Debian Cheat Sheet

Carlo Wood, March 2007

Table of contents

Finding things

If you want to find something, there are several classes of objects that you can start with. Click on the link describing what you already have.

- [Packages](#)

    These are the debian packages. They have a *name*.

- [Package files](#)

    These are the `.deb` files that contain a package. The first part of the file name is usually the package *name*.

- [Sources](#)

    A source tree containing a debian directory.

- [Installed Files](#)

    Any file that was installed and is part of a debian package.

- [Other files](#)

    Files that are not yet installed.

- [Some keyword](#)

    If you look for a packages related to some keyword.

Packages

When looking for information, you normally first want to find the **package name**. Here is how to **list all packages** that are installed:

```
dpkg -l
```

The package *names* are listed in the second column. The third column is the version, the fourth and last column a short description. The meaning of the first column is printed in the first three lines of the output. If the package name is already known, one can get this information for a single package by running:

```
dpkg -l name
```

Package files

These are the .deb files that contain a package. The first part of the file name up till the first underscore is usually the package name.

Information about a package file can be obtained with the command:

```
dpkg --info package.deb
```

This should extract the *name* of this package:

```
dpkg --info package.deb | grep '^Source: ' | sed -e 's/Source: //'
```

Sources

Sources are normally not installed. You can only install them if you know the package name. When installing the sources of a debian package, a package ending on `.dsc` (description) is downloaded along with an original tar ball and a debian specific diff (compressed). The `.dsc` will contain the name of the package, both, in its filename as well as content (after the `Source:` keyword).

Installed files

If you have a file `/path/filename` installed, and you want to know what package it belongs to (if any), then issue the following command:

```
dpkg -S /path/filename
```

This will print `name: /path/filename`.

Actually it's `dpkg -S filename-search-pattern`, where *filename-search-pattern* can contain the usual shell wildchars. The command can therefore also used to search in a fuzzy way for packages that have files installed matching the given pattern. It does however not find/return extra files created by maintainer scripts, nor will it list alternatives. You can therefore not conclude that it is safe to delete a file if `dpkg -S` doesn't return any packages.

Note that it's a capitial `-S`, a lower case `-s` returns status information for a package name.

Other files

In order to find packages that contain file patterns that you do not have installed, you need to install apt-file: `apt-get install apt-file`.

Like for `apt-get` you need to regularly synchronize the apt-file database by running,

```
sudo apt-file update
```

Then, search for the filename using,

```
apt-file search pattern
```

will give you a list of packages containing *files* matching *pattern*.

Some keyword

`apt-file search` is somewhat equivalent to

```
apt-cache search regexp
```

They use the same data from the repositories listed in your sources.list file (although each uses it's own cache and needs to be seperately synchronized with `apt-file update` and `apt-get update` respectively.)

The big difference is this: `apt-file search` matches against the *filenames* of packages, while `apt-cache search` matches against *package name*, filenames and the *long description* of packages. The second has an option to limit what is matched against though: `apt-cache --names-only search` only matches against the *package names*. Only `apt-file` has options to influence how the pattern is interpreted: `--regexp` (`-x`) treats the pattern as a regular expression, like `apt-cache`, while `--ignore-case` (`-i`) does a case insensitive search. Finally, `--fixed-string` (`-F`) interprets the given string as the trailing fixed string of the filename (thus, contrary what apt-file's man page says, the search pattern is still expanded with generic characters at pattern's start).

Queries

Now you have the package *name*, you can query the package database about it.

Contents

To print a list of all installed files (again excluding extra files as generated by maintainer script or alternatives), issue the command:

```
dpkg -L name
```

You can also use this command to find out if a package with that exact name is installed at all.

Or, if you have a package file that isn't installed yet, you can list it's contents with

```
dpkg --contents package.deb
```

Version / Short description

To print the version and a short description of of an *installed* package, type

```
dpkg -l name
```

Everything

More information about an installed package can be obtained with the status command:

```
dpkg -s name
```

Or, if the package isn't installed yet

```
dpkg --info package.deb
```

Customizing packages

- [Getting the source](#)
- [Building from source](#)
- [Creating a patch](#)
- [Adding a patch to a debian package](#)
- [Creating a source package](#)
- [Setting up a repository](#)
- [Pinning the packages of your local repository](#)

Getting the source

To work with Debian source packages, do as root:

```
apt-get install build-essential fakeroot devscripts debhelper docbook-to-man
```

Often, a single source package generates multiple binary packages. In order to find the name of the source that belongs to some package, use `dpkg -s`. For example,

```
dpkg -s name | grep '^Package:' | sed -e 's/Package: //'
```

should print the source name of the package.

This name can then be used to download the sources with the command:

```
apt-get source name
```

This command downloads the original source tar-ball (`.orig.tar.gz`), a file with some decriptive info (`.dsc`) and a (compressed) diff (`.diff.gz`). The tar-ball is unpacked and the diff is applied.

If for some reason you want to redo this later again, you can use the commands:

```
dpkg-source -x name_VERSION.dsc
```

or, manually (I'll remove this once I know the above addition really works ;)

```
tar xzf name_VERSION.orig.tar.gz
mv name_VERSION.orig name_VERSION
cd name_VERSION
gzip -dc ../name_VERSION.diff.gz | patch -p1 -s
chmod +x debian/rules
```

Building from source

In order to build a package, execute once as root:

```
apt-get build-dep name
```

This will install packages that are needed to build this package.

A source tree which supports debian has a debian directory in the root of the source tree. This directory contains a script called `rules`, which is actually a Makefile, but is executed as a script. There are several targets, including `configure`, `build`, `install` and `clean`. Normally you don't want to install the package onto your real system of course! Never run `debian/rules` as root therefore, or bad things might happen.

While testing changes that you make to the code, you can (re)build the package with the command:

```
debian/rules build
```

Of course, you can pass `-j <n>` to `debian/rules`, which is actually a self-executing Makefile, to do parallel builds.

If you want to debug this code, you should first set the following environment variable:

```
export DEB_BUILD_OPTIONS="nostrip,noopt"
```

This will prevent the binaries to be stripped and compile without optimization.

If you want to build a binary package, the package has to be 'installed'. The correct way to do this is by using the utility `fakeroot`, which fakes an install, so that you can build a binary package as non-root and without really installing it. The following command should build a binary package:

```
fakeroot debian/rules binary
```

The `binary` target is yet another Makefile target that will first execute a `configure`, `build` and `install` if still necessary. If you already did build the package, then only an `install` would first be done, etc.

This will have created a file `../somename_VERSION_ARCH.deb`, where `ARCH` is for example `i386`. More than one binary package can have been created.

Creating a patch

If you want to change the source code, you will need to make a diff of the changes that you made, otherwise it is not possible to create a debian package. In order to create this diff, you need to keep a copy of the original source tree, with all debian patches already applied. Therefore, before you make any changes, make a backup copy of the source tree using the following commands:

```
cd $NAME-$VERSION
fakeroot debian/rules clean
dpatch apply-all
cp -pr ../$NAME-$VERSION ../$NAME-$VERSION-origdeb
```

Next, you can make changes and test the code by building it with `debian/rules build`. Once you are satisfied, create a new patch as follows:

```
cd ..
diff -rc $NAME-$VERSION-origdeb $NAME-$VERSION > patchname.diff
```

Check if the diff looks sane (ie, doesn't contain unexpected hunks)

Adding a patch to a debian package

Add the patch to the source tree with (create the directory `debian/patches` if it does not already exist):

```
cd $NAME-$VERSION
dpatch patch-template -p "99_some_patch" "Description of patch" < ../patchname.diff > debian/patches/99_patchname.dpatch
chmod +x debian/patches/99_patchname.dpatch
```

Where the '99' should be chosen such that all your patches are applied in the correct order and after the original debian patches. Use `dpatch list-all` to obtain a list

Edit `debian/patches/99_patchname.dpatch` to fix your name and email address.

Edit `debian/patches/00list` (create it if it does not already exist) and add "99_patchname" to the list of patches.

Test if the patch applies with:

```
dpatch deapply-all
dpatch apply-all
```

Finally, add a changelog entry to `debian/changelog`. This is crucial since it is the only way to change the version of the package, reflecting that you made changes to it.

```
cd $NAME-$VERSION
dch -i
```

The format of the changelog entry is:

```
name (VERSION) unstable; urgency=low

  * Comments.

 -- Your Name <your@email> DATE
```

`dch -i` makes an 'official' increment to the VERSION, but that is probably not what you want. The VERSION that you use should be larger than the current version. For example, if the current version is 2.6.4-2, then you could change it into 2.6.4-2foo, or just add your initials. Do not add a hypen or dot. If uncertain, you can check if the version is considered larger with the command:
`dpkg --compare-versions 2.6.4-2 lt 2.6.4-2foo && echo OK`.

Now you can (re)build your debian package with `fakeroot debian/rules binary`.

Creating a source package

Because you made changes, you now also might want to create a source package (the original `.diff.gz` and `.dsc` files).

The command to do this is:

```
fakeroot debian/rules clean
cd ..
dpkg-source -b name-VERSION
```

Setting up a repository

In order to be able to continue to use `apt-get update` and `apt-get upgrade` painlessly, you should set up your own local repository with any packages that you made changes to.

For example, I made changes to the xchat package, and created a directory `/usr/src/dists/xchat` in which I downloaded the sources, and later created the `.deb` files. For example, an `ls` inside the directory `/usr/src/dists /xchat` on my machine shows:

```
/usr/src/dists/xchat>ls
overridefile   xchat_2.6.8-0.3.diff.gz     xchat_2.6.8-0.3run_i386.deb
Packages.gz    xchat_2.6.8-0.3.dsc         xchat_2.6.8.orig.tar.gz
Release        xchat_2.6.8-0.3run.diff.gz  xchat-common_2.6.8-0.3run_all.deb
xchat-2.6.8/   xchat_2.6.8-0.3run.dsc
```

We recognize the following files:

The files that were downloaded with the command `apt-get source xchat`:

- `xchat_2.6.8.orig.tar.gz`: The original source tar-ball.

- `xchat_2.6.8-0.3.diff.gz`: The original debian `.diff.gz`.

- `xchat_2.6.8-0.3.dsc`: The original debian `.dsc`.

The files generated by `fakeroot debian/rules binary`:

- xchat_2.6.8-0.3run_i386.deb

- xchat-common_2.6.8-0.3run_all.deb

New source package files generated with `dpkg-source -b xchat-2.6.8`:

- xchat_2.6.8-0.3run.diff.gz: The new debian `.diff.gz`.

- xchat_2.6.8-0.3run.dsc: The new debian `.dsc`.

Finally, we see three files that are needed to turn this directory into a local repository:

- overridefile

- Release

- Packages.gz

The contents of overridefile are:

```
xchat optional net Carlo Wood <carlo@alinoe.com>
xchat-common optional net Carlo Wood <carlo@alinoe.com>
```

Of course, replace "Carlo Wood <carlo@alinoe.com>" with your own name and email address. This string will be used for the 'Maintainer' of packages. The 'optional' and 'net' strings are respectively the 'Priority:' and 'Section:' status fields (dpkg -s or dpkg -info), for example use

```
dpkg --info xchat_2.6.8-0.3run_i386.deb | egrep '(Priority|Section):'
```

to print the original values.

The Packages.gz file is generated from that with the following command:

```
dpkg-scanpackages . overridefile | gzip > Packages.gz
```

The Release file contains:

```
Origin: Run
Label: Local repository
Suite: testing
Architectures: i386
Components: contrib
Description: Customized packages by Carlo Wood
MD5Sum:
 f71c05625f8690c2f9cae621dfbe0116 900 ./Packages.gz
```

There are several things important here: The *Suite:* must match whatever suite you are using (stable, testing, unstable). The *Origin:* must be a unique string that we will use in /etc/apt/preferences to pin this package to this repository: that way you won't accidently "upgrade" the package, losing your changes. The md5sum can be calculated with the command:

```
md5sum Packages.gz
```

The size (900 above) can be retrieved with ls -l Packages.gz.

Optionally, one can add more or other sums, using sha1sum (with label SHA1:), and/or using sha256sum (with label SHA256:). It is also possible to add an uncompressed Packages file. For example,

```
MD5Sum:
 70c80c40007f69ce04b9697bbe0ed3bd 1818 ./Packages
 f71c05625f8690c2f9cae621dfbe0116  900 ./Packages.gz
SHA1:
 44e38fde666e6d2093b8ae523e8bba17fa19549c 1818 ./Packages
 003e795f97c21b35da573f20c86f325cceabf29a  900 ./Packages.gz
SHA256:
 b4fef40d5d901a0b310ffcc1a8930e2d994d5c87b71926365061e3d3399616ed 1818 ./Packages
 03825be1af8a4f1b552928b190aaae3408af6601c2fb178245e823f46ef50b71  900 ./Packages.gz
```

Next, tell apt where the new repository is. Add like the following to your /etc/apt/sources.list file:

```
# Packages with local changes are listed here.
deb file:///usr/src/dists/xchat ./
```

where you have to replace the directory with the path that you used.

The final step needed is setting up the pinning, see the next paragraph. After that, you can simply run sudo apt-get update and apt-get install xchat, or whatever package you created.

Pinning the packages of your local repository

In order to give priority to the packages in your local repository, create a file /etc/apt/preferences with the following content:

```
Package: *
Pin: release o=Run
Pin-Priority: 1000
```

where you have to replace the string "Run" with whatever you used as `Origin:` in your Release file.

Important: If you have a line in your `/etc/apt/apt.conf` file like: `APT::Default-Release "testing";`, then *delete* that line from `/etc/apt/apt.conf`! The effect of that line is *exactly* the same as adding a pin with Pin-Priority of 990 for a release **a**rchive "testing" at the *top* of your `/etc/apt/preferences` file, completely destroying the functionality of how pinning works because you don't want that at the top of your preferences file. The reason for that is this: Packages are matched top down, and the first match found is used for a particular repository (independent of the priority, and independent of any existing version Pins (see below)) to set it's priority. Thus, if you put a package with the name 'xchat' in your local repository, and there is also a package with that name in the "testing" repository, then the first match would be for the version in the "testing" repository, setting the priority of your local repository to 990, regardless. If you use:

```
Package: *
Pin: release a=testing
Pin-Priority: 990

Package: *
Pin: release o=Run
Pin-Priority: 1000
```

in your `/etc/apt/preferences` file, or the equivalent of `APT::Default-Release "testing";` in your `/etc/apt/apt.conf` file, then the output of `apt-cache policy xchat` would be:

```
xchat-common:
  Installed: (none)
  Candidate: 2.6.8-0.3run
  Version table:
     2.6.8-0.3run 0
        990 file: ./ Packages
     2.6.8-0.3 0
        990 http://ftp.nluug.nl testing/main Packages
        500 http://ftp.nluug.nl unstable/main Packages
```

where our local repository has a priority of 990 as you see. See below for more explanations on how to interpret the output of `apt-get policy`.

We want our local repository to overrule, therefore it needs to be at the top instead:

```
Package: *
Pin: release o=Run
Pin-Priority: 1000

Package: *
Pin: release a=testing
Pin-Priority: 990
```

As above, replace "Run", and replace "testing" with whatever *you* have installed as Suite!

Pinning errata

The documentation of pinning is a bit incomplete and unclear. I had to debug the actual source code of apt in order to find out how it *really* worked. Hopefully, this document will answer questions of many others who are trying to do something with pinning. I am assuming that you already read all other documentation on pinning before you read this.

- There are three types of pinning: `Pin: release`, as used above, `Pin: origin` and `Pin: version`. The 'origin' here should be read as "site" and is matched against the site name of the url in the sources.list file (ie, a line `deb http://ftp.nluug.nl/pub/os/Linux/distr/debian/ testing main contrib non-free` would match against the string "ftp.nluug.nl"). Lines in sources.list that contain `file:` result in the empty string for this "site". The documentation of apt says that you can match against this empty string by using `Pin: origin ""` (note the double quotes), but this is wrong -- that matches against the string "\"\"" and will therefore never match anything.

- The `*` behind the `Package:` label is not a wildcard. It is a special case that means "everything". Wildcards are NOT supported.

- True wildcards in versions are also not supported. The only thing that is supported is a trailing wildcard.

Thus, "2.6*" will match both "2.6" and "2.6.18", but "*run" will not match anything, because there is no package with a version that contains the literal character '*'. A star can only appear once, and only as the last character in the version match pattern.

- The `Pin-Priority:` has a *different* meaning when used together with `Package: *` and `Pin: release` (or `origin`) than when used together with `Pin: version` Pin: or with a named `Package:`. In the first case, it actually sets the priority of (the instance found in) that repository, while in the latter case it ignores any repository package instances with a priority less than the one specified. One can list the priorities of the available repository package instances as well as the requested minimum priority with the command:

```
apt-cache policy {packagename}
```

A general output would be:

```
package-name:
  Installed: <installed-version>
  Candidate: <version-installed-when-doing-apt-get-upgrade>
  Package-Pin: <version-of-Pin-in-etc-apt-preferences>
  Version table:
 *** <some-version> <minimum-priority-to-consider>
      <priority-of-this-instance> <repository1>
      <priority-of-this-instance> <repository2>
 *** <some-other-version> <minimum-priority-to-consider>
      <priority-of-this-instance> <repository3>
      <priority-of-this-instance> <repository4>
```

For example, `apt-cache policy xchat-common` gives for me:

```
xchat-common:
  Installed: 2.6.8-0.3run
  Candidate: 2.6.8-0.3run
  Version table:
 *** 2.6.8-0.3run 0
      1000 file: ./ Packages
       100 /var/lib/dpkg/status
     2.6.8-0.3 0
       990 http://ftp.nluug.nl testing/main Packages
       500 http://ftp.nluug.nl unstable/main Packages
```

The `/var/lib/dpkg/status` stands for the installed package. The highest priority is thus 1000 for the (local) *repository* `./` (as we listed in the sources.list file, see [above](#)).

The repository package instance that will be used for upgrading is now found as follows: The largest value of the instance-priorities in the left column, but AT LEAST with the Pin-Priority value of the first 'Pin: version' that matches the package, or of an explicitly specified 'Package: name' that matches (the `<minimum-priority-to-consider>`). If there is no such priority (all values in the left column are less), than you get the error that no such package is available. The minimum priority to consider is listed after *every* version in the version table, not just after the version that actually matches, although it would have made more sense to have given it it's own header, for example 'Requested Priority: 0'.

For example, assume we add a "Pin: version" line to our `/etc/apt/preferences` as follows:

```
Package: *
Pin: release o=Run
Pin-Priority: 1000

Package: *
Pin: release a=testing
Pin-Priority: 990

Package: xchat-common
Pin: version 2.6.8-0.3
Pin-Priority: 1100
```

Again, using "Package: xchat*" is not supported, it would not match anything. Then the output of `apt-cache policy xchat-common` becomes:

```
xchat-common:
  Installed: 2.6.8-0.3run
  Candidate: 2.6.8-0.3
  Package pin: 2.6.8-0.3
  Version table:
 *** 2.6.8-0.3run 1100
      1000 file: ./ Packages
       100 /var/lib/dpkg/status
     2.6.8-0.3 1100
```

```
      990 http://ftp.nluug.nl testing/main Packages
      500 http://ftp.nluug.nl unstable/main Packages
```

The exact meaning of this is as follows: There is a package "xchat-common" installed, with version "2.6.8-0.3run". There exists a "Pin: version" for the package "xchat-common" that (first) matches "2.6.8-0.3". This version exists (in both, testing/main and unstable/main of ftp.nluug.nl), and therefore a "Package pin: 2.6.8-0.3" is printed. The Pin-Priority of this Package Pin is 1100, the value of which is printed regardless after every version in the Version table. This 1100 is a request, but since there does not exist such instance for this package, an upgrade will never happen.

Things are slightly different when no package of the given name is installed yet. Lets remove xchat and xchat-common with the command `sudo dpkg --purge xchat; sudo dpkg --purge xchat-common`, after which the output of `apt-cache policy xchat-common` is:

```
xchat-common:
  Installed: (none)
  Candidate: 2.6.8-0.3
  Package pin: 2.6.8-0.3
  Version table:
     2.6.8-0.3run 1100
       1000 file: ./ Packages
     2.6.8-0.3 1100
        990 http://ftp.nluug.nl testing/main Packages
        500 http://ftp.nluug.nl unstable/main Packages
```

A command `apt-get install xchat-common` would install the listed "Candidate:". This candidate version is initialized with the found "Package pin:" version. Because no repository has a priority higher than 1100, this version is not changed. Finally, the version from testing/main (priority 990) is actually installed, because that repository has the higher priority.

However, if we change the Pin-Priority to something below 1000, for example,

```
Package: xchat-common
Pin: version 2.6.8-0.3
Pin-Priority: 999
```

then `apt-cache policy xchat-common` shows us:

```
xchat-common:
  Installed: (none)
  Candidate: 2.6.8-0.3run
  Package pin: 2.6.8-0.3
  Version table:
     2.6.8-0.3run 999
       1000 file: ./ Packages
     2.6.8-0.3 999
        990 http://ftp.nluug.nl testing/main Packages
        500 http://ftp.nluug.nl unstable/main Packages
```

The Candidate changed into "2.6.8-0.3run"! The reason for that is, that now there exists a version (any) with a priority *higher* than the found pin (1000 > 999). So, you see that a version pin does NOT set a priority; it kicks in when any version exists anywhere that matches the version (only allowing a trailing wildcard) initializing the version that is going to be installed or upgraded, and then replacing that version with any version anywhere that has a LARGER priority than the requested version Pin-Priority, if any.

- There is a significant difference between `Package: *` and `Package: packagename`. In the latter case, the specified `Pin-Priority` is always a request (the minimum priority to consider). This is unfortunate, as it makes the pinning less flexible. For example, it is not possible to tell your system that you prefer `testing` over `unstable` for one specific package. I have a customized package `metacity`. When I patched it, I got the source from CVS (although I created my own .deb files), so the version was newer than anything else. Currently, the version in unstable is newer. I have lines for both unstable and testing in my /etc/apt /sources.list file because for some *other* packages I am interested in using unstable. However, now I get:

```
metacity:
  Installed: 1:2.17.8-svn-egg-run-13
  Candidate: 1:2.18.2-3
  Version table:
     1:2.18.2-3 0
        500 http://ftp.nluug.nl unstable/main Packages
 *** 1:2.17.8-svn-egg-run-13 0
        100 /var/lib/dpkg/status
     1:2.14.5-4 0
        990 http://ftp.nluug.nl testing/main Packages
```

As you see, the version of testing is still a lot older than my patch version (1:2.17.8-svn-egg-run-13), and therefore I see no reason to upgrade it yet. However, the Candidate: is now 1:2.18.2-3 from unstable - and upgrading I would get the version from unstable, which I certainly don't want.

So, now one could think that the following works:

```
Package: metacity
Pin: release a=unstable
Pin-Priority: -1
```

thinking that this would cause unstable to never be used for the package `metacity`. Unfortunately, this is *not* the case. The result is:

```
metacity:
  Installed: 1:2.17.8-svn-egg-run-13
  Candidate: 1:2.18.2-3
  Package pin: 1:2.18.2-3
  Version table:
     1:2.18.2-3 -1
        500 http://ftp.nluug.nl unstable/main Packages
 *** 1:2.17.8-svn-egg-run-13 -1
        100 /var/lib/dpkg/status
     1:2.14.5-4 -1
        990 http://ftp.nluug.nl testing/main Packages
```

As you see, the `-1` appears as "request" pin level. The effect is therefore that STILL version 1:2.18.2-3 from `unstable` is the candidate, because 500 > 100 is the highest priority available for which the version 1:2.18.2-3 > 1:2.17.8-svn-egg-run-13.

The final conclusion has to be that not only the official documentation of `apt`, in regard to pinning, sucks— but the whole design and algorithm that determines the candidate version is borked. It is counter-intuitive, ambiguous and not as flexible as it could be.

If this documentation helped you, then I'd appreciate it if you dropped me a mail: carlo@alinoe.com.