

rPath Appliance Platform Agent Customization Guide

3.0.2.1-2008-09-11

rPath Appliance Platform Agent Customization Guide:

3.0.2.1-2008-09-11

Copyright © 2008 rPath, Inc.

rPath, Inc.
701 Corporate Center Drive, Suite 450
Raleigh, North Carolina 27607
USA

rPath, rBuilder, rPath Appliance Platform, and Conary are registered trademarks of rPath, Inc. All other trademarks and service marks are property of their respective owners.

1. Introduction	9
What You Should Already Know	9
What You Need	9
2. Overall rAPA Structure	11
Open-source Technologies in rAPA	11
Two Parts of rAPA: Service and Interfaces	11
rAPA Interfaces	11
rAPA Task Scheduler	11
rAPA Plugin Structure	12
rAPA Configuration Structure	13
3. Include rAPA in an Appliance	15
Select Which rAPA to Include	15
Add rAPA to the Appliance Group Recipe	15
Shadow and Use a Template Customization Package	16
Use a Testing Environment for Customizations	17
4. Branding the rAPA Interface	19
Appliance Information Text	19
Vendor Text for the Status Page and Collection Tool	20
Replacement Logos and Other Graphics	21
Custom CSS	21
5. Available Tasks and the Configuration Wizard	23
Identify the Plugin Associated with a Task	23
Set Available Tasks in rAPA	23
Enable and Disable Tasks	23
Hide Enabled Tasks	24
Override Default Tasks	24
Adjust the Initial Configuration Wizard	25
6. Custom Help Pages in rAPA	27
7. Task-specific Modifications for rAPA	29
8. External Authentication for rAPA	31
Identity Provider Object	31
rAPA Configuration for Using External Authentication	31
9. Set Up an XML-RPC Client for rAPA	33
XML-RPC Python Client	33
XML-RPC Java Client	34
XML-RPC for Segmented Updates	34
10. Maintaining Packaged Customizations	37
11. Standard Plugins in rAPA	39
Standard Plugin: Appliance Status and System Information	41
Appliance Status and System Information: Custom Configuration	41
Appliance Status and System Information: XML-RPC	41
Standard Plugin: Back Up and Restore	42
Back Up and Restore: Sequences	42
Back Up and Restore: Backup Targets	43
Back Up and Restore: Data to be Backed Up	43
Back Up and Restore: Backup and Restore Scripts	44
Back Up and Restore: Custom Configuration	45
Back Up and Restore: XML-RPC	47
Standard Plugin: Change Password	47

Change Password: Custom Configuration	47
Change Password: XML-RPC	48
Standard Plugin: Collection Tool	49
Collection Tool: Data to be Collected	49
Collection Tool: Custom Configuration	49
Collection Tool: XML-RPC	50
Standard Plugin: Configuration	51
Standard Plugin: Configuration, Configure Networking	51
Configure Networking: Hostname Change Scripts	51
Configure Networking: Custom Configuration	51
Configure Networking: XML-RPC	52
Standard Plugin: Configure Notification	52
Configure Notification: Custom Configuration	52
Configure Notification: XML-RPC	53
Standard Plugin: Configuration, Configure Proxy	53
Configure Proxy: Custom Configuration	53
Configure Proxy: XML-RPC	53
Standard Plugin: Configuration, System Time	54
System Time: Custom Configuration	54
System Time: XML-RPC	54
Standard Plugin: Configuration, Manage Entitlements	54
Manage Entitlements: Include entitlements.xml	54
Manage Entitlements: Custom Configuration	55
Entitlements: XML-RPC	55
Standard Plugin: Configuration, Upload SSL Certificate	55
Upload SSL Certificate: Custom Configuration	55
Upload SSL Certificate: XML-RPC	56
Standard Plugin: Appliance Logs	56
Appliance Logs: Custom Configuration	57
Appliance Logs: XML-RPC	58
Standard Plugin: Schedule Reboot	58
Schedule Reboot: Custom Configuration	58
Schedule Reboot: XML-RPC	58
Standard Plugin: Rollbacks	58
Rollbacks: Custom Configuration	58
Rollbacks: XML-RPC	59
Standard Plugin: Manage Services	59
Manage Services: Custom Configuration	59
Manage Services: XML-RPC	60
Standard Plugin: Updates	60
Updates: Configure the Number of Kernels to Keep	61
Updates: Show or Hide the Migrate Tab	61
Updates: Display Extended Information	61
Updates: Offline Updates Using External Media	63
Updates: Custom Configuration	63
Updates: XML-RPC	65
Standard Plugin: User Management	65
User Management:Roles	66
User Management: Custom Configuration	66

User Management: XML-RPC	66
Standard Plugin: Flip-flop Update	67
Flip-flop Update: Partitions and Kickstart	68
Flip-flop Update: Generate Update Images	70
Flip-flop Update: Copy Data Between Partitions	70
Flip-flop Update: Signed Update Images	71
Flip-flop Update: Custom Configuration	72
Flip-flop Update: XML-RPC	74
Standard Plugin: Factory Reset	74
Factory Reset: Images	74
Factory Reset: Configure	74
Factory Reset: XML-RPC	76

Chapter 1. Introduction

rPath offers its customers products and services to build appliances based on the rPath Appliance Platform. Part of the platform is the rPath Appliance Platform Agent (rAPA) which provides an interface for performing initial configuration and ongoing maintenance on the deployed appliance. rAPA is made up of a server and exposed interfaces, and there are options to use either its native web interface or its XML-RPC calls to perform the administrative tasks.

One of the most powerful features of rAPA is the ability to customize it and extend it so that it provides a complete administrative interface for all appliance configuration and maintenance needs. rPath provides two documents for working with rAPA:

- *rPath Appliance Platform Agent Customization Guide* -- This guide introduces the default rAPA interface and provides instructions for packaging a custom configuration to brand the interface, modify the available functions, and override default values. This includes ensuring appliances that are using the rPath Appliance Platform Entitlement Service (rES) have the necessary `entitlements.xml` file included as part of the appliance.
- *rPath Appliance Platform Agent Plugin Development Guide* -- This guide describes the structure of rAPA with an emphasis on its extensible nature, and it provides instructions for developing a custom *plugin* from a basic template and packaging that plugin to be part of an appliance. Use the following URLs to access this guide online or download a PDF version of it:

http://docs.rpath.com/rAPA_Plugin_Development_Guide/rAPA_Plugin_Development_Guide/index.html (HTML)

[<http://docs.rpath.com/>

http://docs.rpath.com/rAPA_Plugin_Development_Guide.pdf

What You Should Already Know

This guide assumes the reader has some basic appliance development and Canary packaging experience. If you are using this guide for the first time, and have not yet worked on your own appliance development with rPath products and services, rPath recommends either attending a training class, or just starting out with the self-paced *Application to Appliance: A Hands-on Guide*, including an optional pre-configured development environment and available as a PDF download at the following URL:

http://wiki.rpath.com/wiki/Application_to_Appliance

What You Need

Before customizing rAPA for an appliance, you should have an environment already configured for conducting development work on your appliance. This guide assumes you are able to do the following in that environment:

- Shadow a package from one Canary label to another
- Check out and modify packages in Canary
- Build Canary packages using rMake
- Check in a package to a product repository in rBuilder
- Add packages to install as part of an appliance

What You Need

- Use a test image of an appliance to test packaged software

Chapter 2. Overall rAPA Structure

The rPath Appliance Platform Agent is written in the Python programming language and makes use of existing open-source technologies to provide its functions for configuration and maintenance for an appliance. The following sections introduce the technologies and outline the basic structure of rAPA.

Open-source Technologies in rAPA

rAPA is build on an powered by open-source technologies, enhancing its extensible nature while providing a solid web application foundation. The following technologies are used in rAPA:

- *Python* -- Dynamic object-oriented programming language which is loosely-typed, has extensive standard libraries, and has strong support for integration with other languages and tools; to read more about Python, go to: <http://www.python.org>
- *Kid template framework* -- Templating language for XML written in Python, focused on web interface layout and presentation; to read more about Kid, go to: <http://www.kid-templating.org>
- *CherryPy* -- HTTP framework written in Python, handling everything from backend database activities to web interface layout; to ready more about CherryPy, go to: <http://www.cherrypy.org>

Two Parts of rAPA: Service and Interfaces

The rPath Appliance Platform Agent consists of two parts: a server and exposed interfaces. The server part is driven by a task scheduler, and the interfaces are driven by HTTP and JSON. The plugin structure within rAPA makes use of each of these two parts for each plugin. The following sections describe these parts further.

rAPA Interfaces

Each of the functions within rAPA are provided through one of two interfaces: a web interface and XML-RPC. The web interface uses HTTP and HTTPS, accessibly by any W3 standard-compliant web browser, and a JSON JavaScript encoder/decoder for Python. The XML-RPC interface is accessible by an XML-RPC client configured to make the appropriate remote calls to rAPA.

For XML-RPC, any XML-RPC client can be created or modified to send calls to rAPA to perform the same tasks that are built in to the web interface. See the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] for information about how to develop an XML-RPC client to interact with rAPA, and how to identify and use the XML-RPC exposed functions in rAPA.

rAPA Task Scheduler

Tasks in rAPA are completed through the use of a scheduler. The scheduling model for the server recognizes three types of scheduled tasks:

- *Immediate Tasks* are scheduled to execute once, usually within 5 seconds, which is the rAPA equivalent to manually launching the task.

rAPA Plugin Structure

- *One-time Scheduled Tasks* are scheduled to execute once, but often include long-running tasks to be scheduled as resources permit.
- *Repeating Tasks* are scheduled to be executed with repeatable schedules, such as hourly, daily, weekly, and monthly.

The built-in functions for rAPA are developed to use the scheduler as appropriate based on how the appliance developer and end user have configured the appliance. Developers writing their own plugins can add their tasks to the scheduler as appropriate as described in the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html].

rAPA Plugin Structure

A *plugin* in rAPA consists of Python code and related files used to accomplish a particular administrative task. Plugin files reside in the directories associated with rAPA or in an alternate location configured for searching plugins. rAPA finds plugins at these configured locations and interprets the code to render the plugin interface and to execute its underlying functions.

Each plugin is referenced in configuration by a plugin identifier. The plugin identifier has the format `/plugin/PluginName` where `plugin` is the subdirectory in which the plugin files reside and `PluginName` is the name of the Python class that defines the plugin. Standard rAPA plugins are installed in `/usr/lib/raa/raaplugins/` by default.

Just as rAPA has two parts, a server and interfaces, the action of a plugin can also be divided into two parts: web and service:

- *Web* -- The web part of a plugin handles the user interaction through the rAPA web interface. rAPA expects to find the web portion of the plugin in the `web` subdirectory of the plugin.
- *Service* -- The service part of a plugin handles scheduled tasks, especially prolonged tasks or tasks requiring root privileges. rAPA expects to find the service portion of the plugin in the `srv` subdirectory of the plugin.

In the Python code of each plugin, developers use methods (functions) associated with a class to pull in different rAPA operations (as defined in the rAPA API). Methods in the web portion can correspond to a URL so that when the browser navigates to that URL, the method is run and its return value is sent to the browser. For example, if the plugin is identified by Python as `project.plugin.web.plugin.Plugin`, then the `index` method is evaluated when the user browses to `/raa/plugin/PluginName/`, and the `example` method is evaluated upon the request of `/raa/plugin/PluginName/example`. When executing a method from the web portion of the plugin, that web portion can request two types of tasks to be executed by the service component:

- *Immediate execution* is for tasks which complete quickly (such as changing the root password), but require root privileges. For such tasks, the call for execution made by the web portion must wait for the service portion to complete the task.
- *Scheduled execution* is for tasks which may run longer than a few seconds and which may or may not require root privileges to execute.

See the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] for further break-down of the rPath Appliance Platform Agent infrastructure and plugin structure.

See Chapter 11, *Standard Plugins in rAPA* for a complete listing of all of the built-in plugins for rAPA with the plugin-specific information you need to reference throughout this guide.

Note

Developers can statically define user roles in the plugin code. These roles allow rAPA administrators to set up user groups in rAPA that can be used to assign one or more roles to the users in those groups. This is useful for appliances that require different administrative roles for configuration and maintenance tasks. For more information on this feature, combine the information from the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] with information about user roles in the section called “Standard Plugin: User Management”.

rAPA Configuration Structure

rAPA has a set of default configuration files and expected locations for overriding custom configuration files. This guide describes creating and packaging custom configuration as part of customizing rAPA for a particular appliance. The following outlines the configuration files in rAPA in the order in which they are loaded:

1. `/usr/lib/python2.4/site-packages/raa/config/app.cfg` and `/usr/lib/python2.4/site-packages/raa/config/log.cfg`
2. One of the following:
 - `/etc/raa/prod.cfg`
 - `/etc/raa/plugins.d/*` (read in ASCII sort order)
 - `/etc/raa/custom.cfg`
3. All of the `.cfg` files in `/etc/raa/plugins.d/` parsed in order returned by the Python `list.sort()` function
4. `/etc/raa/custom.cfg`
5. `custom.cfg` if it exists in the current working directory

Appliance developers can create and package a `/etc/raa/custom.cfg` and any accompanying files to override the default configuration directives in rAPA. Complete your rAPA customization using the information in the remainder of this guide along with the use of the `rapa-custom` template package (introduced in Chapter 3, *Include rAPA in an Appliance*).

Chapter 3. Include rAPA in an Appliance

To include rAPA in an appliance, you must know two things:

- What is the label of the rAPA I need to include in the appliance?
- What is the appropriate way to include rAPA based on how I am writing the appliance group recipe?

Use the information in the following sections to help answer these questions, allowing you to include the default rAPA installation in your appliance. The instructions in this guide for customizing rAPA assume you are first including the default rAPA in this way.

Note

Several labels, packages, groups, and file and directory names related to the rPath Appliance Platform Agent use the "raa" string, which carries over from when rAPA used to be known as the rPath Appliance Agent.

Select Which rAPA to Include

Select the appropriate rAPA label based on which version of rAPA you are using and whether you are using the free version from rBuilder Online or the version that is supported for customers. The label is important in all cases, and the name of the group is important depending on the way the appliance group recipe is written. Though the labels for both versions 2.x and 3.x are provided in this guide, rPath recommends that developers move to using 3.x on appliances, which is more robust and easier to customize:

- For the free rAPA version 3.x as provided from rBuilder Online, use `group-rapa` from `raa.rpath.org@rpath:rapa-3`
- For rAPA version 3.x that is supported for customers, use `group-rapa` from `products.rpath.org@rpath:rapa-3`
- For the free rAPA version 2.x as provided from rBuilder Online, use `group-raa` from `raa.rpath.org@rpath:raa-2`
- For rAPA version 2.x that is supported for customers, use `group-raa` from `products.rpath.org@rpath:raa-2`

Add rAPA to the Appliance Group Recipe

The code requires in the appliance group recipe to add rAPA depends on how the group recipe is constructed. There are three possible options described here, each accompanied by the code required to add rAPA:

- Using `group-appliance` and `addPackages --` When using this combination, add a line to assign a value to `rapaLabel`. Use the appropriate label for the rAPA you want on the appliance:

```
loadSuperClass('group-appliance=rap.rpath.com@rpath:linux-2')
```

Shadow and Use a Template Customization Package

```
class GroupExample(ApplianceGroupRecipe):
    name = 'group-example'
    version = '0.1'
    rapaLabel = 'products.rpath.com@rpath:rapa-3'

    def addPackages(r):
        r.add('rapa-custom')
```

- Using `group-appliance` and `setup --` When using this combination, add `group-raa` or `group-rapa` using an `r.add` line, as in the following example recipe:

```
loadSuperClass('group-appliance=rap.rpath.com@rpath:linux-2')
class GroupExample(ApplianceGroupRecipe):
    name = 'group-example'
    version = '0.1'

    def setup(r):
        r.addAppliancePlatform()
        r.add('group-rapa', 'products.rpath.com@rpath:raa-3')
```

- Using `group-dist --` Though `rPath` recommends using `group-appliance` for all appliances now, some developers may still be maintaining an appliance group recipe that loads the `group-dist` superclass instead. If this is the case, be sure to add `group-raa` or `group-rapa` from the appropriate label, and include that same label in the search path for the recipe to find its build requirements. The `r.add` line in the recipe code is the same as when using `group-appliance` with `setup`.

To learn more about developing the appliance recipe when using the `group-appliance` superclass, see the `rPath` Wiki: http://wiki.rpath.com/wiki/rPath_Linux:group-appliance

Shadow and Use a Template Customization Package

In addition to adding `rAPA` to an appliance, you should add a separate package that includes your customizations for `rAPA`. For your convenience, a template customization package is available from the `app2app` repository at `rBuilder Online`. The recipe and other source files that make up the package are self-documenting, though developers working with `rAPA` customization should use this guide as a more comprehensive reference.

In your appliance development environment, shadow `rapa-custom` from `app2app.rpath.org@rpath:app2app-3-devel` to the development label for your appliance. The command should be similar to the following, replacing your label for the example label:

```
$> cvc shadow example.rpath.org@corp:example-1-devel rapa-custom:source
```

Then, add a line to your appliance group recipe to add `rapa-custom` as shown in the following example recipe:

Use a Testing Environment for Customizations

```
loadSuperClass('group-appliance=rap.rpath.com@rpath:linux-2')
class GroupExample(ApplianceGroupRecipe):
    name = 'group-example'
    version = '0.1'
    rapaLabel = 'products.rpath.com@rpath:rapa-3'

    def addPackages(r):
        r.add('rapa-custom')
```

After rebuilding the appliance group to incorporate the `rapa-custom` package, rAPA should still have its default settings because the template package just incorporates all of the rAPA defaults as placeholders. Back at the root of your current Conary context, check out your shadow of `rapa-custom` to begin your modifications to the files included in the package:

```
$> cvc co rapa-custom
$> cd rapa-custom
```

The remaining sections of this guide indicate which contents of your shadowed `rapa-custom` package can be used to perform the customization work described.

Use a Testing Environment for Customizations

Developers should use a test environment to test rAPA customizations while they are in progress. Ideally, this consists of an appliance image with rAPA plus the customization package (`rapa-custom`). Modifications can be tested and rolled back as necessary throughout the process. However, this should not replace overall appliance testing, and some rAPA customizations may require additions to the quality assurance testing for the appliance.

During customization work, modifications can be made manually and tested prior to packaging, and they can be made in the files in the customization package and brought in with an appliance update. In the case of making manual changes, note that some changes may require restarting the `raa` service before they appear in rAPA. However, in the case of making changes to the package followed by an appliance update, rAPA automatically restarts this service for you.

Chapter 4. Branding the rAPA Interface

The most common customization work for rAPA is to brand the interface with logos and text specific to the appliance and to the company or organization that provides that appliance. The table below shows the locations of the default branding settings along with the expected location for any overriding configuration and files:

Table 4.1. rAPA Branding Locations

Type of Branding	Default Location	Customization Location
Appliance Information Text	product.directives in /etc/raa/prod.cfg	product.directives in /etc/raa/custom.cfg
Location of XHTML for Status Page	status.VendorInformationPath in the [/status] section of /etc/raa/prod.cfg	status.VendorInformationPath in a [/status] section of /etc/raa/custom.cfg
Location of XHTML for Collection Tool	collectiontool.vendorInformationPath in the [/collectiontool] section of FIXME	collectiontool.VendorInformationPath in a [/collectiontool] section of /etc/raa/custom.cfg
Product Logo	/usr/share/conary/web-common/apps/raa/images/prodlogo.png	/usr/share/raa/content/images/prodlogo.png
Other graphics	Graphic files in /usr/share/conary/web-common/apps/raa/images/	Graphic files with the same names as the defaults, but in /usr/share/raa/content/images/
Site-wide CSS	/usr/share/conary/web-common/apps/raa/css/raa.css	/usr/share/raa/content/css/raa.css
Plugin-specific CSS	Various locations, but usually under a plugin-specific directory under /usr/lib/raa/raaplugins/	In a parallel tree structure to that under plugin-specific directory, but under /usr/share/raa/content/plugins/

Reference the following sections for special considerations about each of these branding customizations.

Appliance Information Text

The header and footer of the rAPA interface displays information about the appliance itself, including the title of the appliance, the version, and company information. This is part of the [global] section of the main rAPA configuration file, /etc/raa/prod.cfg. The directives and their default values for rAPA 3.0 appears as follows in that file:

Vendor Text for the Status Page and Collection Tool

```
# Branding
product.companyName="rPath"
product.companyCopyYears="2006-2007"
product.companyURL="http://www.rpath.com/"
product.productName="rPath Appliance Platform Agent"
# Version string to display at the bottom of every page
#product.productVersion="1.0"
```

The product version line is commented out (with "#" at the beginning of the line) because rAPA can alternately use the version value used for the Conary group used to build the appliance. Besides that, the other values are specific to rPath as the company and rPath Appliance Platform Agent as the product.

To change this appliance information, modify the file `custom.cfg`, which is installed by your customization package to `/etc/raa/custom.cfg` on your appliance. If you are using a shadow of the template package `rapa-custom`, open `custom.cfg` in a text editor, locate the branding information lines, and modify them as appropriate for your appliance. The following is an example of what that section might look like in `custom.cfg`:

```
[global]

product.companyName="Example Corporation"
product.companyCopyYears="2006-2008"
product.companyURL="http://www.example.com"
product.productName="Sample Appliance"
product.productVersion="1.1"
```

Vendor Text for the Status Page and Collection Tool

The rAPA status page (shown upon login) and the Collection Tool task each have boilerplate text that should be replaced before releasing rAPA as part of an appliance product. This applies to rAPA version 3.0.0 and later only.

First, the status page has a default configuration in the `[/status]` section in `/etc/raa/prod.cfg`. The directive and its default value for rAPA 3.0 appears as follows in that file:

```
[/status]
# Change this path to point to a static 'snippet' of HTML for
# vendor-specific branding. Change to a blank string to disable.
status.vendorInformationPath = "%(top_level_dir)s/../../raaplugins/statu
```

Second, the Collection Tool has a default configuration in the `[/collectiontool]` section in `/etc/raa/prod.cfg`. The directive and its default value for rAPA 3.0 appears as follows in that file:

Replacement Logos and Other Graphics

```
[/collectiontool]
collectiontool.vendorInformationPath = "%(top_level_dir)s/./raaplugin"
```

To change this boilerplate text to vendor-specific branding, first compose the HTML content that should appear in each of the two locations. The content must reside in a text file and be well-formed XHTML code. Use your customization package to install the file in a consistent location on the filesystem. If you are using a shadow of the template package `rapa-custom`, just modify `vendor-custom-status.html` and `vendor-custom-collectiontool.html` as appropriate.

After composing the HTML for each interface, modify the file `custom.cfg`, which is installed by your customization package to `/etc/raa/custom.cfg` on your appliance. If you are using a shadow of the template package `rapa-custom`, you should not need to edit the `custom.cfg` file unless you change the name or location for the two HTML files (which also requires an update to the corresponding `r.addSource` lines in the package recipe).

Replacement Logos and Other Graphics

rAPA provides an alternate location for files intended to override default graphics and style sheets. If a file exists in the alternate location, rAPA uses that file in place of its default equivalent. If a file does not exist in the alternate location, rAPA uses its default.

The default location for graphics and CSS is `/usr/share/conary/web-common/apps/raa/`.

The alternate location for graphics and CSS is `/usr/share/raa/content/`. Any relative paths under the default location must match exactly with the relative paths under the default location.

One example of providing an alternate graphic is in replacing the product logo. The default product logo is 270x98 pixels and resides in `/usr/share/conary/web-common/apps/raa/images/prod.cfg`. To replace this with a custom logo, first create a PNG file that will fit in the same space and that is also named `prodlogo.png`. Then, install your custom `prodlogo.png` file as `/usr/share/raa/content/images/prodlogo.png` (note the use of the `images` subdirectory to be sure the relative paths match).

If you are using a shadow of the template package `rapa-custom`, just replace the packaged graphic files as appropriate, ensuring that your graphics have the same names and suitable dimensions. Also, if necessary, explore other graphics in the default rAPA location to determine if you should make any additional replacements for your appliance (adding a `r.addSource` line to the package recipe for each replacement graphic).

Custom CSS

As described in the section called “Replacement Logos and Other Graphics”, rAPA provides an alternate location for files intended to override default graphics and style sheets. This guide does not describe the details of the CSS used in rAPA. However, if you are using the shadowed `rapa-custom` package, you can use the packaged `raa.css` file as a starting point to customizing the rAPA CSS. The `raa.css` file is a copy of the default CSS for rAPA (version 3.0 or later).

Custom CSS

Return to this documentation periodically for any suggestions or precautions from rPath about for CSS development for the rPath Appliance Platform Agent. Also, if you are developing custom plugins for rAPA, reference the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] when ensuring that your plugin works with the rAPA CSS.

Chapter 5. Available Tasks and the Configuration Wizard

rPath Appliance Platform Agent configuration provides developers with the ability to control what rAPA tasks are available in the web interface or by XML-RPC, and to modify the order and steps in the initial configuration wizard presented upon an appliance user's first access to the rAPA web interface. The following sections describe how to identify the plugin associated with a task, how to set available tasks, and how to adjust the configuration wizard.

Identify the Plugin Associated with a Task

Each task in the rAPA web interface has an associated plugin, which is the Python code used to power that task. That same plugin provides the XML-RPC-exposed functions also associated with that task. Identify a plugin in rAPA configuration by combining its plugin subdirectory and its Python class name in the plugin code. For the standard rAPA plugin identities, see the *Python Identity* column in quick-reference table at the beginning of Chapter 11, *Standard Plugins in rAPA*. For custom plugins, use the identity determined at the time of plugin development (see *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html]).

Set Available Tasks in rAPA

Tasks in rAPA can be completely enabled or disabled, they can hidden from the web interface without being disabled, and they can be overridden by a customized version of the task. The following sections describe each of these options.

Enable and Disable Tasks

By using the `plugins.disable` directive in the *[global]* section of `/etc/raa/custom.cfg`, developers can designate which rAPA tasks are disabled, even if they are installed with rAPA on the appliance. Each item to be disabled is listed by its Python identity, as described in the section called “Identify the Plugin Associated with a Task”. The following is the default value for `plugins.disable` from `prod.cfg`:

```
plugins.disable = ['/flipflop/Update', 'factoryreset/FactoryReset
```

Note

The standard tasks Flip-flop Update and Factory Reset are disabled by default because they require a special partition structure for the appliance (set up by its developers), and they replace the native update mechanism that uses Conary. Read more about these tasks in Chapter 11, *Standard Plugins in rAPA*.

Note the following when disabling tasks:

Hide Enabled Tasks

- Disabling a task affects both its appearance in the web interface and the availability of its XML-RPC functions. rAPA provides an ability to hide the task from the web interface without disabling it entirely, as explained in the section called “Hide Enabled Tasks”.
- Not only can rAPA standard plugins be part of this list, but any custom plugins included in the appliance can also be added to the list. See the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] for more information about identifying your custom plugin as part of this configuration.

If you are using a shadow of the `rapa-custom` package as part of your appliance development, open the packaged `custom.cfg` in a text editor and modify the `plugins.disable` line as desired. Be sure to keep all listed items on the same line in the square brackets, to separate items by commas, and to enclose each item in single quotes as the defaults are shown. In the following example, the Root Password task has been enabled again, Flip-flop and Factory Reset remain disabled, and the Services task has also been disabled:

```
plugins.disable = ['/flipflop/Update', 'factoryreset/FactoryReset
```

Hide Enabled Tasks

If a task is enabled, such as for use through XML-RPC, developers can select to hide those tasks from the rAPA web interface. This also allows other tasks to access the functions of the enabled task's plugin without exposing its web interface to rAPA users. All tasks that have a web interface for rAPA are displayed by default.

To hide an enabled task from the rAPA web interface, add a directive to the *[global]* section of `/etc/raa/custom.cfg` that uses the following structure:

```
<plugin_dir>.<python_class>.hidden = True
```

The values of **plugin_dir** and **python_class** are the same plugin directory and Python class items that make up the task's plugin identity as described in the section called “Identify the Plugin Associated with a Task”.

If you are using a shadow of the `rapa-custom` package as part of your appliance development, open the packaged `custom.cfg` in a text editor and add `".hidden"` lines as desired. In the following example, the Backup and System Time tasks are hidden from the web interface, though their functions can be called from an XML-RPC client:

```
backup.Backup.hidden = True  
configure.SetTimeZone.hidden = True
```

Override Default Tasks

Appliance developers can override default tasks in rAPA with custom plugins developed specifically for the appliance. Developers accomplish this by creating the custom plugins with

Adjust the Initial Configuration Wizard

a different plugin directory, but the same Python class name so that it will override the default plugin. The `pluginDirsTop` configuration directive in `/etc/raa/raa_service.conf` is used to determine which plugin will win in case of a conflict.

To learn more about developing and incorporating plugins, including those used to override default rAPA tasks, see the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html].

Adjust the Initial Configuration Wizard

The rPath Appliance Platform Agent prompts for a step-by-step initial configuration the first time an administrator logs in to the web interface. Until the wizard steps are completed successfully, rAPA is said to be in *wizard mode*.

As the administrator steps through the wizard, each set of changes is applied when the data is submitted back to the rAPA web server. The wizard will time out, just as any rAPA login session (another value which can be customized by developers). If the wizard times out, it picks up at the last task that was not completed.

By using the `wizard.plugins` directive in the *[global]* section of `/etc/raa/custom.cfg`, developers can set which rAPA tasks are part of the initial configuration wizard, including the order in which they appear. Each item is listed by its Python identity in rAPA, as described in the section called “Identify the Plugin Associated with a Task”. The following is the default value for `wizard.plugins` from `prod.cfg`:

```
wizard.plugins = ['/changepassword/ChangePassword', '/configure/No
```

The order in which the plugins appear in the list is the order they prompt for input when a user steps through the wizard.

Note

Even though the Entitlements task is part of the standard plugins in rAPA, and part of the default initial configuration wizard, the task is not displayed unless developers added an `entitlements.xml` file in the appliance. If you are using a shadow of the `rapa-custom` package in your appliance, use the information in the section called “Standard Plugin: Configuration, Manage Entitlements” to learn how to generate an `entitlements.xml` file, and add that file to the `rapa-custom` package along with an appropriate `r.addSource` line in the package recipe.

If you are using a shadow of the `rapa-custom` package as part of your appliance development, open the packaged `custom.cfg` in a text editor and modify the `wizard.plugins` line as desired. Be sure to keep all listed items on the same line in the square brackets, to separate items by commas, to enclose each item in single quotes as the defaults are shown, and to list them in the order they should occur in the wizard. To determine what Python identity to associate with the desired plugins, see the summary chart at the beginning of Chapter 11, *Standard Plugins in rAPA*. In the following example, the Change Password, Network, and Entitlements tasks are still

Adjust the Initial Configuration Wizard

a part of the wizard, but a custom plugin has been added for further initial configuration of the example appliance:

```
wizard.plugins = ['/changepassword/ChangePassword', '/configure/Ne
```

Warning

rAPA will not exit its wizard mode until all tasks in the sequence are completed successfully. After all tasks are completed successfully, rAPA will not run in wizard mode again. Additionally, individual plugins can be developed to run in their own wizard modes when included in the rAPA initial configuration wizards, and will continue wizard mode until its initial task is completed successfully for more details, see the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html]).

Chapter 6. Custom Help Pages in rAPA

The rPath Appliance Platform Agent provides inline help for its tasks with *Learn More* links on most of the task pages and with a compilation of all the inline help in a user guide accessible by clicking the *Help* link in the rAPA header. Though the help information for the default tasks are written in a generic way designed to apply to most appliances, developers may need to replace or extend this information with custom content.

In its earlier versions, rAPA help customization required checking out the development code, rewriting or replacing certain files, and rebuilding the code. Even though the help was compiled from DocBook XML files, there was no way to modify the help without modifying rAPA itself.

Starting with rAPA 3.0.2, appliance developers can provide a well-formed XHTML document to replace the user guide that is accessed using the *Help* link in the rAPA header. However, the individual *Learn More* links still require changes to the plugin code (see the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html]).

If you are using a shadow of the `rapa-custom` package, modify `userguide.kid` as desired with your custom user guide content. Do not modify the first two lines of that file, though, which are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "ht
```

Then, uncomment the `r.addSource` lines in the *Custom User Guide* example in the package recipe (`rapa-custom.recipe`), and uncomment the `help.userguide` line in `custom.cfg`.

Note

If you are already using a vendor-specific location for custom plugins in `/usr/lib/raa/` (as described in the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html]), you can modify the filename and plugin directory string "vendorcustom" (in the `rapa-custom` recipe and `custom.cfg` files) to match your plugin directory. If you do this, be sure to remove the `r.addSource` line that adds the `__init__.py` file to `/usr/lib/raa/vendorcustom/` because you should have `__init__.py` in your own custom plugin directory, already supporting those custom plugins.

Note

See the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] for information about how to include "Learn More" links in your custom plugins and how to get those compiled in the rAPA user guide.

Chapter 7. Task-specific Modifications for rAPA

Besides branding the web interface and customizing what tasks are available, appliance developers can establish a custom configuration for the default tasks in the rPath Appliance Platform Agent. Chapter 11, *Standard Plugins in rAPA* outlines each default task and its options for custom configuration. Be sure to use the following steps when packaging task-specific modifications for an appliance:

1. Ensure the task is part of the available tasks in rAPA as described in the section called “Set Available Tasks in rAPA”.
2. Identify what custom configuration and supporting files are required for the task from its corresponding section in Chapter 11, *Standard Plugins in rAPA*.
3. Create any of the supporting files needed and package them to install on the appliance. If you are using a shadow of the `rapa-custom` package, add those supporting files to the package checkout, and then add one or more `r.addSource` lines as necessary to the `rapa-custom.recipe` file to install those supporting files in the appropriate locations.
4. Add configuration directives to the rAPA customization file to be installed as `/etc/raa/custom.cfg` on the appliance. If you are using a shadow of the `rapa-custom` package, modify the `custom.cfg` file to add both of the following:
 - The task-specific configuration section from the *Configuration Section* column in the task's list of configuration directives.
 - The configuration directives that require custom configuration, along with those custom values, under the bracketed title of the configuration section.
5. Build and test the rAPA customization package.
6. Build and test the appliance with its custom rAPA configuration.

The following is an example `custom.cfg` that includes some global directives for rAPA in addition to some task-specific configuration sections:

```
[global]
plugins.disable = ['/flipflop/Update', '/factoryreset/FactoryReset']

product.companyName = "Example Corporation"
product.companyCopyYears = "2006-2008"
product.companyURL = "http://www.example.com"
product.productName = "Example Application Appliance"
product.productVersion = "1.0"

wizard.plugins = ['/changepassword/ChangePassword', '/configure/Net

[/backup]
backup.disabled.types = ['NFS', 'LABEL']
```

```
[/logs]
logs.files = {'Example Application Log': '/var/log/example', 'Agent

[/updatetroves]
updatetroves.kernelNumber = 3
updatetroves.rebootAfterUpdate = True
updatetroves.backupBeforeUpdate = True
```

Warning

Each configuration directive should be on its own line without a carriage return at the end. If your text editor is configured to wrap text for better viewing, be sure that the editor does not insert a carriage return automatically, and do not press **Enter** until you have typed the entire configuration directive and value.

Chapter 8. External Authentication for rAPA

The rPath Appliance Platform Agent provides developers with the option to use an external authentication mechanism in place of its built-in authentication for users.

By default in rAPA, when a user logs in to the web interface or passes credentials as part of an XML-RPC call, the appliance calls its *validate_identity* object. If the credentials are valid, the Python code returns an *IdentityObject*; if credentials are not correct, the code returns *None*.

Appliance developers should develop external authentication for rAPA as a Python library which employs *setuptools* for building and deployment. This library may contain a single object, or it may incorporate other custom-developed rAPA plugins as appropriate. The library should include an identity provider object to replace the default *IdentityObject*, and the appliance should have a custom configuration that points to that other object for authentication.

Documentation about *setuptools* is provided at the following site: <http://peak.telecommunity.com/DevCenter/setuptools>

Note

Currently, rAPA only supports username/password authentication mechanisms.

Use the following sections as further reference when creating a custom identify provider object and configuring the appliance to authenticate with that object. To set up and use an rAPA development environment as part of developing an external authentication mechanism for rAPA, see the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html].

Identity Provider Object

The custom identity provider object should either inherit from the built-in class *raa.identity.IdentityProvider*, or it should explicitly defined particular functions it would otherwise inherit. Use the rAPA API documentation for the *IdentityProvider* class as a reference for which functions (methods) must be inherited or defined for the custom external authentication to work in rAPA: <http://cvs.rpath.com/rapa-3.0-docs/raa.identity.IdentityProvider-class.html>

The pam plugin is an example of how an identity provider object can be written for external authentication. The plugin is available as part if the rAPA development environment file *raa/identity/pam/__init__.py*.

rAPA Configuration for Using External Authentication

As with other rAPA tasks, add configuration directives to */etc/raa/custom.cfg* to modify the appliance behavior with regards to authentication. The primary directive to use is *raa.identity.current_provider* in the *[global]* section of *custom.cfg*.

rAPA Configuration for Using External Authentication

The value for *raa.identity.current_provider* is determined by entry points registered with the `setuptools` framework. Developers should leverage the *setup* code to register the custom identity provider class under the *raa.identity.current_provider* entry point used by rAPA.

para>

In the following example, the *pam* class is registered as *pam_identity* in the following entry point code in rAPA's *setup.py*:

```
entry_points=[raa.identity.current_provider]
tools.identity_tool.provider = "raa_identity"
tools.identity_tool.provider_paths = ["raa/identity/builtin", "
```

This default configuration indicates that all the modules under *raa/identity/builtin* and *raa/identity/pam* will be imported until an identity provider class is found (such as **`class.name == "raa_identity"`**).

To specify the PAM identity provider to override this default, use custom configuration as follows:

```
tools.identity_tool.provider = "pam_identity"
```

Then, that name under which the PAM identity provider is registered is the value needed for *raa.identity.current_provider* in *custom.cfg*:

```
raa.identity.current_provider = "pam"
```

Chapter 9. Set Up an XML-RPC Client for rAPA

All the tasks that can be performed in the rPath Appliance Platform Agent web interface can also be performed from client software making XML-RPC calls to the rAPA interface on the appliance. By using the XML-RPC interface for rAPA, appliance management can be incorporated into almost any existing infrastructure management framework, eliminating the need to manage appliances through their separate rAPA web interfaces. For more information about XML-RPC, see its documentation at www.xmlrpc.com [<http://www.xmlrpc.com/>].

The functions within rAPA are exposed to the XML-RPC interface with a `@raa.web.expose` Python decorator that includes `allow_xmlrpc=True` as an argument. Exposed functions are as follows:

- General rAPA operations, driven by the `raa.controllers` module. To reference the list of exposed functions in `raa.controllers`, see the `raa.controllers.Root` class page from the rAPA API: <http://cvs.rpath.com/rapa-3.0-docs/raa.controllers.Root-class.html>
- Individual tasks functions, driven by modules in the standard rAPA plugins. To reference the list of exposed functions for each task, including the expected arguments and a link to view the rAPA API for each, see that task's XML-RPC section in Chapter 11, *Standard Plugins in rAPA*.

Note

When using the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] to develop custom plugins for rAPA, reference how the default rAPA plugins exposed methods using `@raa.web.expose` Python decorators with `allow_xmlrpc=True`. By using this same means to expose the method, its function should be available through the rAPA XML-RPC interface alongside other rAPA tasks.

Use the following sections for further reference on developing an XML-RPC client to call tasks from the rAPA XML-RPC interface.

XML-RPC Python Client

If the XML-RPC client is written in Python, the code should import `xmlrpclib` and create an XML-RPC proxy object using the HTTP URL to the rAPA XML-RPC interface. The URL is as follows, replacing the user credentials as applicable for the appliance, and replacing "localhost" if the XML-RPC client does not reside on the appliance:

```
https://admin:password@localhost:8003/xmlrpc/
```

The following code shows the library import and the creation of the XML-RPC proxy object called `server`:

XML-RPC Java Client

```
import xmlrpclib
server = xmlrpclib.ServerProxy('http://admin:password@localhost')
```

To use HTTPS instead, load *m2xmlrpclib* from the *M2Crypto* module also. The following code shows the same library import and creation of the XML-RPC proxy object *server*, but with this addition and using the HTTPS URL instead:

```
import xmlrpclib
from M2Crypto import m2xmlrpclib, SSL
server = xmlrpclib.ServerProxy('https://admin:password@localhost')
transport = m2xmlrpclib.XML_Transport
```

Note

During testing with HTTPS, ignoring SSL certificate mismatches may be useful. To do so, add this line prior to the *xmlrpclib.ServerProxy* call:

```
SSL.Connection.clientPostConnectionCheck = None
```

The remaining Python code for the XML-RPC client can then call the exposed functions as necessary through its proxy object. For example, the following corresponds to a function provided by the email notifications plugin:

```
new_emails = ('user1@example.com', 'user2@example.com')
server.configure.Notify.notifySave(", ".join(new_emails))
```

XML-RPC Java Client

XML-RPC clients can be written in any language with the appropriate XML-RPC modules. Detailed examples and reference for writing a Java client for XML-RPC are currently published at the rPath Wiki. Use the wiki page as a reference, and look for that information to be published in this document in the future: http://wiki.rpath.com/wiki/rPath_Appliance_Platform_Agent:XML-RPC_Java_Client

XML-RPC for Segmented Updates

Starting with version 2.1.3, rAPA provides a segmented update system through XML-RPC. Segmented updates may reduce testing costs through ensuring a standard upgrade path from one version to the next. For example, updating a product from version 1.0 to 3.0 with segmented updates allows the version update path to start with upgrading version 1.0 to version 2.0, followed by a restart. Then, the version 2.0 to version 3.0 update may continue, also followed by a restart.

Segmented updates can be performed utilizing the following Python script as part of a Python XML-RPC client. Replace "192.168.X.X" with the IP address of the appliance, and

XML-RPC for Segmented Updates

replace "group-example-appliance=1.0.3" and "group-example-appliance=1.0.6" to indicate the appliance group update path for the appliance:

```
#!/usr/bin/python
import xmlrpclib
import time

url = 'https://admin:password@192.168.X.X:8003/xmlrpc-request'
server = xmlrpclib.ServerProxy(url)
updateProxy = server.updatetroves.UpdateTroves
schedId = updateProxy.callSegmentedCheck(['group-example-appliance=1.0.3'])
time.sleep(5)
running = updateProxy.callGetRunStatus()
while running['schedId'] != -1:
    running = updateProxy.callGetRunStatus()
    time.sleep(.1)
updates = updateProxy.callGetAvailableUpdates()
updateId = updates[0]['updateId']
updateProxy.callDownload(updateId)
time.sleep(5)
running = updateProxy.callGetRunStatus()
while running['schedId'] != -1:
    running = updateProxy.callGetRunStatus()
    time.sleep(.1)
updates = updateProxy.callGetAvailableUpdates()
updateId = updates[0]['updateId']
updateProxy.callUpdate(updateId)
time.sleep(3)
running = updateProxy.callGetRunStatus()
while running['schedId'] != -1:
    running = updateProxy.callGetRunStatus()
    time.sleep(.1)
```

Chapter 10. Maintaining Packaged Customizations

Creating, testing, and releasing a customized rPath Appliance Platform Agent interface with an appliance is covered throughout this guide. For ongoing maintenance of packaged customizations, use the following guidelines:

- Version customizations a way that is easy to track the version of rAPA and of the appliance to which they apply.
- When new versions of rAPA are released, read the release notes to determine what changes impact your appliances: http://wiki.rpath.com/wiki/rPath_Appliance_Platform_Agent:Release_Notes
- When selecting to update appliances to a new version of rAPA, modify the customizations as appropriate and test them with the new rAPA version.
- Thoroughly test potential update scenarios for appliances to be sure that rAPA and its customizations will update as desired on deployed appliances.

Chapter 11. Standard Plugins in rAPA

The following table and sections provide a complete reference for the standard plugins powering the default tasks in the rPath Appliance Platform Agent. Use the table as a quick reference, and use the linked task names in the table to jump directly to the section covering the plugin. Tasks are listed in the order they are loaded from `/usr/lib/raa/raaplugins/`, which is alphabetical order of the sub-directory names under that directory:

			to select a previous state to which to roll back the system (prior to one or more of the most recent updates)
Table 11-1. rAPA Default Tasks and Standard Plugins			
Manage Services	services /	Services	Displays system services and prompts for stopping, starting, or restarting any of those services
	the section called “Standard Plugin: Manage Services”		
Updates	updatetroves / *	/updatetroves / UpdateTrove	Displays an available update corresponding to the last check, prompts to check for updates, and provides for scheduling checks for updates and (if desired) automatic update operations; appliance updates with this task go beyond performing an "updateall" Conary operation and could cease to work properly if command-line updates are performed on the appliance
	the section called “Standard Plugin: Updates”		
User Management	usermanagement / *	/ usermanagement / UserInterface	Provides management of rAPA users and role-associated user groups
	the section called “Standard Plugin: User Management”		
Updates with Flip-flop	Maintained separately for customers using group-raa from products.rpath.com		Creates an alternate boot partitions and a toggle between those partitions, one being the system prior to update and the other being the updated system; requires a special partition layout established by appliance developers
	the section called “Standard Plugin: Flip-flop Update”		
Updates with Factory Reset	Maintained separately for customers using group-raa from products.rpath.com		Uses an alternate partition to store the appliance as it is originally installed as a fast restore option; requires a special partition layout established by appliance developers
	the section called “Standard Plugin: Factory Reset”		

Standard Plugin: Appliance Status and System Information

Python identity: `/status/Status`

The status plugin provides the *Appliance Status* page and the *System Information* page in the rAPA web interface and retrievable status information using XML-RPC.

Appliance Status and System Information: Custom Configuration

Use the following table as a guide to customizing the behavior of this plugin or its appearance in the rAPA web interface:

Table 11.2. Status Plugin Configuration

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
<code>[/status]</code>	<code>status.vendorInformationPath</code>	Abstraction path to an HTML file formed by Python variables when rAPA is compiled	Vendor-specific text that is displayed on the <i>Appliance Status</i> page	YES
<code>[global]</code>	<code>plugins.statusList</code>	<code>['/update/UpdateTrove', '/backup/Backup', '/configure/Notify']</code>	Summary information displayed on the <i>Appliance Status</i> page, provided the plugin has implemented a <code>getStatus</code> method in its code	no

See the section called “Vendor Text for the Status Page and Collection Tool” for more information about customizing the vendor text associated with the `status.vendorInformationPath` configuration. Additionally, see the section called “Replacement Logos and Other Graphics” for instructions on replacing graphics, including the logo displayed on the *Status* page in the web interface.

Appliance Status and System Information: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.status.web.status.Status-class.html>

- `index(self)`
- `sysinfo(self)`

Standard Plugin: Back Up and Restore

Python identity: `/backup/Backup`

The backup plugin provides the *Back Up and Restore* page in the rAPA web interface and backup and restore tasks through XML-RPC.

Back Up and Restore: Sequences

rAPA has a particular sequence in which it performs its backup and restore operations. Backup operations use the following sequence:

1. Run each script in `/etc/raa/backup.d/` with `clean`
2. Assemble a list of files to back up
3. Run each script in `/etc/raa/backup.d/` with `prebackup`
4. Run each script in `/etc/raa/backup.d/` with `metadata`
5. Run each script in `/etc/raa/backup.d/` with `backup`
6. Mount the backup destination
7. Create the tar archive on the destination
8. Validate the tar archive
9. Unmount the backup destination
10. Run each script in `/etc/raa/backup.d/` with `postbackup`

Restore operations use the following sequence:

1. Run each script in `/etc/raa/backup.d/` with `clean`
2. Mount the location where backups are stored
3. Extract the backup metadata from the tar archive
4. Run each script in `/etc/raa/backup.d/` with `isValid`, passing the metadata as a standard input stream (`stdin`)
5. Run each script in `/etc/raa/backup.d/` with `prerestore`
6. If `backup.validateSha1` is set, validate the SHA-1 digest of the backup archive
7. Extract the backup files
8. Run each script in `/etc/raa/backup.d/` with `restore`
9. Run each script in `/etc/raa/backup.d/` with `postrestore`
10. Reboot the system

Note

For instructions and reference for writing the scripts that are run in these sequences, see the section called “Back Up and Restore: Sequences”.

Back Up and Restore: Backup Targets

The backup types in rAPA are identified by target as listed in the following table:

Table 11.3. Backup Targets

Name	rAPA Identity	Enabled by default?	Description
Downloadable Backup File (HTTP)	URL	no	Single tar archive saved on the appliance and available to the appliance user for download over HTTP
Network File System Share (NFS)	NFS	<i>YES*</i>	Network-accessible file share using the NFS protocol
	* Additional requirements for NFS on the appliance: <code>portmap</code> and <code>nfs-utils</code> packages		
"Windows" File System Share (CIFS/SMB)	CIFS	<i>YES**</i>	Network-accessible file share using either the SMB protocol or its extended CIFS protocol
	** Additional requirements for CIFS/SMB on the appliance: <code>smb:runtime</code> component		
Mounted Filesystem (LABEL)	LABEL	<i>YES</i>	Linux volume, typically on another physical device, that is mounted to the appliance's filesystem by its volume label

See the default inline help in the rAPA web interface for more information on how to configure the appliance to perform backups for each target shown here. Be sure to keep sufficient information available to users, even if the help is customized for the appliance.

See the section called “Back Up and Restore: Custom Configuration” for how to change which backup targets are disabled.

Back Up and Restore: Data to be Backed Up

Create file lists defining critical files on the appliance that should be included in each backup, and install those file lists in a configured location that rAPA will check on each backup operation. A file list consists of the following:

- A text file without its executable bits set
- A list of files and directories, one per line, with the absolute path to each as they should exist on the appliance

Back Up and Restore: Backup and Restore Scripts

If an item on the list is a directory, rAPA automatically backs up everything in that directory. If an item on the list is a file, rAPA will remember the filesystem location, including all the directories from "/" to that file in the absolute path, but it will not back up any files not otherwise specified in those directories. For example, if you back up `/etc/example/config`, the backup will contain directories `/etc` and `/etc/example`, but those directories are empty except for what is specified in the file list.

By default, rAPA will recognize the following file lists. When creating custom file lists, be sure to include whatever is appropriate from these defaults:

- `/etc/raa/backup.d/rAAFiles`
- `/etc/raa/backup.d/SuggestedFiles`

When applicable, simplify file list entries by using shell-recognized wildcards, such as in the following example. See general bash shell documentation for Linux as a reference for such wildcards:

```
/etc/appconfig
/etc/httpd/conf.d/{vlan-*,app}.conf
```

Note

File lists add flat files to the rAPA backup for the appliance. To run scripts that can help to preserve and restore other data, such as database contents, see the section called “Back Up and Restore: Backup and Restore Scripts”.

If you are using a shadowed version of the `rapa-custom` package, use the file `rapa-backup-locations` in that package as your file list or as a reference for creating additional file lists. Then, modify the "Custom Backup Configuration" part of `rapa-custom.recipe` to be sure your file lists are installed in the default location which rAPA checks on each backup.

The location that rAPA checks for file lists is set using the `backup.file_list_dir` configuration directive. Be sure that if this location is changed in custom configuration, the location to which the file lists are installed is also changed to match.

Back Up and Restore: Backup and Restore Scripts

Create backup scripts to run backup-related tasks before or after backup and restore operations. Use scripts when file lists are not sufficient, such as when dumping data from a database to be included in the backup. If a script is used to create additional files to back up, though, be sure to include those files in a file list as described in the section called “Back Up and Restore: Data to be Backed Up”.

Scripts should reside with file lists in the same directory: `/etc/raa/backup.d/`. rAPA distinguishes between files because scripts are executable (with executable bits set) and file lists are non-executable. Be sure to make each script executable using `chmod` (a standard Linux command for setting file permissions and mode).

Back Up and Restore: Custom Configuration

All scripts in `/etc/raa/backup.d/` need to either accept a single argument, or to require no arguments.

The following functions are defined in the example script `/etc/raa/backup.d/rAADatabase` from the default rAPA install, and they can be reused and adapted for the needs of your appliance:

Table 11.4. rAADatabase Example Backup Script Functions

Function	What it instructs the script to do
<i>backup</i>	Perform backup functions and print a list to standard out (stdout) of files or directories to back up
<i>restore</i>	Perform restore functions (script must be configured to handle appropriate files, such as responding to files in a temporary directory by cleaning it up when the script receives the <code>clean</code> argument described here)
<i>clean</i>	Clean any temporary data generated and used during backup or restore operations
<i>prebackup</i>	Perform any necessary functions before a backup starts
<i>postbackup</i>	Perform any necessary functions after a backup finishes
<i>prerestore</i>	Perform any necessary functions before a restore starts
<i>postrestore</i>	Perform any necessary functions after a restore finishes
<i>metadata</i>	Output metadata (in "key=value" form, one key per line) to be stored in the backup's metadata section
<i>isValid</i>	Read the metadata from standard in (stdin) and validate whether a backup can be restored to a particular appliance

If you are using a shadow of `rapa-custom`, you can also modify `rapa-dbbbackup` in the package as a script for your appliance. Then, modify the "Custom Backup Configuration" part of `rapa-custom.recipe` to be sure your scripts are installed in the default location which rAPA checks on each backup and restore operation.

Back Up and Restore: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.5. Backup Plugin Configuration

			backup type	
<i>[/backup]</i>	backup.file_list_dir	etc/raa/backup.d'	Directory location on the appliance through which to iterate for reading file lists and running scripts	no
<i>[/backup]</i>	backup.file_format	backup-%Y%m%d-%H%M-%Z.tar'	Syntax of the filename for the backup file (see the manual page for <code>strftime</code> for available substitutions)	YES
<i>[/backup]</i>	backup.filter_expression	.*\.(tar tgz)'	Regular expression for filtering which available backup files to keep on the system (using the prune method)	no
<i>[/backup]</i>	backup.validateSha1	False	Toggle for whether to verify a backup's SHA-1 digest after a backup or before a restore (takes lots of time for large backups over a network share)	no
<i>[/backup/files]</i>	tools.staticdir.on	True	Toggle for whether to keep a local copy for use by the 'URL' backup type	no
<i>[/backup/files]</i>	tools.staticdir.dir	"/var/lib/raa/backups"	Directory location on the appliance where, if <i>tools.staticdir.on</i> is True, the backups are stored locally for use by the 'URL' backup type; value MUST match the value used in <i>backups.local_storage</i>	no

Note

The following directives are part of this task, are not supported for modification. Their default values are essential for proper function of rAPA: `server.max_request_body_size`

Back Up and Restore: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.backup.web.backup.Backup-class.html>

- `schedScan(self)`
- `getBackupList(self)`
- `index(self)`
- `saveConfig(self, enableBackups, enableBackupSchedule, checkFreq, timeHour, timeDay, timeDayMonth, numBackups, locationType, locationHost, locationPath, locationUsername, locationPassword, locationLabel, fileformat="", filterexpression="")`
- `config(self)`
- `download(self, backupId=None)`
- `upload(self, backupfile=None)`
- `backupNow(self)`
- `restore(self, backupId)`
- `getJobData(self, schedId, execId)`
- `getSettings(self)`
- `getBackupProperties(self, backupId)`
- `setAvailableBackups(self, files)`
- `saveBackupProperties(self, backupId, start, end, size, tarfilename, avail, properties, sha1)`
- `saveBackup(self, schedId, execId, start, end, size, tarfilename, properties)`

Standard Plugin: Change Password

Python identity: `/changepassword/ChangePassword`

The change password plugin provides the *Change Password* page in the rAPA web interface and password change tasks through XML-RPC. This plugin deals only with changing the password for a rAPA user, not for a user on the appliance's underlying Linux operating system.

Change Password: Custom Configuration

This task does not have any task-specific customization options. However, it is included by default in the initial configuration wizard (see the section called "Adjust the Initial Configuration Wizard"), and the initial username and password of the first user (also an administrator) for rAPA have default credentials as configured with the following global directives:

Table 11.6. Default Initial rAPA User

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
[global]	defaults.initialAdminUsername	admin	The user name of the initial rAPA user; required for web interface login and executing any XML-RPC operations if no other rAPA users are configured	no
[global]	defaults.initialAdminFullName		The full name of the initial rAPA user	no
[global]	defaults.initialAdminPassword		The password of the initial rAPA user; required for login on first entry and until the <i>Change Password</i> task is run for that user	no

See the section called “Standard Plugin: User Management” as a reference for creating and managing additional rAPA users.

Change Password: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.changepassword.web.changepassword.ChangePassword-class.html>

- index(self)
- changePassword(self, oldpwd=", pwd1=", pwd2=)

Note

Because the XML-RPC client will call a method by passing the necessary user credentials, changing the password automatically applies to the user whose credentials are passed. See the User Management plugin (the section called “Standard Plugin: User Management”) for managing users other than the one whose credentials are used to make the XML-RPC call.

Standard Plugin: Collection Tool

Python identity: `/collectiontool/CollectionTool`

The collection tool plugin provides the *Collection Tool* page in the rAPA web interface and the ability to collect and download the same data using XML-RPC.

Collection Tool: Data to be Collected

The collection tool plugin uses file lists and scripts similar to the those used in the backup plugin. Create scripts to run before the files are collected to place any data from binary entities such as a database into a flat file on the filesystem. Then, create file lists to gather all the flat files that should be included in the data collection. Finally, install those scripts and file lists in a configured location that rAPA will check on each backup operation. By default, rAPA will check `/etc/raa/collect.d/` for these scripts and lists.

To learn more about how to structure file lists and scripts for use by rAPA, see information about those used in the backup plugin: the section called “Back Up and Restore: Data to be Backed Up” and the section called “Back Up and Restore: Backup and Restore Scripts”.

If you are using a shadowed version of the `rapa-custom` package, add any scripts and file lists to the package as necessary to customize the data that is collected, and modify the file `vendor-custom-collectiontool.html` to customize the text that is displayed in the web interface.

Collection Tool: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.7. Collection Tool Configuration

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
<code>[/collectiontool]</code>	<code>collectiontool.vendor</code>	<code>%(InformationPath)raaplugins/collectiontool/web/static/vendor.html</code>	Location of the HTML fragment which provides the instructional text on the <i>Collection Tool</i> page in rAPA, generally be used to describe how a customer should collect and submit this data as part of appliance support	YES*
	* See the section called “Vendor Text for the Status Page and Collection Tool” for more information about customizing the vendor text associated with this configuration.			
<code>[/collectiontool]</code>	<code>collectiontool.storageDir</code>	<code>%(raa)raa/collections</code>	Default storage location for data collections	no
<code>[/collectiontool]</code>	<code>collectiontool.configDir</code>	<code>%(raa)raa/collect.d</code>	Directory from which configuration files are read; there is typically no need to change this value	no
<code>[/collectiontool]</code>	<code>collectiontool.fileNameFormat</code>	<code>%(format)Y-%m-%d.%H-%M</code>	Syntax of the filename for the data collection file (see the manual page for <code>strftime</code> for available substitutions)	no

Collection Tool: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.collectiontool.web.collectiontool.CollectionTool-class.html>

- `index(self)`
- `collectNow(self)`
- `getJobData(self, schedId, execId)`
- `downloadNow(self)`

Standard Plugin: Configuration

Even though `configure` is a single directory under the rAPA plugins, it provides several configuration tasks. The *Configuration* sections that follow provide reference for each separate task.

Note

Even though configuration is contained within a single directory, the individual Python classes for each task allow them to be enabled and disabled individually as described in the section called “Set Available Tasks in rAPA”.

Standard Plugin: Configuration, Configure Networking

Python identity: `/configure/Network`

The network part of the configuration plugin provides the *Configure Network* page in the rAPA web interface and the ability to modify network and hostname configuration using XML-RPC calls.

Configure Networking: Hostname Change Scripts

Because a hostname change on an appliance can require additional change on the appliance, rAPA can be configured to run scripts before and after a hostname change. To make use of this feature, create scripts (including making the script files executable), include them to be installed on the appliance, and use the `configure.network.prescript` and `configure.network.postscript` lines in `/etc/raa/custom.cfg` to indicate the script locations. See the section called “Configure Networking: Custom Configuration” for more details about each directive, including its appropriate section.

If you are using a shadow of `rapa-custom`, add the script files to the package, take the following steps to add hostname change scripts:

1. Create and test the hostname change scripts.
2. Add `r.addSource` lines to the `setup` method in `rapa-custom.recipe` to install those scripts to a particular location on the appliance.
3. Modify `custom.cfg` in the package to have a `[/configure/Network]` section with both `configure.network.prescript` and `configure.network.postscript`, each set equal to the appropriate absolute path of the installed script (use single quotes around the path).

Configure Networking: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.8. Configure Networking Configuration

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
<i>[/configure/Network]</i>	configure.network	prescript	Script to be run before changing the system's hostname	no
<i>[/configure/Network]</i>	configure.network	postscript	Script to be run after changing the system's hostname	no
<i>[/configure/Network]</i>	configure.network	ifExclude "tun", "vmnet", "ird"]	A list of interfaces that, even if they exist on the appliance, should not be displayed in the rAPA web interface	no

Configure Networking: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.configure.web.network.Network-class.html>

- `index(self)`
- `saveGeneral(self, usesdhcp, hostName, dnsDomain, dnsServer, gateway, dnsdhcp="0")`
- `saveAll(self, **kwargs)`
- `saveInterface(self, device, dhcp, ip, netmask, gateway, applyNow=False)`
- `restartInterface(self, device)`
- `wizDone(self)`
- `getConfigData(self)`

Standard Plugin: Configure Notification

Python identity: `/configure/Notify`

The notify part of the configuration plugin provides the *Configure Notification* page in the rAPA web interface and the ability manage email notifications and SMTP settings using XML-RPC calls.

Configure Notification: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.9. Configure Notification Configuration

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
[global]	configure.Smtplib	hidden	Hide the SMTP configuration part of the notification task (which was in its own separate task prior to rAPA 3.x)	no

Configure Notification: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.configure.web.notify.Notify-class.html>

- index(self)
- notifySave(self, emails)
- mailUpdate(self, mailRelay=None, mailFrom=None)
- save(self, mailRelay=None, mailFrom=None, emails=[])
- getStatus(self)

Standard Plugin: Configuration, Configure Proxy

Python identity: `/configure/Proxy`

The proxy part of the configuration plugin provides the *Configure Internet Proxy* page in the rAPA web interface and the ability to configure HTTP and HTTPS proxies using XML-RPC calls.

Configure Proxy: Custom Configuration

There are no custom configuration options for the *Configure Internet Proxy* task except to enable or disable the plugin as described in Chapter 5, *Available Tasks and the Configuration Wizard*.

Configure Proxy: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.configure.web.proxy.Proxy-class.html>

- getConfs(self)
- index(self)

Standard Plugin: Configuration, System Time

- `saveProxy(self, httpProxy="", httpPort=0, httpUser="", httpPass="", httpsProxy="", httpsPort=0, httpsUser="", httpsPass="")`

Standard Plugin: Configuration, System Time

Python identity: `/configure/SetTimeZone`

The system time part of the configuration plugin provides the *System Time* page in the rAPA web interface and the ability to configure the time zone and system time using XML-RPC calls.

Note

If the appliance is a virtual appliance running on a private hypervisor or a cloud computing environment, the system time is managed by the hardware on which the virtual machine is running. In these cases, rAPA displays a message indicating this, but still allowing the rAPA user to modify the time zone.

System Time: Custom Configuration

There are no custom configuration options for the *System Time* task except to enable or disable the plugin as described in Chapter 5, *Available Tasks and the Configuration Wizard*.

System Time: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.configure.web.settimezone.SetTimeZone-class.html>

- `index(self)`
- `skip(self)`
- `save(self, zoneList="", daysList="", monthsList="", yearsList="", hoursList="", minutesList="", ntpserver="", usesNtp=False, zoneOnly=False)`
- `getProperties(self)`
- `getTimeZoneList(self)`

Standard Plugin: Configuration, Manage Entitlements

Python identity: `/configure/Entitlements`

The entitlements part of the configuration plugin provides the *Manage Entitlements* page in the rAPA web interface and the ability to upload an entitlement key to an appliance using XML-RPC.

Manage Entitlements: Include entitlements.xml

Even though this task is included and enabled by default in rAPA, appliance developers must have a valid entitlements file (`entitlements.xml`) on the appliance in order for this task to

Manage Entitlements: Custom Configuration

be active. The contents of this file and the updates to which it entitles an appliance are controlled as part of the rPath Appliance Platform Entitlement Service (rES). See rES documentation to learn how to generate the `entitlements.xml` file for the appliance.

If you are using a shadow of the `rapa-custom` package for customizations, you can add an `entitlements.xml` file to that package to install along with your other customizations. Then, add the following `r.addSource` to the `setup` method in `rapa-custom.recipe` to install the file in the location which rAPA checks by default for this file:

```
r.addSource('entitlements.xml',
            dest='/etc/conary/entitlements.xml')
```

Manage Entitlements: Custom Configuration

There are no custom configuration options for the *Manage Entitlements* task except to enable or disable the plugin as described in Chapter 5, *Available Tasks and the Configuration Wizard*.

Entitlements: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.configure.web.entitlements.Entitlements-class.html>

- `index(self)`
- `doSaveKey(self, key)`
- `saveKey(self, key)`
- `getEntitlementMap(self)`

Standard Plugin: Configuration, Upload SSL Certificate

Python identity: `/configure/SSLCert`

The SSL certificate part of the configuration plugin provides the *Upload SSL Certificate* page in the rAPA web interface and the ability to upload a new or replacement SSL certificate using XML-RPC calls.

Upload SSL Certificate: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.10. Upload SSL Certificate Configuration

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
[global]	server.ssl.socket	8003	The port on which to run the secure HTTP connection to rAPA using SSL	no
[global]	server.ssl.certificate	/etc/ssl/pem/raa.pem"	The location of the SSL certificate that is uploaded by the <i>Upload SSL Certificate</i> task	no
[global]	server.ssl.private	/etc/ssl/pem/raa.pem	The location of the SSL certificate that is uploaded by the <i>Upload SSL Certificate</i> task	no

Upload SSL Certificate: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.configure.web.sslcert.SSLCert-class.html>

- index(self)
- upload(self, sslCert=None)
- regenerate(self, confirm=False)

Standard Plugin: Appliance Logs

Python identity: /logs/Logs

The logs plugin provides the *Appliance Logs* page and log file lines or entire log file downloads using XML-RPC.

Using configuration options, appliance developers can customize the number of lines displayed in the web interface for a log, and the logs that are available for download from the web interface or using an XML-RPC call. Files that are added to the comma-separated *logs.files* list should be indicated by absolute path.

Compressed files and null (None) entries may be included for download in the *logs.files* list. Compressed files can be included for download, but the plugin does not generate the compressed

Appliance Logs: Custom Configuration

log files themselves. The plugin supports the download of compressed files of the following mime types:

- tar.gz
- tgz
- Z
- bz2
- zip
- gz

Null entries (or entries using None instead of a file path) will have the name of the log in the drop-down list in the web interface, but they do not contain any log file or compressed archive.

If you are using a shadow of `rapa-custom`, modify the `logs.num_lines` and `logs.files` lines as desired for your appliance, and use the comments in the file as a syntax guide for the file list.

Appliance Logs: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.11. Appliance Logs Configuration

Configuration Section	Directive	Default Value	Description	Already in <code>rapa-custom</code> ?
<code>[/logs]</code>	<code>logs.num_lines</code>	100	The number of lines displayed from the selected log file, always from the end of the file	no
<code>[/logs]</code>	<code>logs.files</code>	{'Appliance Logs': None, 'Conary': '/var/log/conary', 'Agent Service': '/var/log/raa/raa-service.log', 'Agent Web Service': '/var/log/raa/web', 'System': '/var/log/messages'}	The log selection list	no

Note

In earlier versions of rAPA, `logs.display_lines` was used to set the options for users when selecting the number of lines to display. Starting with rAPA 3.0.0, this feature is removed and appliance developers select a single value for the number of lines to display.

Appliance Logs: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.logs.web.logs.Logs-class.html>

- `index(self)`
- `downloadLogs(self, logName)`
- `getLog(self, logName, lines)`

Standard Plugin: Schedule Reboot

Python identity: `/reboot/Reboot`

The reboot plugin provides the *Schedule Reboot* page and the ability to reboot, shut down, or schedule a reboot for the appliance using the web interface or XML-RPC calls.

Schedule Reboot: Custom Configuration

There are no custom configuration options for the *Schedule Reboot* task except to enable or disable the plugin as described in Chapter 5, *Available Tasks and the Configuration Wizard*.

Schedule Reboot: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.reboot.web.reboot.Reboot-class.html>

- `rebootNow(self)`
- `shutdownNow(self)`
- `index(self)`
- `saveSchedule(self, year, month, date, hour, minute, schedId=None)`
- `unscheduleReboot(self, schedId=None)`
- `getTaskType(self, schedId)`

Standard Plugin: Rollbacks

Python identity: `/rollbacks/Rollbacks`

The rollbacks plugin provides the *Rollbacks* page and the roll back all the changes made to the system during one or more of the most recent appliance updates, using either the web interface or XML-RPC calls.

Rollbacks: Custom Configuration

There are no custom configuration options for the *Rollbacks* task, but some of the *Updates* task configuration requires that the rollbacks plugin is enabled (see the section called "Standard

Rollbacks: XML-RPC

Plugin: Updates”). Also, note that the rollback mechanism is powered by the appliance's underlying Canary filesystem management, and a custom Canary configuration can be packaged for the appliance when necessary.

Rollbacks: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.rollbacks.web.rollbacks.Rollback-class.html>

- `index(self, start=0, length=3)`
- `callRollback(self, rbName)`
- `getRollbackName(self, schedId)`
- `finishRollback(self, rbName)`

Standard Plugin: Manage Services

Python identity: `/services/Services`

The services plugin provides the *Services* page and the ability to stop, start, or restart services running on the underlying operating system of the appliance, using either the web interface or XML-RPC calls.

All services displayed when running `chkconfig --list` on the appliance can be managed in the *Services* web interface except for those hidden by using `services.hidden`. Default configuration for the services plugin, including the default `services.hidden` value, is in `/etc/raa/plugins.d/services.cfg`.

If you are using a shadow of `rapa-custom`, modify `custom.cfg` to add a `[/services/Services]` section with the desired custom configuration for the services plugin.

Manage Services: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Table 11.12. Manage Services Configuration

Configuration Section	Directive	Default Value	Description	Already in rapa-custom?
<code>[/services/Services]</code>	services.hidden	<code>['raa','raa-restore','aaa_raa-shutdown','Makefile]</code>	The services that may be running on the appliance that should not be managed in the web interface	no
<code>[/services/Services]</code>	services.introText	"Services to start and stop."	The introductory text displayed in the <i>Manage Services</i> web interface	no
<code>[/services/Services]</code>	services.description	<code>['http': 'Apache Web Server', 'mysql': 'MySQL Database', 'crond': 'Cron Daemon', 'sshd', 'Secure Shell']</code>	Friendly descriptions of services to replace displaying the names of the services in the web interface (such as "Apache Web Server" to be displayed in place of "httpd")	no

Manage Services: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.services.web.services.Services-class.html>

- `index(self)`
- `callChangeState(self, srv, act)`
- `getJobProp(self, schedId)`

Standard Plugin: Updates

Python identity: `/updatetroves/UpdateTrove`s

The `updatetroves` plugin provides the *Updates* page and the ability to check for and perform updates and manage the update schedule using either the web interface or XML-RPC calls.

Updates: Configure the Number of Kernels to Keep

Warning

Appliance updates using rAPA are specially designed to operate on the top-level group that defines the appliance, doing more than a typical command-line update (such as with `conary update` or `conary updateall`). Because of this, any updates on the appliance done using the command line instead of rAPA could cause future rAPA updates to fail or to corrupt the appliance. Remember that in the appliance model, interfaces such as rAPA should completely replace all command line system management, including updates.

Updates: Configure the Number of Kernels to Keep

Appliance developers can configure the number of Linux kernels the underlying operating system will keep after an update. From the web or XML-RPC interfaces, though, users cannot see this configuration or override it. Though the default configuration is usually sufficient, developers are encouraged to recognize the importance of keeping "last known good" kernels in case of boot failures after an update.

Use the `updatetroves.kernelNumber` configuration directive to override the default value of "2" set in `/etc/raa/plugins.d/updatetroves.cfg`. Select a value as follows:

- Use **0** (zero) to allow Conary's configuration to determine what actions should be taken. By default, Conary will keep all kernels "pinned" so that no older kernels are removed after updates to the `kernel` package.
- Using **1** (one) will give the same results as **2**, preventing the removal of a "last known good" kernel to which the appliance can boot.
- Use **2** or greater to keep that number of kernels pinned after update to the `kernel` package. This is inclusive of the currently running kernel, so the default "2" implies that the appliance can update to the newest kernel and keep just one other "backup" kernel for a total of two.

Updates: Show or Hide the Migrate Tab

A migrate is a special type of appliance update that replaces all the software from the top-level group down. A migrate also removes some data on the system that is not managed as part of that group, possibly including some application data. Because a migrate action is usually a form of system repair or performed in the rare case of a major appliance restructuring, the *Migrate* tab in the web interface is hidden, even though it can be called with a particular URL.

Enable the *Migrate* tab in the web interface by setting `updatetroves.migrate.hide` to **True** in the `[/updatetroves]` section of your configuration. Otherwise, to call the migrate operation in the web interface even when the tab is hidden, access the following URL on the appliance (replacing "hostname" as appropriate):

```
https://hostname:8003/updatetroves/UpdateTrovEs/migrate
```

Updates: Display Extended Information

Appliance developers can provide users with extended information about an available update, including the differences between current and pending software versions. To do this, developers

Updates: Display Extended Information

must use the *updatetroves.rootUrl* directive that displays a hyperlink beside the *System Update* text when updates are available for the appliance. When the user clicks to view the extended information, the web interface displays additional update details provided by the rPath Appliance Platform Update Service appliance.

The Update Service connects the appliance to the URL specified by the *updateTroves.rootUrl* configuration directive. When it accesses that service using HTTP GET requests, the URL is modified by appending a question mark (?) and a set of variables which correspond to naming and version information for the software comprising your appliance. The following table shows the list of variables that can be passed:

Table 11.13. Update Service Query Variables

Variable	Name	Description	Example
og	Old Group	The current top-level group	group-example
ov	Old Version	The current version of the current top-level group	1.0.5-0.3-2
ng	New Group	The desired top-level group	group-example
nv	New Version	The desired version of the desired top-level group	1.0.6-0.1-1

The following is an example query to the Update Service showing the modified URL:

```
http://updates.example.com?og="group-dist";ov="1.0.5-0.3-2";ng="group-example";nv="1.0.6-0.1-1"
```

The variables used in the platform's HTTP GET query correspond to data which is stored in a database for use by the rPath Appliance Platform between the appliance and an Update Service appliance. The following table is an example table structure:

Table 11.14. Example Structure for the Update Service Database

Column	Type
GroupId	INTEGER PRIMARY KEY
GroupName	VARCHAR(100)
GroupVersion	VARCHAR(100)
Details	TEXT

The *Details* field should contain a list of all changes from the previous *GroupId*. Thus, you can join all of the *Details* for all of the *GroupId* values between the old group name and version and the new group name and version.

Updates: Offline Updates Using External Media

To display the additional update information from this query, use the following two configuration directives as listed in the section called “Updates: Custom Configuration”:

- *updateToves.migrate* -- Set this to **True**; this corresponds to your appliance migrating software to a new version rather than updating it
- *updateToves.rootUrl* -- Set this to your custom Update Service URL and specify the update details

Updates: Offline Updates Using External Media

Appliance vendors can offer appliance updates using external media instead of over a network connection from an Update Service appliance. This is useful for appliances that are deployed over an unreliable Internet connection or are completely disconnected from the Internet or other network-accessible Update Service resource.

rAPA can detect and can connect devices, such as USB drives, to determine if they contain an available update for the appliance. Even though the device will have all the components needed to bring any appliance up-to-date to a specific version, it is not a means of reinstalling the appliance in its entirety.

See the *rBuilder User Guide* [http://docs.rpath.com/rBuilder_User_Guide/index.html] for more information on creating an offline update that can be used to update deployed appliances.

Updates: Custom Configuration

Use the following table as a guide to customizing the behavior of this task or its appearance in the rAPA web interface:

Updates: Custom Configuration

			Update Service appliance)	
		updatekernelNumber	Set the number of kernels to keep as "last known good" when updates occur (see the section called "Updates: Configure the Number of Kernels to Keep")	no
Table 11.15. Updates Configuration				
<i>[/updateoves]</i>	updateoves.freezePath	<i>/lib/raa/frozen</i>	The location on the appliance's filesystem where update jobs are frozen	no
<i>[/updateoves]</i>	updateoves.downloadPath	<i>/lib/raa/download</i>	The location on the appliance's filesystem where update jobs are downloaded if available	no
<i>[/updateoves]</i>	updateoves.mountPath	<i>/lib/raa/mounts</i>	The location on the appliance's filesystem where external devices are mounted for the purpose of providing "offline" updates (see the section called "Updates: Offline Updates Using External Media")	no
<i>[/updateoves]</i>	updateoves.rebootAfterUpdate		Reboot the machine after every update	no
<i>[/updateoves]</i>	updateoves.backupBeforeUpdate		Perform a backup before each update operation, and perform a restore after each rollback; requires the rollback plugin to be enabled, also (see the section called "Standard Plugin: Rollbacks").	no

Updates: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.updatetroves.web.updatetroves.UpdateTrove-class.html>

- `getStatusByUpdateId(self, updateId)`
- `index(self, schedId=None)`
- `getMountedMedia(self)`
- `callSegmentedCheck(self, updateSegmentList=None, repoPath=None)`
- `callCheck(self, troveSpec=None, method=None, repoPath=None)`
- `callDownload(self, updateId)`
- `callUpdate(self, updateId)`
- `revertJournal(self)`
- `callGetRunStatus(self)`
- `prefs(self)`
- `prefsSave(self, checkFreq, timeHour, timeDay, timeDayMonth, doCheck="off", doAuto="off")`
- `getAvailableUpdatesRss(self)`
- `callGetAvailableUpdates(self)`
- `saveSchedule(self, updateId, year, month, date, hour, minute)`
- `unscheduleUpdate(self, updateId)`
- `callCancelUpdate(self)`
- `mediaSearch(self)`
- `mediaUnmountSingle(self, mntpoint)`
- `mediaUnmountAll(self)`
- `mediaUnmount(self)`
- `getTaskDetails(self, schedId)`
- `reportDone(self, updateId, stage, ret)`
- `clearAllButCurrentUpdate(self, updateId)`
- `migrate(self)`
- `doMigrate(self, troveName, troveVersionFlavor)`

Note

Starting with version 2.1.3, rAPA provides a segmented update system through XML-RPC. Segmented updates may reduce testing costs through ensuring a standard upgrade path from one version to the next. To learn more about this feature, see its dedicated section, the section called "XML-RPC for Segmented Updates".

Standard Plugin: User Management

Python identity: `/usermanagement/UserInterface`

The usermanagement plugin provides the *User Management* page and the ability to add, modify, and delete users and user groups using either the web interface or XML-RPC calls.

Note

rAPA can be modified to use external authentication of users instead of its native user management. To use this feature, refer to the instructions in Chapter 8, *External Authentication for rAPA*.

User Management:Roles

Each task in the rPath Appliance Platform Agent has one or more roles to which that task is considered available. rAPA has a single built-in role of "admin" for all its standard plugins, and appliance developers can add other roles as appropriate. the *rAPA Plugin Development Guide* [http://docs.rpath.com/rAPA_Plugin_Development_Guide/index.html] covers how to add roles to plugins so that administrators can delegate responsibilities to those in other roles.

The *User Management* interface displays all the available roles for all the plugins loaded when rAPA starts. Select from the available roles when creating a user group to associate the roles with that group. This ensures users who are members of a particular user group have the appropriate available tasks when accessing the rAPA interfaces.

User Management: Custom Configuration

The only custom configuration options for the *User Management* task are those associated with using external authentication. These extend the `/etc/raa/custom.cfg` file as with other configuration directives. See Chapter 8, *External Authentication for rAPA* for further instruction and reference.

User Management: XML-RPC

When creating an XML-RPC client for managing and appliance using rAPA, reference the "Method Details" at the following API link to see what arguments to pass and what data types are returned for each of the exposed methods listed here:

<http://cvs.rpath.com/rapa-3.0-docs/raaplugins.usermanagement.web.usermanagement.UserInterface-class.html>

- `index(self)`
- `users(self)`
- `saveUserViaXmlrpc(self, username, pwd1, pwd2, groups=[])`
- `saveUser(self, username, pwd1, pwd2, groups=[])`
- `addUserViaXmlrpc(self, username, pwd1, pwd2, groups=[])`
- `addUser(self, username, pwd1, pwd2, groups=[])`
- `deleteUser(self, username)`
- `groups(self)`
- `deleteGroup(self, id)`

- `addGroup(self, groupName, roles=[], groupDesc=")`
- `saveGroup(self, groupName, roles=[])`
- `changePassword(self, oldpwd="", pwd1="", pwd2=")`

Standard Plugin: Flip-flop Update

Python identity: `/flipflop/Update`

The flipflop plugin, only available to customers using rAPA from `products.rpath.com`, replaces the default update mechanism (which uses Canary on a single partition) with an update mechanism that uses multiple alternating partitions. The plugin uses two or more system partitions and a special boot loader configuration to provide a failsafe in case a new system update causes issues on an appliance.

rPath recommends using the default *Updates* mechanism to provide the benefits of Canary during updates (see the section called “Standard Plugin: Updates”). The flipflop plugin provides an alternative to this recommendation for those that place a high priority on the time it takes to perform an update or to revert to a last known good state should an update fail.

When the appliance administrator selects to update an appliance configured to use this flip-flop mechanism, rAPA performs an appliance update as follows:

1. Upload or download the update image
2. Unmount the alternate system partition
3. Format the alternate system partition
4. Unpack the update image to the alternate system partition
5. Copy designated files from the current system partition to the alternate system partition
6. Configure the boot loader to boot once to the alternate system partition
7. Reboot the system to the alternate partition

After performing this update, the appliance user has the opportunity to test the updated system and to either accept the update or reboot the appliance to its "last known good" state.

Note

If the partition that is to be used as the update target is in use at the beginning of a flip-flop update, the update will fail and return an error. One way to check whether a device is in use is to use `fuser -m` on the system's command line.

To develop an appliance to use the flip-flop update mechanism, use the following as a checklist during development:

- On the appliance, include rAPA from `products.rpath.com`
- Use custom rAPA configuration (such as in a shadow of the `rapa-custom` package) to enable the flipflop plugin and to configure the flip-flop update mechanism.
- Ensure the appliance uses the GRUB bootloader with a GRUB configuration designed to chainload the alternating system partitions.

Flip-flop Update: Partitions and Kickstart

- Modify the `media-template` package for the appliance to include a kickstart file for automating an ISO installation.
- Include a separate partition or the appropriate files and scripts to preserve and copy data between the alternate partitions.
- If desired, include a signed update image mechanism.
- Configure a factory reset and any other partitions as desired.

Flip-flop Update: Partitions and Kickstart

As previously stated, the flipflop plugin uses two or more system partitions and a special boot loader configuration. The following partition structure supports the flip-flop update mechanism and provides an additional partition for preserving application data separate from the updating system:

- Boot partition
- Appliance/operating system partition 1
- Appliance/operating system partition 2
- Application data partition
- Swap partition
- Factory reset partition (if provided by appliance developers, see the section called “Standard Plugin: Factory Reset”)

The appliance installs on one of the two system partitions and makes it the default partition to which the appliance should boot. Developers should also ensure that when using the data partition for preserving application data, the appliance is configured to store all the critical application data on that partition. Whether or not the application data partition is used, though, developers can designate the files that should be copied between partitions during the update (see the section called “Flip-flop Update: Copy Data Between Partitions”).

To use the flip-flop update mechanism, developers must use the GRUB bootloader. GRUB is the default bootloader for appliances based on rPath Linux version 1, but SYSLINUX is the default for rPath Linux version 2 and requires appliance developers to override this default when developing the appliance. For flip-flop, the master boot record (`/dev/sda` or `/dev/hda`) contains a GRUB configured to read from the `/mainboot` configuration, the mount point default configured using the `flipflop.mounts.maingrub` directive. GRUB chainloads the two system partitions. The following is a sample for `/mainboot/grub/grub.conf` used on an appliance with the flip-flop update mechanism enabled:

```
#boot=/dev/sda
default=0
timeout=9
title Partition A
    root noverify (hd0,0)
    chainloader +1
title Partition B
    root noverify (hd0,1)
    chainloader +1
```

Flip-flop Update: Partitions and Kickstart

Appliance developers should install GRUB to the first system partition instead of the master boot record when installing the appliance from an ISO image. This and other GRUB configuration can be handled with a kickstart file used to automate the installation. The following is an example kickstart file which can be included by modifying the `media-template` package for your appliance:

```
bootloader --location=partition
clearpart --all
part /mainboot --fstype ext3 --size=20 --asprimary
part / --fstype ext3 --size=2048 --asprimary
part /alt --fstype ext3 --size=2048 --asprimary
part /factory --fstype ext3 --size=2048
part swap --size=1000 --grow --maxsize=2000
part /srv --fstype ext3 --size=100 --grow

...
%post --nochroot
disk=hda #Or sda depending on hardware
#echo "Remounting /dev/${disk}1" |tee /tmp/post.log
mount /tmp/${disk}1 /mnt/sysimage -o remount,ro

#echo "Unmounting /dev/${disk}2" |tee -a /tmp/post.log
umount /tmp/${disk}2

echo "Copying /tmp/${disk}1 to /tmp/${disk}2" |tee -a /tmp/post.log
dd if=/tmp/${disk}1 of=/tmp/${disk}2

echo "Remounting /dev/${disk}1" |tee -a /tmp/post.log
mount /tmp/${disk}1 /mnt/sysimage -o remount,rw
mount -t ext3 /tmp/${disk}2 /mnt/sysimage/alt

echo "Modifying fstab" |tee -a /tmp/post.log
cp /mnt/sysimage/etc/fstab /mnt/sysimage/etc/fstab.orig
sed -e 's/LABEL=\/[0-9]*\([ \t]*\/ \)\/dev\/'$disk'1\1/' -e 's/LABEL=\/alt

cp /mnt/sysimage/alt/etc/fstab /mnt/sysimage/alt/etc/fstab.orig
sed -e 's/LABEL=\/[0-9]*\([ \t]*\/ \)\/dev\/'$disk'2\1/' -e 's/LABEL=\/alt

echo "Modifying grub configuration" |tee -a /tmp/post.log
mkdir /mnt/sysimage/mainboot/grub
cp /mnt/sysimage/boot/grub/* /mnt/sysimage/mainboot/grub/
cat <<EOT >/mnt/sysimage/mainboot/grub/grub.conf
#boot=/dev/sda
default=0
timeout=10
title System Image 1
```

Flip-flop Update: Generate Update Images

```
        rootnoverify (hd0,0)
        chainloader +1
title System Image 2
        rootnoverify (hd0,1)
        chainloader +1
EOT

cp /mnt/sysimage/boot/grub/grub.conf /mnt/sysimage/boot/grub/grub.conf.orig
sed -e 's/root=LABEL=\\/[0-9]*/root=\\dev\\/'$disk'1/' </mnt/sysimage/boot/grub/grub.conf >/mnt/sysimage/boot/grub/grub.conf
sed -e 's/root=LABEL=\\/[0-9]*/root=\\dev\\/'$disk'2/' </mnt/sysimage/alt/boot/grub/grub.conf >/mnt/sysimage/alt/boot/grub/grub.conf

echo "Installing grub" |tee -a /tmp/post.log

echo "install (hd0,2)/grub/stage1 (hd0) (hd0,2)/grub/stage2 (hd0,2)/grub/grubstage2.img" >/mnt/sysimage/boot/grub/grub.conf
echo "install (hd0,0)/boot/grub/stage1 (hd0,0) (hd0,0)/boot/grub/stage2 (hd0,0)/boot/grub/grubstage2.img" >/mnt/sysimage/boot/grub/grub.conf
echo "install (hd0,1)/boot/grub/stage1 (hd0,1) (hd0,1)/boot/grub/stage2 (hd0,1)/boot/grub/grubstage2.img" >/mnt/sysimage/alt/boot/grub/grub.conf

#Optionally set up the factory images
echo "Creating Factory Reset Datasets"
/mnt/sysimage/bin/tar -czvplf /mnt/sysimage/factory/systemimage.tgz -C /mnt/sysimage
/mnt/sysimage/bin/tar -czvplf /mnt/sysimage/factory/dataimage.tgz -C /mnt/sysimage

#copy the post log into the system image.
mkdir -p /mnt/sysimage/root/postdebug
cp /tmp/post.log /mnt/sysimage/root/postdebug
```

For further reference on GRUB and customizing the appliance installer, see the following:

- www.gnu.org/software/grub/
- http://wiki.rpath.com/wiki/rPath_Linux:Installer_Customization/media-template

Flip-flop Update: Generate Update Images

To generate an update image to work with the flipflip plugin, use rBuilder create a compressed tar file from the same appliance group as the appliance image. Make that compressed tar file available as part of the rPath Appliance Platform Update Service (see the *rBuilder User Guide* [http://docs.rpath.com/rBuilder_User_Guide/index.html] for more information on generating and releasing appliance images).

Flip-flop Update: Copy Data Between Partitions

Whether or not a separate application data partition is used, developers can designate data that should be copied from the current partition to the alternate partition after the update image is unpacked there. Two configuration directives are used to manage accomplish this: *flipflop.files.system* to indicate a list of files that should be that should be included, and *flipflop.commands.applianceconfig* for a script that should be run (such as to copy a database or move a series of configurations).

rAPA copies over several settings by default. It preserves its own database (`/var/lib/raa/raadb`), and it copies the files indicated in the default value of *flipflop.files.system*.

Flip-flop Update: Signed Update Images

For more information about these configuration options, see their entries in the table in the section called “Flip-flop Update: Custom Configuration”.

Flip-flop Update: Signed Update Images

Appliance developers can configure an appliance to validate update images for a flip-flop update. This uses GnuPG signatures, requiring that `gpg:runtime` is installed as part of the appliance. To configure this feature, developers prepare the appliance as follows:

- Generate the GnuPG key for signing images on a system that has OpenPGP installed (packaged as `gnupg` for Conary-based systems). Use the `gpg --gen-key` and follow the prompts, and remember the passphrase you associate with the key.
- Package and install as part of the appliance the keyring to be used to validate signed images, ensuring it is installed to the location indicated by the `flipflop.validate_signature.key_ring` configuration directive for rAPA.
- Use the custom rAPA configuration package (such as `rapa-custom`) to set the following in the `[/flipflop]` section of `/etc/raa/custom.cfg`:
 - `flipflop.validate_signature` to "True"
 - `flipflop.update_check_url` to the URL of the FTP or web server that the appliance should check for update images
- Establish a system to create signed update images that can be validated by the keys on the appliance.

Use the following steps to create a signed update image:

1. Generate a *Compressed Tar File* image of the updated appliance.
2. Download the compressed tar file of the image to a system that has OpenPGP installed and that has a GnuPG signing key that can sign an image for validation with the key ring on the appliance.
3. Sign the compressed tar file of the image with the following command, replacing "filename.tgz" with the appropriate name of the downloaded file:

```
gpg --sign --compress-algo none filename.tgz
```

4. Upload the signed update image to an FTP server or web server from which appliances can download those images. Be sure that the value of `flipflop.update_check_url` in the `[/flipflop]` section of the rAPA configuration is set to the URL of this server.

If desired, verify the signed image file using `gpg --verify` on the filename ending in `.gpg`, or extract the image by running `gpg` on that filename. For more information about managing keys with GnuPG, see the GNU Privacy Handbook: <http://www.gnupg.org/gph/en/manual.html>

Industry-standard signing operations using GnuPG state that an individual is responsible for the signing key while the signing operation is executed by another individual (who is subject to the security policies created by the corporate entity). rPath recommends that the process NOT be automated in order to validate and verify individual signatures (for security purposes).

Flip-flop Update: Custom Configuration

Saving private keys within rBuilder or an Update Service appliance would introduce significant additional liability in securing those private keys on a networked server. However, because the image must be delivered to the end customer, a distribution method outside of rBuilder or rUS must be used, such as FTP or HTTP.

When validating against an expired public key, the Conary OpenPGP library uses the signature date (as opposed to the current date) when validating signatures. This means that a flip-flop update may be provided which updates the OpenPGP keyring itself, but that update needs to be signed before the original key expires in order for it to be validated by the deployed appliance.

Flip-flop Update: Custom Configuration

Use the following table as a guide to customizing the behavior of the flip-flop mechanism when it replaces the default update mechanism for the *Updates* task:

Table 11.16. Flip-flop Update Configuration

			<p>is executed while the alternate partition is mounted, this value is relative to that mount point. For example, if the alternate partition is mounted as /alt</p>	
<p>[/flipflop]</p>	<p>flipflop.filesystem</p>	<p>/etc/modprobe.conf', '/etc/hosts', '/etc/resolv.conf', '/etc/localtime', '/etc/sysconfig/{network,network-scripts/ifcfg-*, network-scripts/keys-*,clock,i18n,iptables,hwconf,firstboot,grub,keyboard,mouse}', '/etc/raa/raa-service.{privkey,pubkey}', '/etc/ssl/pem/raa.pem']</p>	<p>The list of flat files that should be copied to the alternate (updated) partition as part of a flip-flop update</p>	<p>no</p>
<p>[/flipflop]</p>	<p>flipflop.validate_signature</p>		<p>Validate updates for GPG signatures (require signed updates); requires that <code>gpg:runtime</code> is also included on the appliance</p>	<p>no</p>
<p>[/flipflop]</p>	<p>flipflop.validate_signature_keyring</p>	<p>keyring"</p>	<p>Location that contains the GPG keys to validate a signed update; if <code>flipflop.validate_signature</code> is "True," a GPG signature is required for the update image, and that image must be signed directly by a key contained in <code>/var/lib/raa/keyring/pubring.gpg</code></p>	<p>no</p>

Flip-flop Update: XML-RPC

The source code for this plugin is closed, but the Python methods for performing the tasks of a flip-flop update are exposed for XML-RPC. Contact rPath support under the terms of your customer support agreement to request the exposed methods and their expected arguments when programming an XML-RPC client for this task.

Standard Plugin: Factory Reset

Python identity: `/factoryreset/FactoryReset`

The factory reset plugin, only available to customers using rAPA from `products.rpath.com`, and it requires the flipflop plugin to be installed and configured, even if it is not used as the appliance's update mechanism (see the section called “Standard Plugin: Flip-flop Update”).

Though the alternate partition is available in GRUB as with the flip-flop update, the factory reset also changes the data partition, eliminating the ability to revert to the previous system state.

Factory Reset: Images

The factory reset image includes a system image and a data image, each in the form of a compressed tar file. The files are created when the appliance is installed, typically with commands executed from a kickstart file that is automating the ISO installation.

The factory reset reformats the system partition and unpacks the *system image* file into that partition. The default name expected by the factory reset plugin is `systemimage.tgz` and can be configured with the `factoryreset.image.system` configuration directive.

The factory reset reformats the data partition, mounts it, and unpacks the *data image* file into that partition. The default name expected by the factory reset plugin is `dataimage.tgz` and can be configured with the `factoryreset.image.data` configuration directive.

The following should be executed when the appliance is installed and can be included in the kickstart file as described in the section called “Flip-flop Update: Partitions and Kickstart”:

```
$> tar czplf /factory/systemimage.tgz /
$> tar czplf /factory/systemimage.tgz /srv
```

Factory Reset: Configure

Use the following table as a guide to customizing the behavior of the factory reset mechanism:

Factory Reset

			“Factory Reset: Images”	
<p><i>[factoryreset]</i></p> <p>factoryreset.partition</p>	<p>factoryreset.partition</p>	<p>data</p>	<p>Device associated with the factory reset data partition described in the section called “Factory Reset: Images”</p>	<p>no</p>
<p><i>[factoryreset]</i></p>	<p>factoryreset.mount</p>	<p>factory</p>	<p>Mount point on the appliance filesystem for the device associated with the factory reset system partition</p>	<p>no</p>
<p><i>[factoryreset]</i></p>	<p>factoryreset.mount</p>	<p>data</p>	<p>Mount point on the appliance filesystem for the device associated with the factory reset data partition</p>	<p>no</p>
<p><i>[factoryreset]</i></p>	<p>factoryreset.image</p>	<p>system</p>	<p>Filename expected for the system image on the filesystem mounted at <i>factoryreset.mount.system</i></p>	<p>no</p>
<p><i>[factoryreset]</i></p>	<p>factoryreset.image</p>	<p>data</p>	<p>Filename expected for the data image on the filesystem mounted at <i>factoryreset.mount.data</i></p>	<p>no</p>
<p><i>[factoryreset]</i></p>	<p>factoryreset.commands</p>	<p>pre_appliance_reset</p>	<p>Script to be run before the system is reset, such as to stop any services that could corrupt data; script name cannot have while spaces, and any arguments should be separated with spaces as when calling the script at the command line</p>	<p>no</p>

Table 11.17. Factory Reset Configuration

Factory Reset: XML-RPC

In addition to these configuration directives, be sure to adjust the following flip-flop update configurations as necessary to incorporate the factory reset partition: *flipflop.partitions.boot*, *flipflop.mount.boot*, and *flipflop.mount.maingrub*.

Factory Reset: XML-RPC

The source code for this plugin is closed, but the Python methods for performing the tasks of a factory reset are exposed for XML-RPC. Contact rPath support under the terms of your customer support agreement to request the exposed methods and their expected arguments when programming an XML-RPC client for this task.