Mac OS X pkginfo Test

# Introduction

OVAL is frequently used to discover what versions of applications, drivers, libraries, and other software packages have been installed on a system. On Windows, many software products write version information to the registry. On Linux, dpkg and rpm files can be checked. On Windows, OVAL provides the registry\_test. For Linux, the dpkginfo\_test and rpminfo\_test are available. And there are other package tests available in the various UNIX component schemas.

Is there a similar mechanism available on Apple’s Mac OS X? Yes.

Many Mac OS X applications are installed using the “Installer” program. The Installer writes receipts describing what was installed to a database. “pkgutil” is the tool Apple provides to examine these receipts.

The “man page” description of pkgutil is as follows:

pkgutil reads and manipulates Mac OS X Installer flat packages, and provides

access to the ``receipt'' database used by the Installer. Options are pro-

cessed first, and affect the operation of all commands. Multiple commands are

performed sequentially in the given order.

The files and directories where receipts are stored are subject to change.

Always use pkgutil to query or modify them.

We see from the man page that if the Installer is used to install a software package, then pkgutil should be used to examine the receipts. Therefore, it makes sense that OVAL should have a test for checking package state.

# pkgutil

pkgutil is a typical command line tool with various options and command arguments. It supports reading the receipt database, and manipulating package files in various ways. OVAL is a read-only tool so we only need concern ourselves with the read commands. Below is a list of the read commands that pkgutil provides.

Receipt Database Commands:

--pkgs, --packages List all currently installed package IDs on --volume

--pkgs-plist List all package IDs on --volume in plist format

--pkgs=REGEXP List package IDs on --volume that match REGEXP

--groups List all GROUPIDs on --volume

--groups-plist List all GROUPIDs on --volume in plist format

--group-pkgs GROUPID List all PKGIDs in GROUPID

--files PKGID List files installed by the specified package

--lsbom PKGID List files in the same format as 'lsbom -s'

--pkg-groups PKGID List all GROUPIDs that PKGID is a member of

--export-plist PKGID Print all info about PKGID in plist format

--verify PKGID Verify file permissions of the specified package

--pkg-info PKGID Show metadata about PKGID

--pkg-info-plist PKGID Show metadata about PKGID in plist format

--file-info PATH Show metadata known about PATH

--file-info-plist PATH Show metadata known about PATH in plist format

File Commands:

--payload-files PATH List the paths archived within the (m)pkg at PATH

# OVAL Test

Given the above set of capabilities, what commands make sense to incorporate in an OVAL test?

--pkgs

The list of package IDs could be parsed and pattern matched, but that seems a bit heavy. The REGEXP option is more interesting as it will list packages that match a regular expression. This could be used for discovering if a package is installed.

--groups

Would this be valuable?

--files & --lsbom

The rpminfo\_test provides a list of files associated with a package via a “filepath” behavior. Should this be added to this test?

--pkg-groups

Would knowing the groups a package is in be useful for OVAL’s typical usage?

--export-plist

This provides extensive information about a package including the list of files included in the package, and metadata about each file – including version. Is there information here that should also be collected?(See example output at the end of this document.)

--verify

Provides information on expected and actual permission and group settings. Does not support the use of a regular expression to evaluate multiple packages. Other component schemas have package verify type tests. This command may be a candidate for a similar Mac OS test.

--pkg-info

This reports the package id, version, volume where the package is located, location on the volume, install time, and group membership. (See example output at the end of this document.) This looks very much like data that would be useful to OVAL. It seems an OVAL test could be built around this command.

--file-info

Given a file path, this command will report various metadata including what packages installed that file. Would this be useful? And if so it probably ought to be a separate test.

--payload-files

Would examining a package file be useful?

From the above, it appears “--pkg-info” is the appropriate pkgutil command to build an OVAL test around. Given a package ID it is easy to get the version of the software package, which is consistent with OVAL package related tests on other platforms.

# OVAL Object

The object portion of an OVAL test specifies the system artifact(s) the data collection is to be performed against. For the pkgutil test, this would be a package ID. The pkgutil command provide two options that may be of interest here.

Options:

--volume PATH Perform all operations on the specified volume

--regexp Try all PKGID arguments as regular expressions

Would an OVAL audit ever need to be run against other than the root volume?

The --regexp option, however, is more interesting. Some packages encode their version inside their package name. For example, Adobe’s Acrobat Reader uses the package ID “com.adobe.acrobat.reader.11003.reader.app.pkg.en\_US”. And Microsoft Office uses package IDs such as:

com.microsoft.office.en.word\_wizards.pkg.14.3.0

com.microsoft.office.en.word\_wizards.pkg.14.3.2.update

com.microsoft.office.en.word\_wizards.pkg.14.3.5.update

In cases where version information is encoded in the package ID, it is important for OVAL to be able to select all package IDs matching a pattern and then be able to parse out the desired version information from the set of package IDs returned.

OVAL has mechanisms for using regular expressions in an object; specifically, the “pattern match” operation of the “string” datatype which uses the PCRE syntax. It is probably best to use OVAL’s mechanism rather than Apple’s for the sake of consistency.

# OVAL State

The state portion of an OVAL test specifies the desired values of the test. If the desired values are found, the test passes. The list of values available are:

Package-id:

This could be useful if a regular expression is used to collect information on multiple packages.

Version:

The version of the installed package. This is the primary information OVAL is after.

Volume:

The volume the installed package is on.

Location:

The path where the package was directed to be installed.

Install-time:

The install-time indicates when the package was installed in Unix seconds (seconds since 1970). This may be of value to OVAL if an audit is attempting to determine if a package was installed before or after a certain date.

Groups:

The “groups” the package is a part of. It is unclear how OVAL could use this.

Based on the list of possible metadata, the most useful appear to be: package-id, version, and install-time. The other values are easily accessible in the output, and not very long, so including them in the state definition may be worthwhile for completeness sake.

# OVAL System Characteristic Item

The system characteristic reports the actual collected values of an OVAL test. In this case, it can mirror the values specified in the state.

# Schema Design

As mentioned in the introduction, there are already OVAL tests in other component schemas that perform a similar function as what is being proposed here.

## Linux

|  |  |  |
| --- | --- | --- |
| **Element Name** | **Properties** | **Description** |
| dpkginfo\_test |  | The dpkginfo test is used to check information for a given DPKG package. |
| dpkginfo\_object | name | The dpkginfo\_object element is used by a dpkginfo test to define the object to be evaluated. |
| dpkginfo\_state | name  arch  epoch  release  version  evr | The dpkginfo\_state element defines the different information that can be used to evaluate the specified DPKG package. This includes the architecture, epoch number, release, and version numbers. |

|  |  |  |
| --- | --- | --- |
| **Element Name** | **Properties** | **Description** |
| rpminfo\_test |  | The rpminfo\_test is used to check the RPM header information for a given RPM package. |
| rpminfo\_object | name | A rpm info object consists of a single name entity that identifies the package being checked. |
| rpminfo\_state | name  arch  epoch  release  version  evr  signature\_keyid  extended\_name  filepath | The rpminfo\_state element defines the different information that can be used to evaluate the specified rpm. This includes the architecture, epoch number, and version numbers. Most of this information can be obtained through the rpm function. |
| behaviors | filepaths | when true, this behavior means collect all filepaths (directory and file information) from the rpm database for the package. |

|  |  |  |
| --- | --- | --- |
| **Element Name** | **Properties** | **Description** |
| slackwarepkginfo\_test |  | The slackware package info test is used to check information associated with a given Slackware package. |
| slackwarepkginfo\_object | name | A slackware package info object consists of a single name entity that identifies the package being checked. |
| slackwarepkginfo\_state | name  version architecture  revision | A slackware package info object consists of a single name entity that identifies the package being checked. |

## FreeBSD

|  |  |  |
| --- | --- | --- |
| **Element Name** | **Properties** | **Description** |
| portinfo\_test |  | The port info test is used to check the properties of a component of a FreeBSD system. |
| portinfo \_object | pkginst | A port info object consists of a single pkginst element that identifies a specific package. |
| portinfo \_state | pkginst  name  category  version vendor  description | The portinfo\_state element defines the different information that can be used to evaluate the specified package. |

## Solaris

|  |  |  |
| --- | --- | --- |
| **Element Name** | **Properties** | **Description** |
| package\_test |  | The package test is used to check information associated with different packages installed on the system. |
| package\_object | pkginst | A package object consists of a single pkginst entity that identifies the package to be used. |
| package \_state | pkginst  name  category  version vendor  description | The package\_state element defines the different information associated with packages installed on the system. |

Based on the above similar tests, I propose that the name of the test being discussed not be “pkgutil\_test” but “pkginfo\_test”. The “pkginfo” prefix is consistent with the naming convention already used by similar tests, and more accurately describes the operation being performed. Also, “pkgutil” ties the test name to the implementation tightly, which is a poor design. And as noted above, pkgutil supports a “--verify" command which may be a candidate for its own test, and other commands supported by pkgutil may warrant their own tests in the future.

Note also the “behaviors” element that the Linux rpminfo\_object supports. This tells the test to collect the list of files contained in the package. pkgutil supports what appears to be a similar capability with the “--files” command. (From reading the rpminfo\_test schema, it is unclear to me how the data collected using the “filepaths” behavior maps to the “rpminfo\_state” element.) The proposed test may want to support a similar capability – or implement another test to support the function.

See the accompanying schema and sample content documents for implementation examples.

### Version Field

This is defined as an “EntityStateVersionType”. That type has some rules and limitations associated with it. Can we be certain the version will always conform to this type?

### Install Time

This is reported from pkgutil as an integer. The value represents the number of seconds since 1970. This conforms with OVAL’s “seconds\_since\_epoch” DateTimeFormatEnumeration.

# Probe Implementation

The most straightforward way to implement this test would be to make a call to “pkgutil –pkg-info", parse the results, and use them. This is complicated slightly when OVAL’s regular expression or filter functionality is used. In those cases, it would seem a call to “pkgutil --pkgs" would be appropriate. Then apply the expression or filter to the resulting list and repeatedly call “pkgutil --pkg-info” on the matching package ids.

It does not appear the pkgutil functionality is exposed via a public API that could be called from C or another language.

# Example Output --export-plist

$ pkgutil --export-plist com.adobe.pkg.FlashPlayer

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

<dict>

<key>install-location</key>

<string></string>

<key>install-time</key>

<integer>1372179849</integer>

<key>paths</key>

<dict>

<key>Library</key>

<dict>

<key>gid</key>

<integer>80</integer>

<key>install-time</key>

<integer>1372179849</integer>

<key>mode</key>

<integer>1021</integer>

<key>pkg-version</key>

<string>11.7.700.225</string>

<key>pkgid</key>

<string>com.adobe.pkg.FlashPlayer</string>

<key>uid</key>

<integer>0</integer>

</dict>

… (removed several pages of path dictionary entries)

</dict>

<key>pkg-version</key>

<string>11.7.700.225</string>

<key>pkgid</key>

<string>com.adobe.pkg.FlashPlayer</string>

<key>receipt-plist-version</key>

<real>1</real>

<key>volume</key>

<string>/</string>

</dict>

</plist>

# Example Output –pkg-info

$ pkgutil --pkg-info com.adobe.pkg.FlashPlayer

package-id: com.adobe.pkg.FlashPlayer

version: 11.7.700.225

volume: /

location:

install-time: 1372179849

# References

<http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/pkgutil.1.html>

<http://www.mactech.com/articles/mactech/Vol.25/25.12/2512MacEnterprise-PackagesReceiptsandSnow/index.html>

<http://blog.bfitz.us/?p=1158>