` information from the policy server:

- Operating System Name
- Network node hostname
- Operating System Release
- Operating System Version
- Machine (hardware) type

Example

```
print("Master OS is :" + uname());
```

Related Topics

osname

unsetenv

Password functions

These are the built-in password functions available to use within the pmpolicy file.

Table 43: Password functions

Name	Description
getgrouppasswd	Request a name and password of someone in the specified group on the policy server or agent.
getstringpasswd	Request a password from the user to match one generated using `pmpasswd`.
getuserpasswd	Request a user's password on the policy server or agent.

getgrouppasswd

Syntax

```
int getgrouppasswd ( string group [, int attempts])
```

Description

The getgrouppasswd function prompts you for a user name in the user group group on the policy server and then prompts for that user’s password and authenticates the user on the policy server. The user may try up to attempts times to correctly enter the password before the function exits. The default number of allowed attempts is 3.

By default, this function authenticates the user on the policy server. Set the value of getpasswordfromrun in pm.settings to yes to authenticate the user on the client instead.

Returns true if the user successfully authenticates on the policy server, otherwise returns false if the user fails to authenticate after *attempts* tries.

Example

```
if  (getgrouppasswd("admin",  2)  ==  false)
{
      reject;
}
```

Related Topics

getstringpasswd

getuserpasswd

getstringpasswd

Syntax

```
int getstringpasswd ( string password [, string prompt] [, int attempts] )
```

Description

getstringpasswd prompts you for a "code word" which has been encrypted using the pmpasswd program and specified in the configuration file. You can also specify an optional prompt, which defaults to "Password:". And, you can specify the number of attempts to allow; the default is `3`.

Returns `true` if the user enters the correct codeword; otherwise `false`.

Example

```
if  (getstringpasswd("GhDByC9JGIRFI",  "Enter  password  now:  ",  4)  ==  false)
{
      reject  ;
 }
```

Related Topics

getgrouppasswd

getuserpasswd

pmpasswd

getuserpasswd

Syntax

```
int getuserpasswd ( int user [, string prompt] [, int attempts] )
```

Description

`getuserpasswd` prompts the specified user for a password. You can specify an optional prompt, which defaults to "Password:". And you can specify the number of attempts to allow; the default is 3.

By default, this function authenticates the user on the policy server. Set the value of `getpasswordfromrun` in `pm.settings` to `yes` to authenticate the user on the client instead.

Returns `true` if the user enters the correct codeword; otherwise `false`.

Example

```
if (getuserpasswd("admin", "Password: ", 1) == false ) {
     reject;
}
```

Related Topics

getgrouppasswd

getstringpasswd

Remote access functions

These are the built-in remote access functions available to use within the pmpolicy file.

Table 44: Remote access functions

Name	Description
remotefileexists	Check a file exists on a host.
remotegroupinfo	Check if a group exists on a host.
remotegrouplist	Get a list of groups from a host.
remotesysinfo	Get the `uname` information from a host.
remoteusergroups	Get a list of a user's groups on a host.
remoteuserinfo	Get a user's information from a host.
remoteuserlist	Get a list of users on a host.

remotefileexists

Syntax

```
int remotefileexists ( string hostname, string filename )
```

Description

The `remotefileexists` function checks whether a filename exists on the remote system `hostname`.

Returns `true` if the file exists; otherwise, it returns `false`.

The remote host must be configured to run either `pmmasterd` or `pmclientd` to respond to this function.

Example

```
print(remotefileexists(runhost,"/etc/passwd"))
```

remotegroupinfo

Syntax

```
list remotegroupinfo ( string hostname, string groupname )
```

Description

`remotegroupinfo` returns the group ID and a list of members of the specified group from the remote host.

The remote host must be configured to run either `pmmasterd` or `pmclientd` to respond to this function.

Example

```
#print the bin group info from the runhost
print(remoteuserinfo(runhost,"bin"))
```

Related Topics

remoteuserinfo

remotesysinfo

remotegrouplist

Syntax

```
list remotegrouplist ( string hostname )
```

Description

remotegrouplist returns the full list of group names and the associated group IDs located on the remote host.

The remote host must be configured to run either pmmasterd or pmclientd to respond to this function.

Example

```
#print the remote groups on runhost
print(remotegrouplist(runhost))
```

Related Topics

remoteusergroups

remoteuserlist

remotesysinfo

Syntax

```
list remotesysinfo ( string hostname )
```

Description

remotesysinfo returns the full uname output from the remote system.

The remote host must be configured to run either pmmasterd or pmclientd to respond to this function.

Example

```
#print the runhost's uname info
print(remotesysinfo(runhost))
```

Related Topics

remoteuserinfo

remotegroupinfo

remoteusergroups

Syntax

```
list remoteusergroups ( string hostname, string username )
```

Description

`remoteusergroups` returns a list of groups that the specified user belongs to on the remote system.

The remote host must be configured to run either `pmmasterd` or `pmclientd` to respond to this function.

Example

```
# print root's groups on the runhost
print(remoteusergroups(runhost,"root"))
```

Related Topics

remotegrouplist

remoteuserlist

remoteuserinfo

Syntax

```
list remoteuserinfo ( string hostname, string username )
```

Description

`remoteuserinfo` returns user information for the specified user from the remote host.

The remote host must be configured to run either `pmmasterd` or `pmclientd` to respond to this function.

Example

```
#print root's info on the runhost
print( remoteuserinfo(runhost,"root") )
```

Related Topics

remotegroupinfo

remotesysinfo

remoteuserlist

Syntax

```
list remoteuserlist ( string hostname )
```

Description

`remoteuserlist` returns the full list of user names on the remote host.

The remote host must be configured to run either `pmmasterd` or `pmclientd` to respond to this function.

Example

```
#print the user list on the runhost
print(remoteuserlist(runhost))
```

Related Topics

remotegrouplist

remoteusergroups

String functions

These are the built-in string functions available to use within the pmpolicy file.

Table 45: String functions

Name	Description
match	Match a string to a pattern.
pad	Return a new string at a specified character length.
strindex	Return the position of a substring in a string.
strlen	Return the length of a string.
strsub	Return a substring of a string.
sub	Return a new string with specified replacements.
subst	Substitute part of a string.
substr	Return a substring of a string .

match

Syntax

```
int match( string regularexpr, string str )
```

Description

`match` compares a string to a regular expression.

Returns `true` if a match is found; otherwise, `false`.

Example

```
# check if user begins with j and ends with t…
 if (match("^j.*t$", user) ) {
      …
 }
```

pad

Syntax

```
int pad ( string sourcestring, string length, string padchar )
```

Description

`pad` returns a new string at the exact `length` of characters long. The beginning of the string is the `sourcestring`.

If the `length` argument is bigger than the size of the `sourcestring`, then the returned string is padded with the `padchar` argument. Otherwise, the first length characters of `sourcestring` are returned.

The `padchar` argument can also contain multiple characters, in which case the characters return padded repeatedly.

Example

```
result = pad("123",5," "); {
# returns "123"
}
result = pad("123",6,"<>"); {
# returns "123<><"
}
result = pad("User Name", 3, " "); {
# returns "User"
}
```

strindex

Syntax

```
int strindex( string str, string substr )
```

Description

`strindex` returns the numerical offset of a given string within another string. If the `substr` is not found, it returns `-1`.

Example

```
printf("%d\n",strindex("xxxfooxxx","foo"));
```

Returns: "3"

```
printf("%d\n",strindex("xxxfooxxx","bar"));
```

Returns: "-1"

Related Topics

strlen

strlen

Syntax

```
int strlen( string str )
```

Description

`strlen` returns the length of the string, `str`.

Example

```
printf("%d\n",strlen("foo"));
```

Returns: `3`

Related Topics

strindex

strsub

Syntax

```
string strsub ( string str, int start, int length )
```

Description

`strsub` returns the substring of a given length starting at a given position in the string.

Example

```
printf("%s\n",strsub("xxxfooxxx",3,3))
```

Returns "foo".

```
printf(%s\n",strsub(xxxfooxxx",3,-1))
```

-1 returns the remainder of the string, "`fooxxx`".

sub

Syntax

```
int sub ( string <regexp> string replacement string sourcestring string count )
```

Description

`sub` returns a new string from the `sourcestring` argument with the specified regular expression `regexp` replaced with the string specified in the `replacement` argument.

Example

```
result = sub("0x[[:xdigit:]]*:,"hex","These are numbers: 0xA8D, 0x34");
```

subst

Syntax

```
string subst ( string str, string pattern, string replacement )
```

Description

`subst` substitutes part of a string with another string.

Example

```
print(subst("xxxonexxx","one","two"));
```

Returns: "xxxtwoxxx"

substr

Syntax

```
string substr ( string str, int start, int length )
```

Description

`substr` returns the substring of a given length starting at a given position in the string.

Example

```
printf("%s\n",substr ("xxxfooxxx",3,3))
```

Returns "foo".

```
printf(%s\n",substr (xxxfooxxx",3,-1))
```

-1 returns the remainder of the string, "fooxxx".

User information functions

These are the built-in user information functions available to use within the pmpolicy file.

Table 46: User information functions

Name	Description
getfullname	Get a user’s full name from the policy server.
getgroup	Get a user’s primary group from the policy server.
getgroups	Get the list of groups for a user from the policy server.
gethome	Get a user’s home directory from the policy server.
getshell	Get a user’s login shell from the policy server.

getfullname

Syntax

```
string getfullname ( string user )
```

Description

`getfullname` returns the specified user’s full name from the policy server (or from the client host if `getpasswordfromrun` is set to **yes** in the policy server's `pm.settings` file). When called without arguments, the function reports the full name for the user name present inside the `runuser` variable.

Example

```
# print the fullname of root on the policy server
print(getfullname("root"));
```

Related Topics

getgroup

getgroups

gethome

getshell

getgroup

Syntax

```
string getgroup ( string user )
```

Description

`getgroup` returns the specified user’s primary group name from the policy server (or from the client host if `getpasswordfromrun` is set to **yes** in the policy server's `pm.settings` file). If no user is specified, it returns the `submituser`’s primary group.

Example

```
# print root user's primary group on the policy server
print(getgroup("root"));
```

Related Topics

getfullname

getgroups

gethome

getshell

getgroups

Syntax

```
list getgroups ( string user )
```

Description

getgroups returns the list of groups to which the specified user belongs from the policy server (or from the client host if getpasswordfromrun is set to **yes** in the policy server's pm.settings file). If you do not specify a user, it returns the submituser's secondary groups.

The following example returns the list of groups to which root belongs.

Example

```
# print the list of groups to which root belongs
print(getgroups("root"));
```

Related Topics

getgroup

gethome

getfullname

getshell

gethome

Syntax

```
string gethome( string user )
```

Description

gethome returns the specified user’s home directory from the policy server (or from the client host if getpasswordfromrun is set to **yes** in the policy server's pm.settings file).

Example

```
# set working directory to root's home dir on the policy server
runcwd = gethome("root");
```

Related Topics

getgroup

getgroups

getfullname

getshell

getshell

Syntax

```
string getshell ( string user )
```

Description

`getshell` returns the specified user’s login program from the policy server (or from the client host if `getpasswordfromrun` is set to **yes** in the policy server's `pm.settings` file).

Example

```
#check the user's shell on the policy server is in /opt/quest/bin
 shell=getshell(user);
 if (dirname(shell) != "/opt/quest/bin") {
      reject "You are only permitted to run a login shell from
/opt/quest/bin";
 }
```

Related Topics

getgroup

getgroups

gethome

getfullname

Authentication Services functions

These are the built-in Authentication Services functions available to use within the pmpolicy file.

Table 47: Authentication Services functions

Name	Description
vas_auth_user_password	Authenticate a user to Active Directory using Authentication Services.
vas_host_in_ADgrouplist	Check whether selected host name and domain is a member of any group in the selected list.
vas_host_is_member	Check whether selected host name and selected domain is a member of the selected group.
vas_user_get_groups	Check membership of the group lists.
vas_user_in_ADgrouplist	Return membership of the Active Directory group lists.
vas_user_is_member	Check whether a selected user name and selected domain is a member of the selected group.

vas_auth_user_password

Syntax

```
int vas_auth_user_password ( string user, string pmpt, [, int tries] )
```

Description

The `vas_auth_user_password` function attempts to authenticate a user to Active Directory using the Authentication Services API. This feature is platform dependent. The `feature_enabled()` function indicates whether this feature is supported on a particular policy server.

Returns 1 if the user successfully authenticates; otherwise it returns 0 (zero).

Example

```
if (feature_enabled(FEATURE_VAS) ) {
     if (!vas_auth_user_password(user, "AD Password:", 3)) {
         reject “Failed to authenticate to AD”;
     }
}
```

vas_host_in_ADgrouplist

Syntax

```
int vas_host_in_ADgrouplist ( string hostname, string domain, list ADgrouplist [,
boolean verbose] )
```

Description

The vas_host_in_ADgrouplist function checks if the selected host name and domain is a member of any group in the selected list. It calls vas_host_is_member for each item in the list.

Returns: -1 if host is not found in the list, otherwise it returns the index of the matched list entry.

vas_host_is_member

Syntax

```
int vas_host_is_member ( string hostname, string groupname [, string domain [,
boolean verbose]] )
```

Description

The vas_host_is_member function checks whether a selected host name and selected domain is a member of the selected group. If domain is empty, it defaults to the joined domain. You can specify the group name as <domain>/<group> or <group>@<domain>.

Returns:

- 0: host not in group
- 1: host in group
- -1: error

vas_user_get_groups

Syntax

```
int vas_user_get_groups ( string username, string domainname [, boolean verbose] )
```

Description

The vas_user_get_groups function checks membership of the group lists.

Returns the index of the matched list item if found, or -1 if not found.

vas_user_in_ADgrouplist

Syntax

```
int vas_user_in_ADgrouplist ( string username, string domain, list ADgrouplist [,
boolean verbose] )
```

Description

The vas_host_in_ADgrouplist function checks membership of the Active Directory group lists.

Returns the index of the matched list item if found, or -1 if not found.

vas_user_is_member

Syntax

```
int vas_user_is_member (string username, string groupname [, string domain [,
boolean verbose]] )
```

Description

The `vas_user_is_member` function checks whether a selected user name and selected domain is a member of the selected group. If domain is empty, it defaults to the joined domain. You can specify the group name as <domain>/<group> or <group>@<domain>.

Returns:

- 0: user not in group
- 1: user in group
- -1: error

Appendix E

Privilege Manager for Unix programs

This section describes each of the Privilege Manager for Unix programs and their options. The following table indicates which Privilege Manager for Unix component installs each program.

Table 48: Privilege Manager programs

Name	Description	Server	Agent	Sudo
pmbash	Is a wrapper for the GNU Bourne Again SHell that provides transparent authorization and auditing for all commands submitted during the shell session.	X	X	-
pmcheck	Verifies the syntax of a policy file.	X	-	X
pmclientd	The Privilege Manager for Unix Client daemon that listens on the configured policy server port and responds to a remote request.	X	X	-
pmclientinfo	Displays configuration information about a client host.	X	X	-
pmcp	Privilege Manager for Unix remote file copy command.	X	X	-
pmcsh	Privilege Manager for Unix C Shell provides transparent authorization and auditing for all commands submitted during the shell session.	X	X	-
pmincludecheck	Used by `pmsrvconfig` script on the primary server only. When configuring a primary server in pmpolicy type, if you do not have a	X	-	-

Name	Description	Server	Agent	Sudo
	policy file to import into the repository, then `pmincludecheck` initializes the policy from the current set of default policy files provided in the installation.			
pminfo	Registers the local host with the Privilege Manager for Unix 5.5 policy server. Note that `pminfo` is obsolete as of version 5.6 and is included for backwards compatibility only.	X	X	-
pmjoin	Configures a Privilege Manager for Unix agent to communicate with the servers in the group.	X	X	-
pmkey	Generates and installs configurable certificates.	X	X	X
pmksh	Privilege Manager for Unix K Shell provides transparent authorization and auditing for all commands submitted during the shell session.	X	X	-
pmless	A terminal pager program that allows you to view (by not modify) the contents of a text file one screen at a time.	X	X	-
pmlicense	Displays current license information and allows you to update a license (an expired one or a temporary one before it expires) or create a new one.	X	-	-
pmlist	Lists the commands that the user is permitted to run.	X	X	-
pmloadcheck	Controls load balancing and failover for connections made from the host to the configured policy servers.	X	X	-
pmlocald	The Privilege Manager for Unix Local daemon which runs programs when instructed to do so by the appropriate policy server daemon.	X	X	-
pmlog	Displays entries in a Privilege	X	-	-

Name	Description	Server	Agent	Sudo
	Manager for Unix event log.			
pmlogadm	Manages encryption options on the event log.	X	-	-
pmlogsearch	Searches all logs in a policy group based on specified criteria.	X	-	-
pmlogsrvd	The Privilege Manager for Unix log access daemon, the service responsible for committing events to the Privilege Manager for Unix event log and managing the database storage used by the event log.	X		
pmmasterd	The Privilege Manager for Unix Master daemon which examines each user request and either accepts or rejects it based upon information in the Privilege Manager configuration file. You can have multiple `pmmasterd` daemons on the network to avoid having a single point of failure.	X	-	X
pmmg	A special version of an emacs text editor to use with Privilege Manager for Unix (gnu-style key bindings).	X	X	-
pmpasswd	Generates an encrypted password which can be used in the configuration file.	X	-	-
pmpolicy	A command-line utility for managing the Privilege Manager for Unix security policy. This utility checks out the current version, checks in an updated version, and reports on the repository.	X	-	-
pmpolicyconvert	Utility that allows you to verify, and if necessary, convert any number of policy files for use with Privilege Manager for Unix V5.5 (or later).	X	-	-
pmpolsrvconfig	Configures (or unconfigures) a primary or secondary policy	X	-	-

Name	Description	Server	Agent	Sudo
	server. Allows you to grant a user access to a repository.			
pmremlog	Provides a wrapper for the `pmlog` and `pmreplay` utilities to access the event (audit) and keystroke (I/O) logs on any server in the policy group.	X	-	-
pmreplay	Replays an I/O log file allowing you to review what happened during a previous privileged session.	X	-	-
pmresolvehost	Verifies the host name or IP resolution for the local host or a selected host.	X	X	X
pmrun	Allows a user to run a command from their local machine as `root`. The policy server daemon, `pmmasterd`, examines each request from `pmrun`, and either accepts or rejects it based upon the policies specified in the policy file.	X	X	-
pmscp	Allows Privilege Manager for Unix to launch the remote scp daemons.	X	-	-
pmserviced	The Privilege Manager for Unix Service daemon listens on the configured ports for incoming connections for the Privilege Manager for Unix daemons. `pmserviced` uses options in `pm.settings` to determine the daemons to run, the ports to use, and the command line options to use for each daemon.	X	X	X
pmsh	Privilege Manager for Unix Bourne Shell that provides transparent authorization and auditing for all commands submitted during the shell session.	X	X	-
pmshellwrapper	A wrapper for any valid login shell on a host.	X	X	-
pmsrvcheck	Checks the Privilege Manager for	X	-	-

Name	Description	Server	Agent	Sudo
	Unix policy server configuration to ensure it is setup properly.			
pmsrvconfig	Configures a primary or secondary policy server.	X	-	-
pmsrvinfo	Verifies the policy server configuration.	X	-	-
pmstatus	Verifies connectivity between Privilege Manager for Unix and the `pmlocald` and `pmmasterd` daemons on the specified hosts.	X	X	-
pmsum	Generates a simple checksum of a binary.	X	-	-
pmsysid	Displays the Privilege Manager for Unix system ID.	X	X	X
pmtunneld	The Privilege Manager for Unix Tunnel daemon that acts as a proxy for `pmrun` when `pmlocald` communicates with `pmrun` through a firewall.	X	X	-
pmumacs	A special version of a microemacs text editor to use with Privilege Manager for Unix (gosling-style key bindings).	X	X	-
pmverifyprofilepolicy	Verifies the syntax and structure of the policy file and checks whether a particular command will be accepted or rejected.	X	-	-
pmvi	Allows users to access a specific file as `root` but no other `root` functions.			

pmbash

Syntax

```
pmbash -c <command>|-i|-l|-r|-s|-B|[-+]O <option>
```

Description

The Privilege Manager for Unix Bourne Again SHell (`pmbash`) command is a wrapper program for the GNU Bourne Again SHell (bash), that provides transparent authorization and auditing for all commands submitted during the shell session. `pmbash` supports the standard options for `bash`.

Using the appropriate policy file variables, you can configure each command entered during a shell session, to be:

- forbidden by the shell without further authorization to the policy server
- allowed by the shell without further authorization to the policy server
- presented to the policy server for authorization

Once allowed by the shell, or authorized by the policy server, all commands run locally as the user running the shell program.

Unlike the other Privilege Manager for Unix shells, `pmbash` is not a standalone shell. It is a wrapper that runs the system version of the bash shell while logging keystrokes and authorizing shell commands via Privilege Manager for Unix. Command authorization is limited to external commands: `pmbash`, cannot authorize shell built-in commands.

Options

`pmbash` has the following options.

Table 49: Options: pmsh

Option	Description
-B	Allows the shell to run in the background.
-c <command>	Runs the specified command from the next argument.
-i	Runs the shell in interactive mode even when input is not from a terminal.
-l	Acts as a login shell, the shell will read the contents of /etc/profile and $HOME/.profile if they exist.
[+-]O <shopt_option>	Sets or clears one of the shell options accepted by the shopt built-in command.
-r	Runs the shell in restricted mode.
The shell reads commands from standard input even when there are additional non-option arguments.	

Additional long options may also be specified, see the bash manual for details.

pmcheck

Syntax

```
pmcheck [ -z on|off[:<pid>] ] | [ -v ] |
           [ [ -a <string> ] [ -b ] [ -c ] [ -e <requestuser> ]
           [ -f <filename> ] [ -g <group> ] [ -h <hostname> ] [ -i ]
           [ -l <shellprogram> ] [ -m <YY[YY]/MM/DD> ] [ -n <HH[:MM]> ]
           [ -o sudo|pmpolicy ] [ -p <policydir> ] [ -q  ]  [  -r <remotehost> ]
           [ -s <submithost> ] [ -t ] [ -u <runuser> ] [ command [ args ]]]
```

Description

Use the `pmcheck` command to test the policy file. Although the policy server daemon `pmmasterd` reports configuration file errors to a log file, always use `pmcheck` to verify the syntax of a policy file before you install it on a live system. You can also use the `pmcheck` command to simulate running a command to test whether a request will be accepted or rejected.

The `pmcheck` program exits with a value corresponding to the number of syntax errors found.

Options

`pmcheck` has the following options.

Table 50: Options: pmcheck

Option	Description
-a <string>	Checks if the specified string, entered during the session, matches any `alertkeysequence` configured. You can only specify this option if you supply a command. This option is only relevant when using the `pmpolicy` type.
-b	Run in batch mode. By default, `pmcheck` runs in interactive mode, and attempts to emulate the behavior of the `pmmasterd` when parsing the policy file. The `-b` option ensures that no user interaction is required if the policy file contains a password or input function; instead, a successful return code is assumed for any password authentication functions.
-c	Runs in batch mode and displays output in csv format. By default `pmcheck` runs in interactive mode. The `-c` option ensures that no user interaction is required if the policy file contains a password prompt or input function and no commands that require remote connections are attempted.
-e	Sets the value of `requestuser`. This option allows you to specify the

Option	Description
<requestuser>	group name to use when testing the configuration. This emulates running a session using the `pmrun -u <user>` option to request that Privilege Manager for Unix runs the command as a particular `runuser`.
-f <filename>	Sets path to policy filename. Provides an alternative configuration filename to check. If not fully qualified, this path is interpreted as relative to the `policydir`, rather than to the current directory.
-g <group>	Sets the group name to use. If not specified, then `pmcheck` looks up the user on the master policy server host to get the group information. This option is useful for checking a user and group that does not exist on the policy server.
-h <hostname>	Specifies execution host used for testing purposes.
-i	Ignores check for `root` ownership of policy.
-l <shellprogram>	Verifies the command as though it was run from within a Privilege Manager for Unix shell program. This special case of `pmcheck` verifies the specified shell program first, and if accepted, it verifies the specified command as a normal executable program within this shell to determine whether it would be forbidden, accepted, or rejected. This option is only relevant when using the `pmpolicy` type.
-m <YY [YY]/MM/DD>	Checks the policy for a particular date. Enter Date in this format: YY [YY]/MM/DD. Defaults to the current date.
-n <HH[:MM]>	Checks the policy for a particular time. Enter Time in this format: HH [:mm]. Defaults to the current time.
-o <policytype>	Interprets the policy with the specified policy type: • sudo • pmpolicy
-p policydir	Forces `pmcheck` to use a different directory to search for policy files included with a relative pathname. The default location to search for policy files is the `policydir` setting in `pm.settings`.
-q	Runs in *quiet* mode, `pmcheck` does not prompt the user for input, print any errors or prompts, or run any system commands. The exit status of `pmcheck` indicates the number of syntax errors found (0 = success). This is useful when running scripted applications that require a simple syntax check.
-r remotehost	Sets the value of the `clienthost` variable within the configuration file, useful for testing purposes. If you log in by means of `pmksh` or `pmshellwrapper`, the `clienthost` variable is set to the name of the remote host you used to log in. Otherwise the `clienthost` variable is set to the value of the `submithost`

Option	Description
	variable.
-s submithost	Sets the value of the `submithost` variable within the configuration file, useful for testing purposes.
-t	Runs in *quiet* mode to check whether a command would be accepted or rejected. By default, `pmcheck` runs in *interactive* mode. The `–t` option ensures that no user interaction is required if the policy file contains a password prompt or input function, no output is displayed and no commands that require remote connections are attempted. **Exit Status**: • 0: Command accepted • 11: Password prompt encountered. The command will only be accepted if authentication is successful • 12: Command rejected • 13: Syntax error encountered
-u <runuser>	Sets the value of the `runuser` variable within the configuration file, useful for testing purposes.
-v	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables debug tracing, and optionally sends SIGHUP to running process. Refer to Enabling program-level tracing on page 179 before using this option.
Sets the command name and optional arguments.	

You can use `pmcheck` two ways: to check the syntax of the configuration file, or to test whether a request is accepted or rejected (that is, to simulate running a command).

By default, `pmcheck` runs the configuration file interactively in the same way as `pmmasterd` and reports any syntax errors found. If you supply an argument to a command, it reports whether the requested command is accepted or rejected. You can use the `–c` and `–q` options to verify the syntax in batch or silent mode, without any user interaction required.

When you run a configuration file using `pmcheck`, you are allowed to modify the values of the incoming variables. This is useful for testing the configuration file's response to various conditions. When `pmmasterd` runs a configuration file, the incoming variables are read-only.

Example

To verify whether the pmpolicy file `/opt/quest/qpm4u/policies/test.conf` allows user **jsmith** in the **users** group to run the `passwd root` command on host, `host1`, enter:

```
pmcheck -f /opt/quest/qpm4u/policies/test.conf –o pmpolicy –u jsmith –g users
-h host1 passwd root
```

Related Topics

pmkey

pmlocald

pmmasterd

pmpasswd

pmreplay

pmrun

pmsum

pmclientd

Syntax

```
pmclientd [-v]i|[-z on|off[:<pid>]]
```

Description

The `pmclientd` daemon runs on an agent and allows the agent to respond to remote requests sent by a policy server as a result of calling a remote function from the policy file. It is not required on a policy server, as the `pmmasterd` daemon can serve these requests, if received from another policy server. `pmclientd` listens on the configured policy server port and responds to a remote request received from any valid policy server or any host listed in the `clients` setting in `pm.settings`.

Options

`pmclientd` has the following options.

Table 51: Options: pmclientd

Option	Description
-v	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables debug tracing, and optionally sends SIGHUP to running process. Refer to Enabling program-level tracing on page 179 before using this option.

pmclientinfo

Syntax

```
pmclientinfo -v | [-z on|off[:<pid>]]] | -c [-h <host>]
```

Description

The `pmclientinfo` utility displays configuration information about a client host. This utility provides some information about the policy server group and the license features supported by the policy server group. You can specify a host on the command line to retrieve the details from a specific policy server host. Otherwise, the utility checks each policy server listed in the `pm.settings` file in turn until it finds one in a policy server group. Any user can run `pmclientinfo`.

Options

`pmclientinfo` has the following options.

Table 52: Options: pmclientinfo

Option	Description
-c	Displays CSV, rather than human-readable output.
-h <host>	Specifies policy server host name to interrogate for policy group information.
-v	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

Examples

Any user on the host can run this utility. It displays the following information, in human readable or CSV format:

```
- Joined to a policy group                                      : YES
- Policy group name configured for this policy server group     : adminGroup1
- Primary policy server hostname                                : adminhost1
```

Human Readable output from a client:

```
- Joined to a policy group                                      : YES
- Name of policy group                                          : adminGroup1
- Hostname of primary policy server : adminhost1.example.com
```

CSV output from a client:

```
PMCLIENTINFO.JOINED,Joined to a policy group,YES
PMCLIENTINFO.POLICYGROUPNAME,Name of policy group,adminGroup1
PMCLIENTINFO.PRIMARYPOLICYSERVER,Hostname of primary policy
server,adminhost1.example.com
```

Files

- Settings file: `/etc/opt/quest/qpm4u/pm.settings`

Related Topics

pmjoin

pmcp

Syntax

```
pmcp [-v]|[-z on|off[:<pid>]] [-m <masterhost>] file1 rhost:file2
```

Description

Use `pmcp` to copy a file from one host to another. The `pmcp` command allows you to select the policy server host to contact, bypassing the usual selection methods. The specified host must be present in the `masters` setting in the `pm.settings` file. This functionality is the same as using `pmrun [-m masterhost]`.

You can use the following policy variables with `pmcp`:

Table 53: Policy variables: pmcp

Variable	Description
filesize	Specifies the size of the source file.
filename	Specifies the name of the source file, including the full path.
filedest	Specifies the name of the target file, including the full path.
fileuser	Specifies the user name associated with the source file UID.
filegroup	Specifies the group name associated with the source file GID.
Specifies the date that the source file was last modified. This returns a string in the form: **YYYY/MM/DD**.	

Options

`pmcp` has the following options.

Table 54: Options: pmcp

Option	Description
-m <masterhost>	Selects a policy server host to contact.
-v	Displays product version information.
-z	Enables or disables debug tracing, and optionally sends SIGHUP to running process. Refer to Enabling program-level tracing on page 179 before using this option.

pmcsh

Syntax

```
pmcsh
```

Description

The Privilege Manager for Unix C Shell (`pmcsh`) command starts a C shell, an interactive command interpreter and a command programming language that uses syntax similar to the C programming language. The C shell carries out commands either interactively from a terminal keyboard or from a file. `pmcsh` is a fully featured version of `csh`, that provides transparent authorization and auditing for all commands submitted during the shell session. All standard options for `csh` are supported by `pmcsh`.

To see details of the options and the shell built-in commands supported by `pmcsh`, run `pmcsh -?`

Using the appropriate policy file variables, you can configure each command entered during a shell session, to be:

- forbidden by the shell without further authorization to the policy server
- allowed by the shell without further authorization to the policy server
- presented to the policy server for authorization

Once allowed by the shell, or authorized by the policy server, all commands run locally as the user running the shell program.

Options

`pmcsh` has the following options.

Table 55: Options: pmcsh

Option	Description
-b <file>	Runs in batch mode. Reads and runs commands from specified file.
-B	Allows the shell to run in the background.
-c <command>	Runs specified command from next argument.
-d	Loads directory stack from `~/.cshdirs`.
-Dname [=value]	Defines environment variable name as specified value (DomainOS only).
-e	Exits on any error.
-f	Starts faster by ignoring the start-up file.
-F	Uses fork() instead of vfork() when spawning (ConvexOS only).
-i	Runs in interactive mode, even when input is not from a terminal.
-l	Acts as a login shell, must be the only option specified.
-m	Loads the start-up file, whether or not owned by effective user.
-n <file>	Runs in no execute mode, just checks syntax of the specified file.

Option	Description
-q	Accepts SIGQUIT for running under a debugger.
-s	Reads commands from standard input.
-t	Reads one line from standard input.
-v	Echos commands after history substitution.
-V	Like -v but including commands read from the start up file.
-x	Echos commands immediately before execution.
-X	Like **-x** but including commands read from the start up file.
--help | ?	Prints this message and exits.
--version	Prints the version shell variable and exits.

pmcsh supports the following built-in functions:

`:, @, alias, alloc, bg, bindkey, break, breaksw, builtins, case, cd, chdir, complete, continue, default, dirs, echo, echotc, else, end, endif, endsw, eval, exec, exit, fg, filetest, foreach, glob, goto, hashstat, history, hup, if, jobs, kill, limit, log, login, logout, ls-F, nice, nohup, notify, onintr, popd, printenv, pushd, rehash, repeat, sched, set, setenv, settc, setty, shift, source, stop, suspend, switch, telltc, termname, time, umask, unalias, uncomplete, unhash, unlimit, unset, unsetenv, wait, where, which, while`

pmincludecheck

Syntax

```
pmincludecheck [-v][-p <path>][-f][-o]
```

Description

pmincludecheck is used by the pmsrvconfig script on the primary server only. When configuring a primary server in pmpolicy mode, if you do not have a policy file to import into the repository, then pmincludecheck initializes the policy from the current set of default policy files provided in the installation.

Options

pmincludecheck has the following options.

Table 56: Options: pmincludecheck

Option	Description
-v	Displays the version number of Privilege Manager for Unix and exits.
-p <path>	Sets `policyDir` to the specified path.
-f	Sets `policyDir` to the specified file.
-o	Forces rewrite of the current policy file, which archives and replaces the current policy file.

pminfo

Note that `pminfo` is obsolete in version 5.6 or higher and is included for backwards compatibility only.

Syntax

```
pminfo -v | [ -s | -d | -r [ -m <master> ] ]
```

Description

The `pminfo` program allows the local host to register with Privilege Manager for Unix. If your Privilege Manager for Unix policy server has a host license, this registration is mandatory; agents cannot communicate successfully with the policy server until registration is completed and the policy server has allocated a license slot for the agent.

During registration, information about the local host configuration is sent to the Privilege Manager for Unix policy server. This includes a list of the agent's IP addresses.

To view the information that will be sent to the Privilege Manager for Unix policy server, run `pminfo` with the `-s` option.

The `pminfo` program located on an agent identifies itself to the policy server using the agent's fully qualified host name and a unique registration data string.

If the host name or IP addresses of the agent are changed, then the agent must re-register with the policy server.

Options

`pminfo` has the following options.

Table 57: Options: pminfo

Option	Description
-d	Unregisters the local host from Privilege Manager for Unix.
-m <master>	Specifies a single policy server host to register with. By default, `pminfo` attempts to register with all policy servers configured in `etc/opt/quest/pm.settings`.
-r	Registers the local host with Privilege Manager for Unix.
-s	Dumps the local host registration information to stdout.
-v	Displays the version number of Privilege Manager for Unix and exits.

Files

- Privilege Manager for Unix configuration file: `/etc/opt/quest/qpm4u/policy/pm.conf`
- Privilege Manager for Unix communication parameters: `/etc/opt/quest/qpm4u/pm.settings`

Related Topics

pmlicense

pmmasterd

pmjoin

Syntax

```
pmjoin -h | --help [-abitv] [-d <variable>=<value>] [<policy_server_host>]
         [-bv] -u --unjoin
         [--accept] [--batch] [--define <variable>=<value>] [--interactive]
         [--selinux] [--tunnel] [--verbose] <policy_server_host>
```

Description

Use the `pmjoin` command to join a PM Agent to the specified policy server. When you join a policy server to a policy group, it enables that host to validate security privileges against a single common policy file located on the primary policy server, instead of on the host. You must run this configuration script after installing the PM Agent package to allow this agent to communicate with the servers in the group.

Options

`pmjoin` has the following options.

Table 58: Options: pmjoin

Option	Description
-a | --accept	Accepts the End User License Agreement (EULA), `/opt/quest/qpm4u/pqm4u_eula.txt`.
-b | --batch	Runs in batch mode, will not use colors or require user input under any circumstances.
-d <variable>->=<value> | --define <variable>->=<value>	Specifies a variable for the `pm.settings` file and its associated value.
-h | --help	Prints this help message.
-i | --interactive	Runs in interactive mode, prompting for configuration parameters instead of using the default values.
-S | --selinux	Enable support for SELinux in Privilege Manager for Unix. A SELinux policy module will be installed, which allows the pmlocal daemon to set the security context to that of the run user when executing commands. This requires that the policycoreutils package and either the selinux-policy-devel (RHEL7 and above) or selinux-policy (RHEL6 and below) packages be installed.
-t | --tunnel	Configures host to allow Privilege Manager for Unix connections through a firewall.
-u | --unjoin	Unconfigures a Privilege Manager for Unix agent.
-v | --verbose	Displays verbose output while configuring the host.

Examples

See Joining PM Agent to a Privilege Manager for Unix policy server for usage examples.

Files

- Directory when pmjoin logs are stored: `/opt/quest/qpm4u/install`

Related Topics

pmrun

pmlocald

pmmasterd

pmpolicy

pmsrvconfig

pmkey

Syntax

```
pmkey -v | [-z on|off[:<pid>]]
         -a <keyfile>
         [ [-l | -r | -i <keyfile>]
         [-p <passphrase>] [-f]]
```

Description

Use the pmkey command to generate and install configurable certificates.

In order for a policy evaluation request to run, keys must be installed on all hosts involved in the request. The keyfile must be owned by *root* and have permissions set so only *root* can read or write the keyfile.

Options

pmkey has the following options.

Table 59: Options: pmkey

Option	Description
-a <keyfile>	Creates an authentication certificate.
-i <keyfile>	Installs an authentication certificate.
-l	Creates and installs a local authentication certificate to this file: `/etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_localhost` This is equivalent to running the following two commands: • `pmkey -a /etc/opt/quest/qpm4u/.qpm4u/.keyfiles/ key_localhost` -OR- • `pmkey -i /etc/opt/quest/qpm4u/.qpm4u/.keyfiles/ key_localhost`
-f	Forces the operation. For example: • Ignore the password check when installing `keyfile` using `-i` or `-r` • Overwrite existing `keyfile` when installing local `keyfile` using `–l`
-p <passphrase>	Passes the `passphrase` on the command line for the `-a` or `-l` option. If not specified, `pmkey` prompts the user for a passphrase.
-r	Installs all remote keys that have been copied to this directory: `/etc/opt/quest/qpm4u/.qpm4u/.keyfiles/key_<hostname>`

Option	Description
	This provides a quick way to install multiple remote keys.
-v	Displays the Privilege Manager for Unix version and exits.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

Examples

The following command generates a new certificate, and puts it into the specified file:

```
pmkey -a <filename>
```

The following command installs the newly generated certificate from the specified file:

```
pmkey -i <filename>
```

Related Topics

pmcheck

pmlocald

pmmasterd

pmpasswd

pmreplay

pmrun

pmsum

pmksh

Syntax

```
pmksh
```

Description

The Privilege Manager for Unix K Shell (`pmksh`) starts a Korn shell, an interactive command interpreter and a command programming language. The Korn shell carries out commands either interactively from a terminal keyboard or from a file. `pmksh` is a fully featured version of `ksh`, that provides transparent authorization and auditing for all commands submitted during the shell session. All standard options for `ksh` are supported by `pmksh`.

To see details of the options and the shell built-in commands supported by `pmksh`, run `pmksh -?`.

Note that `pmksh` supports the `-B` option which allows the entire shell to run in the background when used in conjunction with '&. For example, `pmksh –B –c backgroundshellscript.sh &` will run the specified shell script in the background using `pmksh`.

Using the appropriate policy file variables, you can configure each command entered during a shell session, to be:

- forbidden by the shell without further authorization to the policy server
- allowed by the shell without further authorization to the policy server
- presented to the policy server for authorization

Once allowed by the shell, or authorized by the policy server, all commands run locally as the user running the shell program.

pmless

Syntax

```
pmless /<full_path_name>
```

Description

The `pmless` pager is similar to the `less` pager. It has been modified so that you can use it securely with the Privilege Manager for Unix programs. Because of this, you must specify a full pathname as a command line argument to `pmless`. Also, you will not be able to access any files other than the ones you specify at startup time. Nor will you be allowed to spawn any processes.

Using this program in conjunction with Privilege Manager for Unix allows you to access a specific file as `root` but not other `root` functions.

pmlicense

Syntax

```
pmlicense -h
            [-c]
            -v [-c]
            -v <xmlfile> [-c]
            -l|-x <xmlfile> [-c] [-f] [-e]
            -u [s|f][-c][-d m|y][-o <outfile>][-s d|h][-t u|p|k]
            -r [e]
            -z on |off[:<pid>]
```

Description

The `pmlicense` command allows you to display current license information, update a license (an expired one or a temporary one before it expires) or create a new one. If you do not supply an option, then `pmlicense` displays a summary of the combined licenses configured on this host.

Options

`pmlicense` has the following options.

Table 60: Options: pmlicense

Option	Description
-c	Displays output in CSV, rather than human-readable format.
–d	Filters a license report; restricting the date to either: • **m**: Only report licenses used in the past month. • **y**: Only report licenses used in the past year.
-e	Does not forward the license change to the other servers in the group.
-f	Does not prompt for confirmation in interactive mode.
-h	Displays usage.
-l <xmlfile>	Configures the selected XML license file, and forwards it to the other servers in the policy group. This option must be run as the root user or a member of the pmpolicy group.
-o <outfile>	Sends report output to selected file rather than to the default. For csv output, the default is file: `/tmp/pmlicense_report_<uid>.txt`; for human-

Option	Description
	readable output, the default is stdout.
-r	Regenerates and configures the default trial license, removing any configured commercial licenses, and forwards this change to the other servers in the policy group.
-s	Sort the report data by either: • **d**: date (newest first) • **h**: hostname (lowest first)
-t	Filters license report by client type: • **u**: Privilege Manager for Unix client • **p**: sudo policy plugin • **k**: sudo keystroke plugin
-u	Displays the current license utilization on the master policy server: • **s**: Show summary of hosts licensed • **f**: Show full details of hosts licensed, with last used times
-v	If you do not provide a file argument, it displays the details of the currently configured license. If you provide a file argument, it verifies the selected XML license file and displays the license details.
-x <xmlfile>	Configures the selected XML license file. This option is deprecated, use the "-l" option instead.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

License data is updated periodically by the `pmloadcheck` daemon. See pmloadcheck on page 417 for details.

Examples

To display current license status information, enter the following:

```
# pmlicense
```

Privilege Manager for Unix displays the current license information, noting the status of the license. The output will be similar to the following:

```
*** One Identity Privilege Manager for Unix ***
*** Privilege Manager for Unix VERSION 6.n.n (nnn) ***
*** CHECKING LICENSE ON HOSTNAME:<host>, IP address: <IP>
*** SUMMARY OF ALL LICENSES CURRENTLY INSTALLED ***
*License Type                               PERMANENT
*Commercial/Freeware License                COMMERCIAL
*Expiration Date                            NEVER
*Max QPM4U Client Licenses                  1000
*Max Sudo Policy Plugin Licenses            0
*Max Sudo Keystroke Plugin Licenses         0
*Authorization Policy Type permitted        ALL
*Total QPM4U Client Licenses In Use         2
*Total Sudo Policy Plugins Licenses in Use  0
*Total Sudo Keystroke Plugins Licenses in Use  0

*** LICENSE DETAILS FOR PRODUCT:QPM4U
*License Version                            1.0
*Licensed to company                        Testing
*Licensed Product                           QPM4U(1)
*License Type                               PERMANENT
*Commercial/Freeware License                COMMERCIAL
*License Status                             VALID
*License Key                                PSXG-GPRH-PIGF-QDYV
*License tied to IP Address                 NO
*License Creation Date                      Tue Feb 08 2012
*Expiration Date                            NEVER
*Number of Hosts                            1000
```

To update or create a new a license, enter the following at the command line:

```
pmlicense -l <xmldoc>
```

Privilege Manager for Unix displays the current license information, noting the status of the license, and then validates the information in the selected `.xml` file, for example:

```
*** One Identity Privilege Manager for Unix ***
*** Privilege Manager for Unix VERSION 7.n.n (nnn) ***
*** CHECKING LICENSE ON HOSTNAME:<host>, IP address:<IP> ***
*** SUMMARY OF ALL LICENSES CURRENTLY INSTALLED ***
*License Type                               PERMANENT
*Commercial/Freeware License                COMMERCIAL
*Expiration Date                            NEVER
*Max QPM4U Client Licenses                  1000
*Max Sudo Policy Plugin Licenses            0
```

```
*Max Sudo Keystroke Plugin Licenses           0
*Authorization Policy Type permitted          ALL
*Total QPM4U Client Licenses In Use           2
*Total Sudo Policy Plugins Licenses in Use    0
*Total Sudo Keystroke Plugins Licenses in Use  0
*** Validating license file: <xmldoc> ***
*** LICENSE DETAILS FOR PRODUCT:QPM4U
*License Version                              1.0
*Licensed to company                          Testing
*Licensed Product                             QPM4U(1)
*License Type                                 PERMANENT
*Commercial/Freeware License                  COMMERCIAL
*License Status                               VALID
*License Key                                  PNFT-FDIO-YSLX-JBBH
*License tied to IP Address                   NO
*License Creation Date                        Tue Feb 08 2012
*Expiration Date                              NEVER
*Number of Hosts                              100
*** The selected license file (<xmldoc>) contains a valid license ***

Would you like to install the new license? y
Type y to update the current license.
Archiving current license… [OK]
*** Successfully installed new license ***
```

Related Topics

pmmasterd

Installing licenses

Displaying license usage

pmlist

Syntax

```
pmlist
```

Description

The `pmlist` command displays a list of commands the current user is permitted to run. It is only valid when using the profile-based policy.

If the server is configured to use the default profile policy, use the `pmlist` command to list the commands that you are permitted to run. The server evaluates all configured profiles in the policy; for those that match the submit user and host, it prints out the commands that are permitted by the profile.

pmloadcheck

Syntax

```
pmloadcheck -v
                -z on | off[:<pid>]
                -s|-p|-i [-e <interval>][-t <sec>]
                [-c|-f][-b][ -h <master>][-t <sec>] [-a][-r]
```

Description

The `pmloadcheck` daemon runs on each host. The pmloadcheck daemon runs on Privilege Manager for Unix policy servers. By default, every 60 minutes the daemon verifies the status of the configured policy servers. It controls load balancing and failover for connections made from the host to the configured policy servers, and on secondary servers, it sends license data to the primary server.

When the `pmloadcheck` daemon runs, it attempts to establish a connection with the policy servers to determine their current status. If `pmloadcheck` successfully establishes a session with a policy server, it is marked as *online* and is made available for normal client sessions. If `pmloadcheck` does not successfully establish a session with a policy server, it is marked as *offline*.

Information is gathered from a policy server each time a normal client session connects to the policy server. This information is used to determine which policy server to use the next time a session is requested. If an agent cannot establish a connection to a policy server because, for example, the policy server is offline, then this policy server is marked as *offline* and no more connections are submitted to this policy server until it is marked available again.

To check the current status of all configured policy servers, and display a brief summary of their status, run `pmloadcheck` with no options. Add the `–f` option to show full details of each policy server status.

Options

`pmloadcheck` has the following options.

Table 61: Options: pmloadcheck

Option	Description
-a	Verifies the connection as if certificates were configured.
-b	Runs in batch mode.
-c	Displays output in CSV format.
-e <interval>	Sets the refresh interval (in minutes) to determine how often the `pmloadcheck` daemon checks the policy server status. Default = 60.
-f	Shows full details of the policy server status when verifying and displaying policy server status.
-h <master>	Selects a policy server to verify.
-i	Starts up the `pmloadcheck` daemon, or prompt an immediate recheck of the policy server status if it is already running.
-P	Sends SIGNUP to a running daemon.
-p	Pauses (sends SIGUSR1) to a running daemon.
-r	Reports last cached data for selected servers instead of connecting to each one.
-s	Stops the `pmloadcheck` daemon if it is running.
-t <sec>	Specifies a timeout (in seconds) to use for each connection.
-v	Displays the version string and exits.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

pmlocald

Syntax

```
pmlocald - v | [-s] [-e <filename>] [-m <polserverspec>] | -z on|off [:<pid>]
```

Description

The Privilege Manager for Unix local daemon (`pmlocald`) runs programs when instructed to do so by the appropriate policy server daemon. `pmlocald` is started from `pmserviced`.

Unless the `-m` option is used, it first checks the `/etc/opt/quest/qpm4u/pm.settings` file to determine the policy server daemons from which it is allowed to accept requests. If the request is legitimate, it then runs and manages the program.

Options

`pmlocald` has the following options.

Table 62: Options: pmlocald

Option	Description
-e <filename>	Sends any errors to the specified file; applies only to local daemon errors.
-m <polserverspec>	Specifies the policy server daemon from which requests are accepted. `polserverspec` is either a host name, or a netgroup name preceded by a + or a - (+ includes the netgroup, - excludes it). You can specify `polserverspec` more than once. If you use the `-m` option, it does not consult `masterhost` setting in the `/etc/opt/quest/qpm4u/pm.settings` file.
-s	Sends any errors generated to `syslog`.
-v	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables tracing for this program and optionally for a currently running process. Refer to Enabling program-level tracing on page 179 before using this option.

Files

File containing Privilege Manager for Unix communication parameters, including the list of valid master hosts:

```
/etc/opt/quest/qpm4u/pm.settings
```

Related Topics

pmcheck

pmkey

pmmasterd

pmpasswd

pmreplay

pmrun

pmsum

pmlog

Syntax

```
pmlog [-dlvq] [-p|a|e|r|x <printexpr>] [-f <filename>] [[-c] <constraint>]
         [[-c] <constraint>] [-f <filename>] -h [-z on|off[:<pid>]]
         [--user <username>]
         [--runuser <username>] [--runhost <hostname>] [--reqhost <hostname>]
         [--masterhost <hostname>][--command <pattern>] [--reqcommand <pattern>]
         [--runcommand <pattern>][--before "<YYYY/MM/DD hh:mm:ss>"]
         [--after "<YYYY/MM/DD hh:mm:ss>"][--result Accept|Reject]
```

Description

Use the `pmlog` command to selectively choose and display entries in a Privilege Manager for Unix event log. Each time a job is accepted, rejected, or completed by `pmmasterd`, an entry is appended to the file specified by the `eventlog` variable in the configuration file. `eventlog` is sent to `/var/opt/quest/qpm4u/pmevents.db` on all platforms.

Options

`pmlog` has the following options.

Table 63: Options: pmlog

Option	Description
-a <expression>	Sets the print expression for accept events to the specified expression.
-c <constraint>	Selects particular entries to print; specify **constraint** as a Boolean expression. See Examples.
-d	Dumps each entry as it is read without matching 'accept' and 'end' entries. The `-d` (dump) option forces `pmlog` to print each entry as it is read from the file. The default output format includes a unique identifier at the start of each record, allowing 'end' events to be matched with 'accept' events.
-e <expression>	Sets the print expression for finish events to the specified expression.
-f <filename>	Reads the event log information from the specified file.
-h	Displays usage information.
-l	Dumps alert log entries only.

Option	Description
-p <expression>	Sets the print expression for all event types to the specified expression.
-q	Runs in quiet mode; no expression errors (for example, undefined variables) are printed.
-r <expression>	Sets the print expression for reject events to the specified expression.
-v	Turns on verbose mode.
-x <expression>	Sets the print expression for alert events to the specified expression.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.
Quick Search Options	
--user <username>	Selects entries in which the requesting user matches `username`.
--runuser <username>	Selects entries in which `runuser` matches `username`.
--runhost <hostname>	Selects entries in which `runhost` matches `hostname`.
--reqhost <hostname>	Selects entries in which the requesting host matches `hostname`.
--masterhost <hostname>	Selects entries in which `masterhost` matches `hostname`.
--command <pattern>	Selects entries in which the requested command matches `pattern`.
--reqcommand <pattern>	Return events where the given text appears anywhere in the requested command line.
--runcommand <pattern>	Selects entries in which the `runcommand` host matches `pattern`.
--before "<YYYY/MM/DD hh:mm:ss>"	Selects entries occurring before the specified date and time.
--after "<YYYY/MM/DD hh:mm:ss>"	Selects entries occurring after the specified date and time.
--result Accept|Reject	Selects entries that were accepted or rejected.

Examples

Without arguments, `pmlog` reads the default `eventlog` file and prints all its entries. If you have chosen a different location for the event log, use the `-f` option to specify the file for `pmlog`.

By default, `pmlog` displays one entry for each completed session (either rejected or accepted). You can filter the results to print only entries which satisfy the specified constraint using the `-c` option. In these examples the -c option is used to specify a constraint as a Boolean expression:

```
pmlog -c'event=="Reject"'
```

```
pmlog -c'date > "2008/02/11"'
```

```
pmlog -c'user=="dan"'
```

which prints only rejected entries, entries that occur after February 11, 2008, or requests by user Dan, respectively.

See Privilege Manager for Unix Variables on page 190 for more information about policy variables.

The following options accept shortcut notations to specify constraints:

- --user username
- --runuser username
- --reqhost hostname
- --runhost hostname
- --masterhost hostname
- --command command
- --runcommand command
- --reqcommand command
- --before "YYYY/MM/DD hh:mm:ss"
- --after "YYYY/MM/DD hh:mm:ss"
- --result Accept|Reject

For example, here are equivalent constraints to the previous example specified using shortcuts:

```
pmlog --result Reject
pmlog --after "2008/02/11 00:00:00"
```

```
pmlog --user dan
```

With shortcuts, you can express user names and host names as patterns containing wild card characters (? and *). For example, to display entries for all requests for user1, user2, and user3, use the following shortcut:

```
pmlog --user “user?"
```

Enclose patterns containing wild card characters in quotes to avoid being interpreted by the command shell.

Use the `-d` and `-v` options for debugging. Normally, when `pmlog` finds an 'accept' entry, it refrains from printing until the matching 'end' entry is found; all requested information including `exitstatus`, `exitdate`, and `exittime` is then available to print.

The `-d` (dump) option forces `pmlog` to print each entry as it is read from the file. The default output format includes a unique identifier at the start of each record, allowing 'end' events to be matched with 'accept' events.

The `-v` (verbose) option prints all the variables stored with each entry.

The `-t` option turns on *tail follow* mode. The program enters an endless loop, sleeping and printing new event records as they are appended to the end of the log file. The `-d` flag is implied when using `-t`.

You can specify the output format for each of the three event types - 'accept', 'reject' or 'finish' - with the `-a`, `-r`, and `-e` options. Use the `-p` option to set the output for all three event types.

For example, to print only the dates and names of people making requests, enter:

```
pmlog -p'date + "\t" + user + "\t" + event'
```

-OR-

```
pmlog -p 'sprintf("%s %-8s %s", date, user, event)'
```

See Listing event logs on page 161 for more examples of using the `pmlog` command.

Note that if you run `pmlog --csv console` to obtain CSV output from pmlog, refer to pmlogsearch on page 427 for a list of the column headings.

pmlogadm

Syntax

```
pmlogadmin> archive <event_log_path> <archive_path> --before <YYYY-MM-DD>
           [--clean-source] [--dest-dir <destination_path>] [--no-zip]
pmlogadmin> archive <event_log_path> <archive_path> --older-than <days>
           [--clean-source] [--dest-dir <destination_path>] [--no-zip]
pmlogadmin> backup <event_log_path> <backup_path>
pmlogadmin> create <new_event_log_path>
pmlogadmin> encrypt enable|disable|rekey <event_log_path>
pmlogadmin> help [<command>]
pmlogadmin> import [-y|-n] <source_event_log> <dest_event_log>
pmlogadmin> info <event_log_path>
pmlogadmin> --help|-h
pmlogadmin> --version|-v
pmlogadmin> -z on|off[:<pid>]
```

Description

Privilege Manager event log administration utility. Use `pmlogadm` to manage encryption options on the event log.

Options

`pmlogadm` has the following options.

Table 64: Options: pmlogadm

Option	Description
-h, --help	Displays usage information. `help [<command>]` By default the `help` command displays the general usage output. When you specify a command, it displays a usage summary for that command.
-v, --version	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables debug tracing, and optionally sends SIGHUP to running process. Refer to Enabling program-level tracing on page 179 before using this option.

Table 65: Global options: pmlogadm

Option	Description
--verbose	Enables verbose output.
--silent	Disables all output to stdout. Errors are output to stderr.

Table 66: Valid commands: pmlogadm

Option	Description

archive	Moves old events to an archive.
	`archive <event_log_path> <archive_name> --before <YYYY-MM-DD> [--cleansource] [--dest-dir <destination_path>] [--no-zip]`
	-OR-
	`archive <event_log_path> <archive_name> --older-than <days> [--cleansource] [--dest-dir <destination_path>] [--no-zip]`
	Moves events that occurred before the indicated date (YYYY-MM-DD) to an archive-named `archive_name`. If you use the second form, specify the date as days before the current date.
	The archive is created in the current working directory unless you specify a destination path using the `--dest-dir` option. By default, the archive is compressed using `tar` and `gzip`, but you can skip this using the `--no-zip` option, in which case the resulting archive is a directory containing the new log with the archived events.
	All files in that directory are required to access the archive. To access the archive, use `pmlog`. Moving events to an archive may not reduce the actual file size of the event log. To reduce the file size, the source log must be cleaned. To clean the source log, add the `--clean-source` option. When a large number of events are present in the source log this option can increase the archive process time and use a large amount of disk space while the process runs. Once started, do not interrupt the process.

backup	Creates a backup of the source log (`event_log_path`), in location `backup_log`.
create	Creates new empty audit files for that log. `create <new_event_log_path>` This may include a keyfile which has the `-kf` suffix, a journal file with the `-wal` suffix, and a `-shm` system file. It is critical that the group of files that make up an event log remain together at all times. Removal of any one of these files may result in permanent loss of access to the event log.
encrypt	Enables or disables encryption of an event log. `encrypt enable|disable|rekey <event_log_path>`

Option	Description
	By default all event logs created by Privilege Manager for Unix are encrypted using the AES-256 standard. The encryption key is stored in the keyfile which is in the same path as the event log and has the same file name, and the `-kf` suffix. It is critical that this file remain in the same path as the main event log file. You can decrypt the whole log file using the encrypt disable command, passing the path of the main event log file as an argument. Enable encryption using `encrypt enable`. The `encrypt rekey` command generates a new encryption key and re-encrypt all data in the event log using that new key data. The key file is automatically updated with the new key data if the operation succeeds.
import	Imports events. `import [-y|-n] <source_event_log> <dest_event_log>` Import events from `source_event_log`, adding them to `dest_event_log`.
info	Displays information about the event log. `info <event_log_path>` Displays information about the `event_log_path`. The information reported includes the current encryption status of the event log, the size of the file and the number of events contained in the log.

Settings

The following entries in the `/etc/opt/quest/qpm4u/pm.settings` file are used by `pmlogadm`

Table 67: Settings: pmlogadm

Option	Description
Specify the location of the event log queue, used by both `pmmasterd` and `pmlogsrvd`. This option is only used to determine whether the `pmlogsrvd` service is currently running.	

For more usage information for a specific command, run:

```
pmlogadm help <command>
```

Files

The default Privilege Manager event log file is located at:

`/var/opt/quest/qpm4u/pmevents.db`

Other files that may be used by `pmlogadm` are:

- settings file: `/etc/opt/quest/qpm4u/pm.settings`
- pid file: `/var/opt/quest/qpm4u/evcache/pmlogsrvd.pid`

Related Topics

pmlog

pmlogsrvd

pmmasterd

## pmlogsearch

Syntax

```
pmlogsearch [--csv] [--no-sort]
               [--before "<YYYY/MM/DD hh:mm:ss>"] [--after "<YYYY/MM/DD hh:mm:ss>"]
               [--user <username>] [--host <hostname>] [--result accept|reject]
               [--text <keyword>]
               -h | --help
               -v | --version
```

Description

Use the `pmlogsearch` command to perform a search on all logs in this policy group based on specified criteria.

You must specify at least one search condition; you can combine conditions.

Options

`pmlogsearch` has the following options.

Table 68: Options: pmlogsearch

Option	Description
--csv	Outputs the search results in CSV format, suitable for consumption by Privilege Manager for Unix. If this option is not present, the output is human-readable. One or more of the search criteria must be present, and any combination of the criteria is accepted. When multiple criteria are present they must all be matched (that is, the query criteria are combined using AND logic) for a log to be included in the results.
--after --before	Returns logs generated for sessions initiated after or before the specified time and date. For example: `# pmlogsearch --after “2012/01/04 00:00:00”` returns all logs for sessions since January 4, 2012.

Option	Description
	`# pmlogsearch --after “2012/01/01 00:00:00” --before “2012/12/31 23:59”` returns all logs generated during 2012.
--user <username>	Searches for logs generated by sessions requested by the specified user name. `username` is case sensitive. For example: `# pmlogsearch --user harry` returns the locations of all keystrokelogs for sessions requested by the user named "harry". The pattern may include the following wild card symbols: • * = match any string • ? = match any single character

--host <hostname>	Searches for logs generated by sessions that ran on hosts matching the given pattern. The pattern may include the following wild card symbols:
	
	• * = match any string
	• ? = match any single character
	
	For example:
	
	`# pmlogsearch --host “myhost?.mydomain.com”`
	
	matches logs for sessions that ran on `myhost1.mydomain.com` or `myhost2.mydomain.com`, but not `myhost1` or `myhost10.mydomain.com`.
	
	`# pmlogsearch --host “myhost*”`
	
	matches logs for sessions that ran on `myhost1.mydomain.com`, `myhost2.mydomain.com`, `myhost1` or `myhost10.mydomain.com`, but will not match `anotherhost.mydomain.com`.
	
	`# pmlogsearch --host myhost11.mydomain.com`
	
	only matches logs for sessions that ran on host `myhost11.mydomain.com`.

--result	Returns only events with the indicated result.

--text “<keyword>”	Searches for events where the specified text occurs in the command line or events with keystroke logs that contain the specified text.
	
	You must enter the keyword or phrase as one argument. If the phrase contains a space, enclose the whole phrase in quotes. For example:
	
	`# pmlogsearch --text “my phrase”`
	
	matches any log containing the string "my phrase".
	
	`# pmlogsearch --text phone`
	
	matches logs containing any word with the substring `phone` (such as, telephone, headphones, phones), or the complete word phone.

Option	Description
--no-sort	Does not sort the results.
–v | --version	Displays the version number of Privilege Manager for Unix and exits.
-h | --help	Displays usage information and exits.

Output

You can output the search results in either human-readable or CSV format.

Human-Readable Output

The following is an example of the human-readable output of a search:

```
# pmlogsearch --user sheldon --text Linux
 Search matches 5 events
 2012/01/19 18:12:25 : Accept : sheldon@host1.example.com
     Request: sheldon@host1.example.com : uname -a
 Executed: root@host1.example.com : uname -a
     IO Log: pmsrv1.example.com: opt/quest/qpm4u/iologs/sheldon/root/uname-
20120119-181225.OiaiBr
 2012/01/19 18:11:56 : Accept : sheldon@host1.example.com
     Request: sheldon@host1.example.com : uname -a
 Executed: root@host1.example.com : uname -a
     IO Log: pmsrv2.example.com: opt/quest/qpm4u/iologs/sheldon/root/uname-
20120119-181156.x46qJP
 2012/01/19 17:59:09 : Accept : sheldon@host2.example.com
     Request: sheldon@host2.example.com : uname -a
 Executed: root@host2.example.com : uname -a
     IO Log: pmsrv2.example.com: opt/quest/qpm4u/iologs/sheldon/root/uname-
20120119-175909.1H0P5n
 2012/01/19 17:58:42 : Accept : sheldon@host2.example.com
     Request: sheldon@host2.example.com : uname -a
 Executed: root@host2.example.com : uname -a
     IO Log: pmsrv2.example.com: opt/quest/qpm4u/iologs/sheldon/root/uname-
20120119-175842.ZvfrMv
 2012/01/19 17:58:14 : Accept : sheldon@host2.example.com
     Request: sheldon@host2.example.com : uname -a
 Executed: root@host2.example.com : uname -a
     IO Log: pmsrv1.example.com: opt/quest/qpm4u/iologs/sheldon/root/uname-
20120119-175814.
```

CVS output

The results are output in CSV format, without field headings. The columns are listed in order below:

1. Session date/time
2. Session Unique ID
3. Master host
4. Submit host (host from which the session was requested)
5. Submit user (the user that requested the session)
6. Requested host
7. Requested user account
8. Requested command line
9. Result (Accept/Reject)
10. Run host (the host on which the command was run)
11. Run user (the user account used to run the command)
12. Command line that ran
13. The exit return code if the command ran successfully, or "NO_EXIT" if the event was rejected or the command failed to run
14. Keystroke log host. This column is blank, if it is the same as #3 Master host.
15. Keystroke log file path

The following is an example of CSV output:

```
# pmlogsearch --csv --user penny --text "Linux"
"2012/01/19 18:10:40", "4d3729207eec", "pmsrv1.example.com", "host1.example.com",
"penny", "uname", "Accept", "host1.example.com", "penny", "uname",
"pmsrv1.example.com", "opt/quest/qpm4u/iologs/host1.example.com/penny/uname-
20120119-181040.hLqZFY"
"2012/01/19 18:10:13", "4d3729057e5f", "pmsrv1.example.com", "host1.example.com",
"penny", "uname", "Accept", "host1.example.com", "penny", "uname",
"pmsrv1.example.com", "opt/quest/qpm4u/iologs/host1.example.com/penny/uname-
20120119-181013.yG1m41"
"2012/01/19 18:00:14", "4d3726ae1ec0", "pmsrv2.example.com", "host1.example.com",
"penny", "uname", "Accept", "host1.example.com", "penny", "uname",
"pmsrv2.example.com", "opt/quest/qpm4u/iologs/host1.example.com/penny/uname-
20120119-180015.Z42heZ"
"2012/01/19 18:00:47", "4d3726cf1f9d", "pmsrv1.example.com", "host1.example.com",
"penny", "uname", "Accept", "host1.example.com", "penny", "uname",
"pmsrv1.example.com", "opt/quest/qpm4u/iologs/host1.example.com/penny/uname-
20120119-180047.GUtrRt"
```

Related Topics

Viewing the log files using command line tools

pmlogsrvd

Syntax

```
pmlogsrvd [-d | --debug] [-h | --help] [--log-level <level>] [--no-detach]
          [--once] [-q | --queue <queue_path>] [--syslog [facility]]
          [-t | --timeout <delay_seconds>] [-v | --version] [-z on|off [:<pid>]]
```

Description

`pmlogsrvd` is the Privilege Manager for Unix log access daemon, the service responsible for committing events to the Privilege Manager for Unix event log, and managing the database storage used by the event log.

When an incoming event is processed by `pmmasterd` that event must be logged to the event log. `pmmasterd` commits a record of the log to the event log queue, which is monitored by `pmlogsrvd`. `pmlogsrvd` takes each event from the queue and commits that event to the actual event log.

Options

`pmlogsrvd` has the following options.

Table 69: Options: pmlogsrvd

Option	Description
-d | --debug	Enables debug operation. This option prevents `pmlogsrvd` from running in the background, and enables debug output to both the log and the terminal.
-h | --help	Displays the usage information and exits.
--log-level <level>	Controls the level of log messages included in the log file. By default the logging level logs only error messages. Valid logging levels, in ascending order by volume of messages, are: • none • error • warning • info • debug
--no-detach	Do not run in the background or create a pid file. By default, `pmlogsrvd` forks and runs as a background daemon. When you specify the `--no-detach` option, it stays in the foreground.

Option	Description
--once	Processes the queue once immediately and then exits.
-q | --queue <path>	Specifies the location of the event log queue as **path**.
--syslog	Enables logging to syslog.
-t | --timeout <delay_ seconds>	Specifies the time delay between processing the queue as time seconds. By default `pmlogsrvd` waits for 120 seconds before waking to scan the event log queue if no other trigger causes it to begin processing. Normally processing is triggered directly by `pmmasterd` immediately after an event is processed.
-v | --version	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

Settings

`pmlogsrvd` uses the following entries in the `/etc/opt/quest/qpm4u/pm.settings` file.

Table 70: Settings: pmlogsrvd

Setting	Description
eventLogQueue <pathname>	Specifies the location of the event log queue, used by both `pmmasterd` and `pmlogsrvd`. This setting is ignored by `pmlogsrvd` when you use the `--queue` option on the command line.
pmlogsrvlog <pathname>	Fully qualified path to the `pmlogsrvd` log file.
By default, `/pmlogsrvd/fR` used this setting to determine whether to send log messages to syslog. When you use the `/syslog/fR` option on the command line, this setting is ignored.	

Files

- settings file: `/etc/opt/quest/qpm4u/pm.settings`
- pid file: `/var/opt/quest/qpm4u/evcache/pmlogsrvd.pid`

Related Topics

pmlog

pmlogsearch

pmmasterd

pmmasterd

Syntax

```
pmmasterd [ -z on|off[:<pid>] ] [ -v ]| [ [ -ars ] [ -e <logfile> ] ]
```

Description

The Privilege Manager for Unix master daemon (`pmmasterd`) is the policy server decision-maker. `pmmasterd` receives requests from `pmrun` or the Sudo Plugin and evaluates them according to the security policy. If the request is accepted, `pmmasterd` asks `pmlocald` or the Sudo Plugin to run the request in a controlled account such as `root`.

A connection is maintained between `pmmasterd` and the Sudo Plugin for the duration of the session. This also occurs between `pmmasterd` and `pmlocald`, if keystroke logging is enabled. When the `pmmasterd` connection is maintained throughout the session, keystroke and event log data is forwarded on this connection.

If keystroke logging is not enabled, `pmlocald` reconnects to `pmmasterd` at the end of the session to write the event log record showing the final completion code for the command run by `pmlocald`. If `pmlocald` is unable to reconnect, it writes instead to a holding file, `pm.eventhold.hostname`. It then attempts to write the `pmevents.db` record to the host the next time `pmmasterd` connects to `pmlocald`. Multiple files can accrue and they will all be delivered to the proper host when the connection is restored.

The policy server master daemon typically resides on a secure machine. You can have more than one policy server master daemon on different hosts for redundancy or to serve multiple networks.

`pmmasterd` logs all errors in a log file if you specify the `-e filename` option.

Options

`pmmasterd` has the following options.

Table 71: Options: pmmasterd

Option	Description
-a	Sends job acceptance messages to syslog.
-e <filename>	Logs any policy server master daemon errors in the file specified.
-r	Sends job rejection messages to syslog.

Option	Description
-s	Sends any policy server master daemon errors to syslog.
-v	Displays the version number of pmmasterd and exits.
-z	Enables or disables tracing for this program and optionally for a currently running process. Refer to Enabling program-level tracing on page 179 before using this option.

Files

- Privilege Manager for Unix policy file (pmpolicy type): `/etc/opt/quest/qpm4u/policy/pm.conf`

Related Topics

pmcheck

pmkey

pmlocald

pmpasswd

pmreplay

pmrun

pmsum

pmmg

Syntax

```
pmmg /<full_path_name>
```

Description

The pmmg text editor is a special version of the mg text editor that you can use securely with Privilege Manager for Unix programs; it is a small version of gnu emacs with gnu-style emacs key bindings. You must specify a full pathname as an argument when starting pmmg. Also, you will not be able to access any files other than the ones you specified at startup time. Nor will you be allowed to spawn any processes.

When you the pmmg program with Privilege Manager for Unix, it allows you to access a specific file as root, but not other root functions.

pmpasswd

Syntax

```
pmpasswd
```

Description

The `pmpasswd` program generates an encrypted password which can be used in a custom configuration script. When you type `pmpasswd`, it asks you to type the password twice, then prints out the encrypted version. You can use the encrypted version as the first argument to the `getstringpasswd` function in the configuration file.

Related Topics

getstringpasswd

pmpolicy

Syntax

```
pmpolicy -v | -z on|off[:<pid>] command [args] [-c] [<command>.] -h
```

Description

`pmpolicy` is a command line utility for managing the Privilege Manager for Unix security policy. Use the `pmpolicy` command to view and edit the policy in use by the group. Any user in the `pmpolicy` group may run this command on any configured policy server host.

This utility checks out the current version, checks in an updated version, and reports on the repository.

You can use the `-c` option to display the result of the command in CSV, rather than in a human-readable form. The CVS output displays the following fields: Resultcode, name, description, Output msg.

The `pmpolicy` utility exits with the following possible exit status codes, unless otherwise stated below:

Exit status codes

- 0: Success
- 1: Repository does not exist
- 2: Specified path does not exist

- 3: Failed to checkout from the repository
- 4: Failed to check in to the repository
- 5: Syntax error found in new policy – check in was abandoned
- 6: Conflict found when attempting a check in – check in was abandoned
- 7: Policy type not found in repository
- 8: Failed to access the repository to report requested information
- 9: The selected version was not found in the repository
- 10: Directory did not contain a working copy
- 11: Check in abandoned
- 12: Invalid path specified
- 13: Invalid configuration

Options

The following is a summary of the commands and options available to `pmpolicy`.

Run any command with a `-h` to get more information about it. For example:

```
pmpolicy <command> -h
```

Table 72: Commands and options: pmpolicy

Command	Description
add	Adds a new file from the specified path to the policy repository. `add -p path -d dir [-n [-l commitmsg]] [-c] [-u <user>]` Records the addition of a new file to the working copy of the policy. Use the `-p` option to specify the file path (relative to the top-level directory in the policy) to add. Use the `-d` option to specify the directory of the working copy. The `-n` option commits the changes to the repository. If you use the `-n` option, you can also use the `-l` option to provide a commit log message. If you use `-n` without the `-l`, the command interactively prompts you for the commit log message
checkout	Checks out a working copy of the policy to the specified directory. `checkout -d <dir> [-c] [-r <revision>]` If the directory does not exist, it is created. If the selected directory exists, the existing contents is overwritten. By default, the latest copy is retrieved; use the `–r` option to check out a particular revision. You can specify a revision using SVN DATE format, or the HEAD keyword, as well as revision numbers.

Command	Description
	A date format specified without a time, defaults to 00:00:00. The earliest time you can use to identify a particular revision is one second after the time you commit the revision. For example, if you committed revision 2 at 12:00:00, then you must specify a time of 12:00:01 or later to check out revision 2. For example: `pmpolicy checkout -d /tmp -r "{2012-01-02 12:00:01}" # checkout revision that existed on 2012-01-02 00:00:00`

commit	Checks in changes from a working copy to the policy repository.
	
	`commit -d <dir> [-l <commitmsg>] [-c] [-a force|-abort|merge|overwrite][-u <user>]`
	
	Commits the working copy of the policy from the indicated directory. All files in the indicated directory are checked in to the repository.
	
	This working copy is first verified for syntax errors using the `pmcheck` utility. The working copy must match the policy type currently in use, otherwise a syntax error will be produced by `pmcheck`.
	
	If no syntax errors are encountered, it attempts to check in this copy into the repository, honoring the `-a` option as described below. Exit status of 0 indicates successful check in.
	
	The `–a` option indicates the action to be taken when checking in a working copy, if the repository has changed since the working copy was checked out, that is, the edits are based on an out-of-date copy of the repository. The resulting differences between the working copy and the repository may or may not conflict.
	
	You can specify the following actions:
	
	• **Merge**: If the only differences are non-conflicting, then merge the changes. If any conflicting changes are found, abort the check in.
	• **Overwrite**: Merge the changes. If any conflicting changes are found in the repository, select those from the working copy.
	• **Force**: Overwrite the copy in the repository with the working copy, discarding any changes that have been committed since the working copy was checked out.
	• **Abort**: Abandon the check in if the working copy is out of date, regardless of whether changes are in conflict (this is the default)
	
	For example:
	
	`pmpolicy commit -d /tmp -a force`

Command	Description

diff	Checks the differences between two revisions of the policy and reports the output to `stdout`, or to the selected output file.
	
	`diff [-o <outfile>][-c][-f][-p <path>][-d <dir> [-r <v1>]] | [-r [<v1>:[<v2>]]`
	
	By default, this option displays the differences between the two selected revisions. If you specify the `–f` option, it displays the incremental differences between each revision in the specified range. You can specify revisions using any acceptable SVN revision format, such as HEAD, COMMITTED, or DATE format. You can use the `–o` option to report the "diff" output to a file, rather than to `stdout` (the default).
	
	• If you specify a directory, it compares the copy in that directory with the selected revision (or the latest revision in the repository, if you do not specify a revision).
	• If you specify one revision, it reports the difference between the latest and selected revision.
	• If you specify two revisions, it reports the difference between the selected revisions.
	
	Exit status codes:
	
	• 0: no differences were detected.
	• 1: differences were detected
	• 2: An error occurred
	
	For example:
	
	`pmpolicy diff -d /tmp -o /tmp/diffs.txt -r2 pmpolicy diff –r1:2 -o /tmp/diffs.txt`

edit	The utility checks out a temporary working copy of the policy and starts the appropriate interactive editor to edit the files. `edit [-a force|abort|merge|overwrite] [-l <commitmsg>] [-p <path>][-u <user>]` This option is useful for manual interactive editing of the policy on the command line. On completion of the edit, it verifies the syntax of the policy. If no errors are found, it checks the edits back in to the repository. If any errors are found, then it exits without checking in the changes. When saving an edited policy, some non-ASCII characters in the commit log message may error and cause all changes to the policy to be discarded. To avoid this possibility, avoid using backspace, arrow

Command	Description
	keys and any other keys that may be interpreted as non-ASCII characters within the shell.
help	Displays usage information.

log	Logs revision information about the repository.
	
	`log [-o <outfile>][-c][-e][-r <revision>]`
	
	Reports information about the repository to `stdout` or to the selected output file. This displays details of the user who changed the repository, the version number for this change, along with the time and date of the change.
	
	By default, this option shows details of each revision in the repository, one version per line. If you specify a version, it shows the details of this version. You can use the `–o` option to report the "log" output to a file, rather than to `stdout`.
	
	The status is displayed in the following format for CSV output:
	
	`"<version>","<username>",<YYYY-MM-DD>,<HH:MM:SS>"<commitmsg>"`
	
	For example:
	
	`pmpolicy log -r 3`

masterstatus	Reports the status of the production copy of the policy used by Privilege Manager for Unix to authorize commands.
	
	`masterstatus [-o <outfile>] [-c]`
	
	The production copy is stored in the following directory by default:
	
	`/etc/opt/quest/qpm4u/policy/`
	
	You can use the `–o` option to report the information to a file instead of to `stdout`.
	
	It reports the following information:
	
	• Path to the production copy
	• Date and time the production copy was checked out
	• Revision number of the production copy
	• Latest trunk revision number of the repository
	• Locally modified flag (indicates that someone manually edited the file)

Command	Description
	The information is displayed in the following format for CSV output: `<path>,<YYYY/MM/DD>,<HH:MM><policyrevision>,<trunkrevision>,0|1`
remove	Removes a file from the specified path in the policy repository. `remove -p path -d dir [-n [-l <commitmsg>]] [-c] [-u <user>]` Removes a file from the indicated working copy directory. Use the `-p` option to specify a path to the file (relative to the top-level directory in the policy). Use the `-d` option to specify the directory of the working copy. The `-n` option commits the changes to the repository. If you use the `-n` option, you can also use the `-l` option to provide a commit log message. If you use `-n` without `-l`, the command interactively prompts you for the commit log message.
revert	Reverts to the selected revision of the policy. `revert [-c] [-r <version>][-l <commitmsg>]` Checks out a copy of the selected revision, edits the files, and checks the copy back in as the latest revision.
status	Verifies the working copy of the policy in the directory indicated. `status -d <dir> [-c]` Verifies the working copy of the policy in the specified directory. You can use this to verify the status of a working copy that was previously checked out, before attempting to commit any edits. Each file in the selected directory is checked against the latest version in the repository. For example: `pmpolicy status -d /tmp` **Exit status codes**: • 0: The working copy is up to date and has not been modified; no action is required. • 1: The working copy is up to date and has been modified; you must check in to commit the edits made in the working copy. To commit the changes, run: `pmpolicy commit -d <dir>` • 2: The working copy is out of date and has not been modified; You must check out to get an up-to-date copy of the policy before

Command	Description

	editing.
	To check out the latest copy, run:
	`pmpolicy checkout -d <dir>`
	• 3: The working copy is out of date and has been modified, but the changes do not conflict with the latest version. Therefore, a default check in will fail. To commit the you must use the `-a` option.
	To commit the changes, run:
	`pmpolicy commit -d <dir> -a merge`
	• 4: The working copy is out of date and has been modified and the changes conflict with the latest version, therefore a default check in will fail.
	To commit the changes and overwrite any conflicts with the working copy's changes run:
	`pmpolicy commit -d <dir> -a force`
	• 5: An error occurred when attempting to verify the status.

sync	Checks out the latest version to the production copy of the policy used by Privilege Manager for Unix to authorize commands. `sync [-f][-c]` Synchronize the local production copy of the policy with the latest revision in the repository.
-v	Displays the Privilege Manager for Unix version.
-z	Enables or disables debug tracing and optionally sends SIGHUP to a running process. Refer to Enabling program-level tracing on page 179 before using this option.

Related Topics

pmcheck

pmpolicyconvert

Syntax

```
pmpolicyconvert [-o <output dir>] [-v [-v]] path [paths...]
```

Description

The `pmpolicyconvert` utility allows you to verify, and if necessary, convert any number of policy files for use with Privilege Manager for Unix V5.5 (or later).

The `pmpolicyconvert` utility is a perl script that takes as input one or more policy files, and makes a copy of each file, performing any translation required to allow these files to be used in Privilege Manager for Unix.

`pmpolicyconvert` also warns about any variables and functions that are not applicable in Privilege Manager for Unix.

You can pass one or more files or directories as parameters to this utility. If a directory is specified, then `pmpolicyconvert` assumes it is to translate all files contained in that directory (and all subdirectories).

It copies the updated files to the specified output directory (mirroring the original directory structure if an entire directory is being translated). All changes are marked with a comment in the copied file.

A report is generated in the file `./ pmpolicyconvert _report.txt` that describes the changes made.

Options

`pmpolicyconvert` has the following options.

Table 73: Options: pmpolicyconvert

Option	Description
-h	Displays a usage message and exit.
-o <output dir>	Specifies an output directory to use. If not specified, the default is `./pm_policy`.
-v	Runs in verbose mode. Multiple –v options increase the verbosity. The maximum is two.
-V	Displays the version number of Privilege Manager for Unix and exits.

pmpolsrvconfig

Syntax

```
pmpolsrvconfig -p <policygroupname> [-b][-i <path>][-o][-r <dir>]
                 [-t sudo|pmpolicy] [-u <policyuser][-w <userpasswd>]
                 [-g <policygroup>][-l <loggroup>] -s <host> [-b][-q] [-q]
                  -a <user> [-b][-q] [-q]
                  -d [-f]
                  -e <host> [-f]
                  -x [-f]
                  -v
                  -h
                  -[-z on|off[:<pid>]]
```

Description

The `pmpolsrvconfig` program is normally run by `pmsrvconfig` script, not by the user, to configure or un-configure a primary or secondary policy server. But, you can use it to grant a user access to a repository.

Options

`pmpolsrvconfig` has the following options.

Table 74: Options: pmpolsrvconfig

Option	Description
-a <user>	Provides the selected user with access to the existing repository. If the user does not exist, it is created. The host must first have been configured as a policy server. This user will be added to the `pmpolicy` group to grant it read/write access to the repository files, and to the `pmlog` group to grant it read access to the log files. On a secondary policy server, an `ssh` key will also be generated to provide access to the `pmpolicy` user account on the primary policy server. The "join" password is required to copy this ssh key to the primary policy server.
-b	Runs the script in batch mode (that is, no user interaction is possible). Default: Runs in interactive mode.
-d	Unconfigures the policy server, and deletes the repository if this is a primary server.

Option	Description
	If you do not specify the `-f` option, then it prompts you to confirm the action.
-e <host>	Removes the selected host from the server group.
-f	Forces the unconfigure action (that is, no user interaction required) Default: Prompt for confirmation for `-x` option.
-g <policygroup>	Specifies the policy group ownership for the repository. If this group does not exist, it is created. Default: pmpolicy
-h	Prints help.
-i <path>	Imports the selected policy into the repository. If this is a directory, the entire contents of the directory will be imported. Default: `/etc/sudoers`.
-l <loggroup>	Specifies the `pmlog` group ownership for the keystroke and audit logs Default: pmlog
-o	Overwrites the repository if it already exists. Default: Does not overwrite if the repository already exists.
-p <policygroup>	Configures a primary policy server for the selected group name.
-q	Reads the pmpolicy user's password from stdin.
-r <dir>	Creates the repository in the selected directory. Default: `/var/opt/quest/qpm4u/.qpm4u/.repository`
-s <host>	Configures a secondary policy server. You must supply the primary policy server host name. The secondary policy server retrieves the details of the policy group from the primary policy server. It creates the `policygroup` and `loggroup` groups to match those on the primary policy server and configures the `policyuser` user to grant it `ssh` access to the repository on the primary server. The "join" password is required to copy this `ssh` key to the primary policy server.
-t sudo|p-mpolicy	Specifies the security policy type: sudo or pmpolicy. Default: sudo policy type
-u <policy-user>	Specifies the policy user account that manages the production copy. If this user does not exist, it is created and added to both the `policygroup` and `loggroup` groups. This user owns the repository on the primary policy server and provides remote access to the repository files to the secondary policy servers. Default: pmpolicy

Option	Description
-v	Prints the product version.
-w <userpasswd>	(Optional) Sets new user's password for `-a` option. Default: No password is configured.
-x	Unconfigures the policy server. If you do not specify the `-f` option, you are prompted to confirm the action. This does not remove the repository.
-z	Enables or disables debug tracing, and optionally send SIGHUP to a running process. Refer to Enabling program-level tracing on page 179 before using this option.

pmremlog

Syntax

```
pmremlog -v | -z on|off[:<pid>]
pmremlog -p pmlog|pmreplay|pmlogtxtsearch [-o <outfile>]
pmremlog [-h <host>] [-b] [-c] -- <program args>
```

Description

The `pmremlog` command provides a wrapper for the `pmlog` and `pmreplay` utilities to access the event (audit) and keystroke (I/O) logs on any server in the policy group. Anyone in the `pmlog` group can run this utility on the primary policy server.

Note that `pmlogtxtsearch` is a command located in `/opt/quest/libexec`.

Options

`pmremlog` has the following options.

Table 75: Options: pmremlog

Option	Description
-b	Disables interactive input and uses batch mode.
-c	Displays output in CSV, rather than human-readable format.
-h <host>	Specifies a host in the policy server group to access.
-o <outfile>	Saves the `pmlog` output to a file.

Option	Description
-p	Specifies program to run: • pmlog • pmreplay • pmlogtxtsearch
-v	Displays the Privilege Manager for Unix version number.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

Examples

To view the audit log on the primary policy server, enter:

```
pmremlog -p pmlog -- -f /var/opt/quest/qpm4u/pmevents.db
```

To view the audit events for user **fred** on secondary policy server **host1**, save the `pmlog` output to a file, and display the result of the `pmremlog` command in CSV format, enter:

```
pmremlog -p pmlog -c -o /tmp/events.txt -h host1 -- --user fred
```

To view the `stdout` from keystroke log **id_host1_x3jfuy**, on secondary policy server **host1**, enter:

```
pmremlog -p pmreplay -h host1 -- -o -f /var/opt/quest/qpm4u/iologs/id_
host1_x3jfuy
```

To retrieve the contents of keystroke log **id_host1_x3jfuy**, from secondary policy server **host1**, formatted for the `pmreplay` GUI, save the output to a temporary file, and display the result of the `pmremlog` command in CSV format, enter:

```
pmremlog -p pmreplay -h host1 -c -o /tmp/replay -- -zz -f
/var/opt/quest/qpm4u/iologs/id_host1_x3jfuy
```

pmreplay

Syntax

```
pmreplay -V
pmreplay -[t|s|i] -[Th] <filename>
pmreplay -[e][I][o] -[EhKTv] <filename>
pmreplay -z on|off[:<pid>]
```

Description

Use the pmreplay command to replay a log file to review what happened during a specified privileged session. The program can also display the log file in real time.

When using Privilege Manager for Unix, enable keystroke logging by configuring the iolog variable. If you are using the default profile policy, please consult global_variable.conf for details about configuring keystroke logging.

pmreplay can distinguish between old and new log files. If pmreplay detects that a log file has been changed, a message displays to tell you that the integrity of the file cannot be confirmed. This also occurs if you run pmreplay in real time and the Privilege Manager for Unix session that generated the events in the log file is active; that is, the client session has not completed or closed yet. In this case, the message does not necessarily indicate that the file has been tampered with.

The name of the I/O log is a unique filename constructed with the mktemp function using a combination of policy file variables, such as username, command, date, and time.

Privilege Manager for Unix sets the permissions on the I/O log file so that only root and users in the pmlog group can read it. That way, ordinary users cannot examine the contents of the log files. You must be logged in as root or be a member of the pmlog group to use pmreplay on these files. You may want to allow users to use Privilege Manager for Unix to run pmreplay.

By default pmreplay runs in interactive mode. Enter **?** to display a list of the interactive commands you can use to navigate through the file.

For example, replay a log file interactively by typing:

```
pmreplay /var/opt/quest/qpm4u/iolog/demo/dan/id_20130221_0855_gJfeP4
```

the results will show a header similar to this:

```
 Log File : /var/opt/quest/qpm4u/iolog/demo/dan/id_20130221_0855_gJfeP4 Date :
2013/02/21 Time : 08:55:17 Client : dan@sala.abc.local Agent : root@sala.abc.local
Command : id Type ’?’ or ’h’ for help
```

Type **?** or **h** at any time while running in interactive mode to display the list of commands that are available.

Options

`pmreplay` has the following options.

Table 76: Options: pmreplay

Option	Description
-e	Dumps the recorded standard error.
-E	Includes vi editing sessions when used with `-K`.
-h	When used with `-o` or `-I`, prints an optional header line. The header is always printed in interactive mode.
-i	Replays the recorded standard input.
-I	Dumps the recorded standard input, but converts carriage returns to new lines in order to improve readability.
-K	When used with `-e`, `-I`, and `-o`, removes all control characters and excludes `vi` editing sessions. Use with `-E` to include `vi` editing sessions.
-o	Dumps the recorded standard output.
-s	Automatically replays the file in slide show mode. Use **+** and **-** keys to vary the speed of play.
-t	Replays the file in tail mode, displaying new activity as it occurs.
-T	Displays command timestamps.
-v	Prints unprintable characters in octal form (\###)
-V	Displays the Privilege Manager for Unix version number.
-z	Enables or disables debug tracing. Refer to Enabling program-level tracing on page 179 before using this option.

Exit codes

`pmreplay` returns these codes:

- 1: File format error – Cannot parse the logfile.
- 2: File access error – Cannot open the logfile for reading
- 4: Usage error – Incorrect parameters were passed on the command line
- 8: Digest error – The contents of the file and the digest in the header do not match

Navigating the log file

Use the following commands to navigate the log file in interactive mode.

Table 77: Log file navigation shortcuts

Command	Description
g	Go to start of file.
G	Go to end of file.
p	Pause or resume replay in slide show mode.
q	Quit the replay.
r	Redraw the log file from start.
s	Skip to next time marker. Allows you to see what happened each second.
t	Display time of an action at any point in the log file.
u	Undo your last action.
v	Display all environment variables in use at the time the log file was created.
Space key	Go to next position (usually a single character); that is, step forward through the log file.
Enter key	Go to next line.
Backspace key	Back up to last position; that is, step backwards through the log file.
/<Regular Expression> **Enter**	Search for a regular expression while in interactive mode.
Repeat last search.	

Display the time of an action at any point in the log file with `t`, redraw the log file with `r`, and undo your last action with `u`.

You can also display all the environment variables which were in use at the time the log file was created using `v`. Use `q` or `Q` to quit `pmreplay`.

Type any key to continue replaying the I/O log.

pmresolvehost

Syntax

```
pmresolvehost -p|-v|[-h <hostname>] [-q][-s yes|no]
```

Description

The `pmresolvehost` command verifies the host name / IP resolution for the local host or for a selected host. If you do not supply arguments, `pmresolvehost` checks the local host name/IP resolution.

Options

`pmresolvehost` has the following options.

Table 78: Options: pmresolvehost

Option	Description
-h <hostname>	Verifies the selected host name.
-p	Prints the fully qualified local host name.
-q	Runs in silent mode; displays no errors.
-s	Specifies whether to allow short names.
-v	Displays the Privilege Manager for Unix version.

pmrun

Syntax

```
pmrun -v | -z on|off[<pid>] [-b][-d][-n][-p] [-m <masterhost>] [-h <hostname>]
        [-u <requestuser>] command [args]
```

Description

The `pmrun` command requests that an application is run in a controlled account. Simply add `pmrun` to the beginning of the command line. For example:

```
pmrun backup /usr dev/dat
```

`pmrun` checks the `/etc/opt/quest/pm.settings` file to determine which the policy server daemon to send the request. Once it has contacted a policy server daemon, it sends a request to the daemon to run the application specified. As with the `ssh` command, you can type **~^Z** to suspend `pmrun`, or **~.** to terminate it. You must enter these commands at the beginning of a new line.

Options

`pmrun` has the following options.

Table 79: Options: pmrun

Option	Description
-b	Allows the `runcommand` process to run in the background, permitting you to run other programs or commands from the same window. You can use the `-b` switch with any application process which does not require output that changes the `tty` mode. Because of this restriction, you can not use the `-b` switch with applications that require a password.
-d	The `-d` option is required if the application you are running uses the `nohup` command. Include the `-d` parameter to ensure that the `nohup` command functions correctly.
-h <hostname>	Allows you to request a particular execution host to run the request. Enter `-h <`**`host`**`>` before the command you are requesting.
-m <masterhost>	Allows you to select the policy server host to contact, bypassing the usual selection methods. The specified host must be in the `masters` setting in the `pm.settings` file.
-n	Redirects the input of `pmrun` to `/dev/null`. Use the `-n` option to avoid unfortunate interactions between `pmrun` and the shell which invokes it. For example, if you are running `pmrun` and start a `pmrun` in the background without redirecting its input away from the terminal, it will block even if no reads are posted by the remote command.
-p	Puts `pmrun` into pipe mode, in which all interactions with the user's terminal are done without changing any of the terminal parameters. Normally, `pmrun` puts the terminal into raw mode, so that programs such as text editors, which require raw mode, can run properly under `pmrun`. Pipe mode is useful when you need to pipe several `pmrun` commands together. For example: `pmrun -p ls /etc/secure | pmrun -p dbadd listing`
-u <requestuser>	Requests to run the command as the specified user. The policy server decides whether to honor this request.
-v	Displays the Privilege Manager for Unix version number and exits.
-z	Enables or disables tracing for this program and optionally for a currently running process. Refer to Enabling program-level tracing on page 179 before using this option.

Files

File containing Privilege Manager for Unix communication parameters, including the list of valid master hosts:

```
/etc/opt/quest/qpm4u/pm.settings
```

Related Topics

pmcheck

pmkey

pmlocald

pmmasterd

pmpasswd

pmreplay

pmsum

pmscp

Description

Use pmscp in conjunction with scp to launch the remote scp -t and scp -f daemons by means of pmrun -h. This allows you to use Privilege Manager for Unix to launch the remote scp daemons.

pmscp provides an alternate encryption channel for the scp command leaving authentication requirements to your Privilege Manager for Unix policy. Either put /opt/quest/bin in your PATH or use the absolute path.

Examples

To copy files to the /tmp directory on remote host, as root run the following:

```
scp -S pmscp <filename> user@remotehost:/tmp
```

pmserviced

Syntax

```
pmserviced [-d] [-n] [-s] [-v] [-z on|off[:<pid>]]
```

Description

The Privilege Manager for Unix service daemon, (pmserviced) is a persistent process that spawns the configured Privilege Manager for Unix services on demand. The pmserviced daemon is responsible for listening on the configured ports for incoming connections for the Privilege Manager for Unix daemons. It is capable of running the `pmmasterd`, `pmlocald`, `pmclientd`, and `pmtunneld` services.

Only one of `pmmasterd` and `pmclientd` may be enabled as they use the same TCP/IP port. See the individual topics in PM settings variables on page 286 for more information about these daemon settings.

Options

`pmserviced` has the following options.

Table 80: Options: pmserviced

Option	Description
-d	Logs debugging information such as connection received, signal receipt and service execution. By default, pmserviced only logs errors.
-n	Does not run in the background or create a pid file. By default, `pmserviced` forks and runs as a background daemon, storing its pid in `/var/opt/quest/qpm4u/pmserviced.pid`. When you specify the `-n` option, it stays in the foreground. If you also specify the `-d` option, error and debug messages are logged to the standard error in addition to the log file or syslog.
-s	Connects to the running `pmserviced` and displays the status of the services, then exits.
-v	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables tracing for `pmserviced`. Refer to Enabling program-level tracing on page 179 before using this option.

pmserviced Settings

`pmserviced` uses the following options in `/etc/opt/quest/qpm4u/pm.settings` to determine the daemons to run, the ports to use, and the command line options to use for each daemon.

Table 81: Options: pmserviced

Daemon Name	Flag to enable daemon	Listen on port	Command line options
pmclientd	pmclientdEnabled	masterport	pmclientdOpts
pmlocald	pmlocaldEnabled	localport	pmlocaldOpts
pmmasterd	pmmasterdEnabled	masterport	pmmasterdOpts
pmtunneldOpts			

Table 82: Settings: pmserviced

Setting	Description
pmservicedLog pathname | syslog	Fully qualified path to the `pmserviced` log file or syslog.
pmmasterdEnabled YES | NO	When set to YES, `pmserviced` runs `pmmasterd` on demand.
masterport number	The TCP/IP port `pmmasterd` or `pmclientd` uses to listen.
pmmasterdOpts options	Any command line options passed to `pmmasterd`.
pmlocaldEnabled YES | NO	When set to YES, `pmserviced` runs `pmlocald` on demand.
localport number	The TCP/IP port `pmlocald` uses to listen.
pmlocaldOpts options	Command line options passed to `pmmasterd`.
pmclientdEnabled YES | NO	When set to YES, `pmserviced` runs `pmclientd` on demand.
pmclientdOpts options	Any command line options passed to `pmclientd`.
pmtunneldEnabled YES | NO	When set to YES, `pmserviced` runs `pmtunneld` on demand.
tunnelport number	The TCP/IP port `pmtunneld` uses to listen.
pmtunneldOpts	Any command line options passed to `pmtunneld`.

Files

- settings file: `/etc/opt/quest/qpm4u/pm.settings`
- pid file: `/var/opt/quest/qpm4u/pmserviced.pid`

Related Topics

pmlocald

pmmasterd

pmsh

Syntax

```
pmsh -a|-b|-c <file>|-e|-f|-i|-m|-n|-o <option>|-s|-u|-v|-x|-C|-E|-I|-B|-V
     [-U <user>]
```

Description

The Privilege Manager for Unix Bourne Shell (pmsh) command is a fully featured version of sh, that provides transparent authorization and auditing for all commands submitted during the shell session. pmsh supports the standard options for sh.

Using the appropriate policy file variables, you can configure each command entered during a shell session, to be:

- forbidden by the shell without further authorization to the policy server
- allowed by the shell without further authorization to the policy server
- presented to the policy server for authorization

Once allowed by the shell, or authorized by the policy server, all commands run locally as the user running the shell program.

Options

pmsh has the following options.

Table 83: Options: pmsh

Option	Description
-a	Flags variables for export when assignments are made to them.
-b	Enables asynchronous notification of background job completion. (UNIMPLEMENTED) .
-B	Allows the shell to run in the background.
-c <file>	Reads commands from a file instead of from standard input.
-C	Does not overwrite existing files with `>'.
-e	Exits immediately if any untested command fails in non-interactive mode. The exit status of a command is considered to be explic- itly tested if the command is part of the list used to control an if, elif, while, or until; if the command is the left hand oper- and of an ``&&'' or ``||'' operator; or if the command is a pipe- line preceded by the ! operator. If a shell function runs and its exit status is explicitly tested, all commands of the function are considered to be tested as well.

Option	Description
-E	Enables the built-in emacs(1) command line editor (disables the -V option if it has been set; set automatically when interactive on terminals).
-f	Disables pathname expansion..
-h	A do-nothing option for POSIX compliance.
-i	Forces the shell to behave interactively.
-I	Ignores EOF's from input when in interactive mode.
-m	Turns on job control (set automatically when interactive).
-n	If not interactive, reads commands but do not run them. This is useful for checking the syntax of shell scripts.
-o <option>	Sets the specified shell option. A list of shell options can be displayed using the set -o builtin command.
-s	Reads commands from standard input (set automatically if no file arguments are present). This option has no effect when set after the shell has already started running (i.e., when set with the set command).
-u	Writes a message to standard error when attempting to expand a variable, a positional parameter or the special parameter ! that is not set, and if the shell is not interactive, exit immediately.
-v	The shell writes its input to standard error as it is read. Useful for debugging.
-V	Enables the built-in vi command-line editor (disables `-E` if it has been set).
-x	Writes each command (preceded by the value of the PS4 variable subjected to parameter expansion and arithmetic expansion) to standard error before it is run. Useful for debugging.

`pmsh` supports the following builtin commands:

```
., :, [, alias, bg, break, cd, chdir, command, continue, echo, eval, exec,
exit, export, false, fg, getopts, hash, jobs, kill, local, printf, pwd, read,
readonly, return, set, shift, test, times, trap, true, type, ulimit, umask,
unalias, unset, wait
```

pmshellwrapper

Syntax

```
pmshellwrapper
```

Description

Use the `pmshellwrapper` program as a wrapper for any valid login shell on a host. It provides full keystroke logging for any normal shell, but does not provide authorization of the commands run from the shell.

To use `pmshellwrapper`, you must create a link for the real shell you want to use. For example:

```
ln –s /opt/quest/libexec/pmshellwrapper
/opt/quest/bin/pmshellwrapper_bash
```

When the user runs `pmshell_bash`, it transparently converts this to `pmrun bash`.

pmsrvcheck

Syntax

```
pmsrvcheck --csv [ --verbose ] | --help | --pmpolicy | --primary | --secondary
```

Description

Use `pmsrvcheck` to verify that a policy server is setup properly. It produces output in either human-readable or CSV format similar to that produced by the preflight program.

The `pmsrvcheck` command checks:

- that the host is configured as a primary policy server and has a valid repository
- has a valid, up-to-date, checked-out copy of the repository
- has access to update the repository
- has a current valid Privilege Manager for Unix license
- `pmmasterd` is correctly configured
- `pmmasterd` can accept connections

`pmsrvcheck` produces output in either human-readable or CSV format similar to the pre-flight output.

Options

`pmsrvcheck` has the following options.

Table 84: Options: pmsrvcheck

Option	Description
--cvs	Displays csv, rather than human-readable output.
--help	Displays usage information.
--pmpolicy	Verifies that Privilege Manager for Unix policy is in use by the policy servers.
--primary	Verifies a primary policy server.
--secondary	Verifies a secondary policy server.
--verbose	Displays verbose output while checking the host.
--version	Displays the Privilege Manager for Unix version number and exits.

Files

- Settings file: `/etc/opt/quest/qpm4u/pm.settings`

Related Topics

pmmasterd

pmsrvconfig

Checking the policy server

pmsrvconfig

Syntax

```
pmsrvconfig -h | --help [-abipqtv] [-d <variable>=<value>] [-f <path>]
            [-l <license_file>]
            [-m sudo | pmpolicy] [-n <group_name> | -s <hostname>]
            [-x [<policy_server_host> ...]] [-bpvx] -u [--accept] [--batch]
            [--define <variable>=<value>] [--import <path>] [--interactive]
            [--license <license_file>]
           [--name <group_name> | --secondary <hostname>]
           [--pipestdin] [--plugin] [--policymode sudo | pmpolicy]
        [--selinux] [--tunnel]
           [--unix [<policy_server_host> ...]]  [--verbose] [--batch]
         [--unix] [-- verbose] --unconfig -N policy_name [--policyname policy_name]
```

Description

Use the `pmsrvconfig` command to configure or reconfigure a policy server. You can run it in interactive or batch mode to configure a primary or secondary policy server.

Options

`pmsrvconfig` has the following options.

Table 85: Options: pmsrvconfig

Option	Description
-a | --accept	Accepts the End User License Agreement (EULA), `/opt/quest/qpm4u/qpm4u_eula.txt`.
-b | --batch	Runs in batch mode; does not use colors or require user input.
-d <variable>=<value> | --define <variable>=<value>	Specifies a variable for the `pm.settings` file and its associated value.
-h | --help	Displays usage information.
-i | --interactive	Runs in interactive mode; prompts for configuration parameters instead of using the default values.
-f <path> | --import <path>	Imports policy data from the specified path. • Privilege Manager for Unix: The path may be set to either a file or a directory when using the pmpolicy type. • Safeguard for Sudo: The path must be set to a file when using the sudo policy type.
-l | --license <license_file>	Specifies the full pathname of an .xml license file. You can specify this option multiple times with different license files.
-m sudo | pmpolicy | --policymode sudo | pmpolicy	Specifies the type of security policy: • sudo • pmpolicy Default: sudo

Option	Description
-n | --name <group_name>	Uses `group_name` as the policy server group name.
-q | --pipestdin	Pipes password to stdin if password is required.
-s | --secondary <hostname>	Configures host to be a secondary policy server where hostname is the primary policy server.
-S | --selinux	Enable support for SELinux in Privilege Manager for Unix. An SELinux policy module will be installed, which allows the pmlocal daemon to set the security context to that of the run user when executing commands. This requires that the policycoreutils package and either the selinux-policy-devel (RHEL7 and above) or selinux-policy (RHEL6 and below) packages be installed.
-t | --tunnel	Configures host to allow Privilege Manager for Unix connections through a firewall. This option is only available when using the pmpolicy policy type (Privilege Manager for Unix).
-u | --unconfig	Unconfigures a Privilege Manager for Unix server.
-v | --verbose	Displays verbose output while configuring the host.
-x | --unix [policy_server_host ...]	Configures Privilege Manager for Unix on the local policy server; that is, configures `pmlocald` and `pmrun` to run on this host. If you do not specify a policy server host, it uses the local host name. This option is only available when using the pmpolicy policy type (Privilege Manager for Unix).

Examples

The following example accepts the End User License Agreement (EULA) and imports the sudoers file from `/root/tmp/sudoers` as the initial policy:

```
# pmsrvconfig –a –f /root/tmp/sudoers
```

By using the `–a` option, you are accepting the terms and obligations of the EULA in full.

By default, the primary policy server you configure uses the host name as the policy server group name. To provide your own group name, use the `–n` command option, like this:

```
# pmsrvconfig –a –n <MyPolicyGroup>
```

where `<MyPolicyGroup>` is the name of your policy group.

See Configuring the primary policy server for Privilege Manager for Unix on page 28 and Policy servers are failing on page 180 for other usage examples.

Files

Directory where pmsrvconfig logs are stored: `/opt/quest/qpm4u/install`

Related Topics

pmrun

pmjoin

pmlocald

pmmasterd

pmpolicy

pmsrvinfo

Syntax

```
pmsrvinfo [--csv] | -v
```

Description

Use the `pmsrvinfo` command to display information about the group in either human readable or CSV format. You can run this program on any server in the policy group.

Options

`pmsrvinfo` has the following options.

Table 86: Options: pmsrvinfo

Option	Description
--csv	Displays information in .CSV format, instead of human readable output.
-v	Displays the Privilege Manager for Unix version number and exits.

Examples

```
# pmsrvinfo
```

```
Policy Server Configuration:
----------------------------
Privilege Manager for Unix version   : 6.0.0 (nnn)
Listening port for pmmasterd daemon    : 12345
Comms failover method                  : random
Comms timeout(in seconds)              : 10
Policy type in use                     : pmpolicy
Group ownership of logs                : pmlog
Group ownership of policy repository   : pmpolicy
Policy server type                     : primary
Primary policy server for this group   : adminhost1
Group name for this group              : adminGroup1
Location of the repository             :
file:////var/opt/quest/qpm4u/.qpm4u/.repository/pmpolicy_repos/trunk
Hosts in the group                     : adminhost1 adminhost2
```

Related Topics

Policy servers are failing

pmstatus

Syntax

```
pmstatus [-v] [-p <port>] [-h <hostname>] [-f <hostfile>] [-o <outfile>]
```

Description

The `pmstatus` program checks connectivity between Privilege Manager for Unix and `pmlocald` and `pmmasterd` on the specified hosts. You must specify at least one host, using either the `-h` or `-f` option.

Options

`pmstatus` has the following options.

Table 87: Options: pmstatus

Option	Description
-f <hostfile>	Specifies the name of a file containing a list of hosts to check.
-h <hostname>	Specifies the name of the host to check. -h supercedes -f if you specify both options.
-o <outfile>	Writes status information to the specified file.
-p <port>	Specifies an alternative port to use when checking for connectivity with pmmasterd.
-v	Displays version information for the pmstatus program.

Examples

The following is an example of the output from pmstatus, if the command is directed at a host that is contactable and that contains Privilege Manager for Unix components:

```
[root@sdfbs02p linux-intel]# ./pmstatus -h sdfbs07p
Master process on sdfbs07p:12345 responded
Agent process on sdfbs07p:12346 responded
```

The following is an example of the output from pmstatus, if the command is directed at a host that is contactable, but does not contain any Privilege Manager for Unix components:

```
[root@sdfbs02p linux-intel]# ./pmstatus -h sdfbs07p
pmstatus5.0.2 (006): 3003 Could not connect to a master daemon for sdfbs07p
No master process responded on sdfbs07p:12345
pmstatus5.0.2 (006): 3001 Connection to pmlocald on sdfbs07p failed:
Connection refused
No agent process responded on sdfbs07p:12346
```

pmsum

Syntax

```
pmsum /<full_path_name>
```

Description

Use `pmsum` to generate a checksum of the named file. The output it produces can be used in a policy with the `runcksum` variable. If the requested binary/command does not match the checksum, it rejects the command.

Options

`pmsum` has the following options.

Table 88: Options: pmsum

Option	Description
-v	Prints the version number of Privilege Manager for Unix and exits.

Examples

```
# pmsum /bin/ls
5591e026 /bin/ls
```

Related Topics

runcksum

pmsysid

Syntax

```
pmsysid [-i] | -v
```

Description

The `pmsysid` command displays the Privilege Manager for Unix system ID.

Options

`pmsysid` has the following options.

Table 89: Options: pmsysid

Option	Description
-i	Shows the system host name and IP address.
-v	Displays the Privilege Manager for Unix version and exits.

pmtunneld

Syntax

```
pmtunneld [ [-v] | [-z on|off[:<pid>]] | [[-e <logfile>] [-s] ] ]
```

Description

The `pmtunneld` command acts as a proxy for `pmrun` when `pmlocald` communicates with `pmrun` through a firewall.

Communication sent from `pmlocald` is transmitted using port number 12347, by default, and received by `pmtunneld`. `pmtunneld` then transmits the data to `pmrun`. See Configuring pmtunneld on page 143 for details.

Options

`pmtunneld` has the following options.

Table 90: Options: pmtunneld

Option	Description
-e <logfile>	Logs any tunnel proxy daemon errors in the file specified.
-s	Sends any tunnel proxy daemon errors to syslog.
-v	Displays the version number of Privilege Manager for Unix and exits.
-z	Enables or disables tracing for this program and optionally for a currently running process. Refer to Enabling program-level tracing on page 179 before using this option.

pmumacs

Syntax

```
pmumacs /<full_path_name>
```

Description

The pmumacs text editor is a special version of microemacs that you can use securely with Privilege Manager for Unix programs; it is similar to the umacs editor. umacs is a small version of emacs with gosling-style emacs key bindings. You must specify a full path name as an argument when starting pmumacs. Also, you will not be able to access any files other than the ones you specified at startup time nor spawn any processes.

Use pmumacs to allow users to access a specific file as root but no other root functions.

pmverifyprofilepolicy

Syntax

```
pmverifyprofilepolicy [-v | [-c][-z on|off[:<pid>]]] [-f <filename>]
                      [-p <policydir>]
```

Description

Use pmverifyprofilepolicy to verify the syntax and structure of the policy file and check whether a particular command will be accepted or rejected. The policy is assumed to match the format of the default profile policy; if it is not in the expected format, then it displays an error for each file that is missing or is not in the correct format.

Options

pmverifyprofilepolicy has the following options.

Table 91: Options: pmverifyprofilepolicy

Option	Description
-c	Displays output in csv, rather than human-readable, format. The following line displays for each syntax error encountered: PMCHECKERROR,<**filename**>,<**linenumber**>,<**error_description**> The overall result displays in the following format:

Option	Description

	`PMVERIFYPROFILERESULT,<`**`result`**`>,<`**`description`**`>`
	
	where **result** can be: `0:success` or `-1:fail`
	
	For each file expected to contain data only, it prints the following line to stdout for each statement found in the file that is not a comment or variable assignment:
	
	`PMVERIFYPROFILECHECK,<`**`filename`**`>,<`**`linenumber`**`>,<`**`description`**`>`
	
	For each file expected to be unchanged, it prints the following line to stdout:
	
	`PMVERIFYPROFILENOMATCH,<`**`filename`**`>,<`**`linenumber`**`>,<`**`description`**`>`

-f <filename>	Provides an alternative policy filename to check. If not fully qualified, this path is interpreted as relative to the `policydir`, rather than to the current directory.
-p <policydir>	Forces `pmverifyprofilepolicy` to search for a different policy directory for include files identified by relative path. The default location is the `policydir` setting in `pm.setting`.
-v	Prints the Privilege Manager for Unix version and exits.
-z	Enables or disables debug tracing, and optionally sends SIGHUP to running process. Refer to Enabling program-level tracing on page 179 before using this option.

pmvi

Syntax

```
pmvi /<full_path_name>
```

Description

The `pmvi` editor is a special version of `vi` that you can use securely with Privilege Manager for Unix programs. You must specify a full path name as an argument when starting `pmvi`. Also, you will not be able to access any files other than the ones you specified at startup time nor spawn any processes.

Use `pmvi` to allow users to access a specific file as `root` but no other `root` functions.

Appendix F

Installation Packages

Privilege Manager for Unix is comprised of the following packages:

- **Privilege Manager for Unix product**

  Contains the Privilege Manager for Unix Policy Server and PM Agent components and uses the native packaging system for each platform (RPM, PKG, etc).
- **Safeguard for Sudo product**

  Contains the Safeguard Policy Server and Sudo Plugin components and uses the native packaging system for each platform (RPM, PKG, etc).
- **Preflight Binary**

  This is a stand-alone native binary for each platform (not zipped, tarred or packaged). This binary exists stand-alone on the ISO to make it available for use prior to installing software. It does not change any Privilege Manager for Unix configuration on the host.

For more information, see Downloading Privilege Manager for Unix software packages on page 22.

Package locations

Privilege Manager for Unix is provided in native platform install packages, which include binary files, online man pages, installation files, and configuration file examples.

The install packages are located in the zip archive in two directories called:

- /server
- /agent
- /sudo_plugin

where <platform> is the name of the platform on which you are running Privilege Manager for Unix.

There are three different packages:

- `qpm-agent` package, which contains only the client (`pmrun`) and agent (`pmlocald`) components for Privilege Manager for Unix.
- `qpm-server` package, which contains the server (`pmmasterd`), the client (`pmrun`) and agent (`pmlocald`), and the Sudo Plugin (`qpm4u_plugin.so`) components for Privilege Manager for Unix.
- `qpm-plugin` package, which contains the offline policy cache server (`pmmasterd`), the Sudo Plugin (`qpm4u_plugin.so`) components for Privilege Manager for Unix.

The Solaris server and agent packages have filenames that start with `QSFTpmsrv` and `QSFTpmagt`, respectively.

Once installed, the packaged files are placed in an installation directory under `/opt/quest` which contains subdirectories and files.

The platform directories contain the Privilege Manager for Unix installer packages for each platform supported by Privilege Manager for Unix.

Table 92: Privilege Manager kit directories

Platform	Architecture
aix71-rs6k	IBM®AIX 7.1, 7.2
freebsd-x86_64	FreeBSD on x86 64-bit architecture
hpux-hppa11	HP-UX 11.31 PA-RISC architecture
hpux11-ia64	HP-UX 11.31 Itanium architecture
linux-aarch64	Linux on ARM 64-bit architecture
linux-ia64	Linux on Itanium architecture
linux-intel	Linux x86
linux-ppc64	Linux on ppc little endian 64-bit architecture
linux-ppc64le	Linux on ppc little endian 64-bit architecture
linux-s390	Linux s390
linux-x86_64	Linux on x86 64-bit architecture
macos-x86_64	macOS on x86 64-bit architecture
Solaris-intel	Solaris Intel architecture
Solaris-sparc	SolarisSPARC® architecture

Installed files and directories

The following table lists files and directories installed on your system.

Table 93: Installed files and directories

Directories and files	Description	Created by
/opt/quest/qpm4u	Install directory containing readme, default trial license file, examples directory, templates, etc.	INSTALL
/etc/opt/quest/qpm4u/pm.settings	Configuration file for Privilege Manager for Unix component communications.	CONFIG
/etc/opt/quest/qpm4u/policy/pm.conf	Default production policy file when using the pmpolicy policy type.	CONFIG
/etc/opt/quest/qpm4u/policies	Default production policy framework directory when using the pmpolicy type.	CONFIG
/etc/opt/quest/qpm4u/policies/sudoers	Default production policy file for the sudo policy type.	CONFIG
/opt/quest/bin	Install directory containing the binaries for user programs, such as `pmrun`, `pmksh` and `pmvi`. These user programs only apply to Privilege Manager for Unix.	CONFIG
/opt/quest/sbin	Install directory containing the binaries for admin programs, such as `pmlog` and `pmreplay`.	INSTALL
/opt/quest/lib	Install directory for shared libraries	INSTALL
/opt/quest/libexec	Install directory for dynamically loaded objects.	INSTALL
/opt/quest/man	This directory contains all the man pages for Privilege Manager for Unix daemons and programs.	INSTALL
/opt/quest/qpm4u/examples	This directory contains useful programs, scripts, or examples which show how to use Privilege Manager for Unix. It also contains a sample configuration file which you can use as a template for implementing your own policies. These scripts and examples only	INSTALL

Directories and files	Description	Created by
	apply to Privilege Manager for Unix.	
/opt/quest/qpm4u/license	This file contains the license information (policy server only). For information about updating license information, see pmlicense on page 413.	INSTALL
/opt/quest/qpm4u/qpm4u_eula.txt	This file contains the End User License Agreement for the Privilege Manager for Unix product.	INSTALL
/opt/quest/qpm4u/README. <architecture>	This file contains the latest information about your version of Privilege Manager for Unix.	INSTALL
/var/opt/quest/qpm4u/iolog	This directory contains the keystroke logs.	EVENTDATA
/var/opt/quest/qpm4u/pmevents.db	This file contains the event logs.	EVENTDATA

About us

One Identity solutions eliminate the complexities and time-consuming processes often required to govern identities, manage privileged accounts and control access. Our solutions enhance business agility while addressing your IAM challenges with on-premises, cloud and hybrid environments.

Contacting us

For sales and other inquiries, such as licensing, support, and renewals, visit https://www.oneidentity.com/company/contact-us.aspx.

Technical support resources

Technical support is available to One Identity customers with a valid maintenance contract and customers who have trial versions. You can access the Support Portal at https://support.oneidentity.com/.

The Support Portal provides self-help tools you can use to solve problems quickly and independently, 24 hours a day, 365 days a year. The Support Portal enables you to:

- Submit and manage a Service Request
- View Knowledge Base articles
- Sign up for product notifications
- Download software and technical documentation
- View how-to videos at www.YouTube.com/OneIdentity
- Engage in community discussions
- Chat with support engineers online
- View services to assist you with your product

Index