Ubuntu Packaging Guide

Publicación 1.0.0 bzr655 ubuntu14.04.1

Ubuntu Developers

09 de April de 2018

Índice general

1. Artículos **2**
- 1.1. Introducción al desarrollo de Ubuntu . . . 2
- 1.2. Fase de preparación . . . 4
- 1.3. Corregir un fallo en Ubuntu . . . 8
- 1.4. Empaquetando nuevo software . . . 14
- 1.5. Actualizaciones de seguridad y de versiones estables . . . 17
- 1.6. Parches a los paquetes . . . 19
- 1.7. Corrección de paquetes que no compilan a partir del código fuente (FTBFS) . . . 23
- 1.8. Bibliotecas compartidas . . . 24
- 1.9. Adaptar actualizaciones de software a versiones anteriores . . . 26

2. Base de conocimiento **28**
- 2.1. Comunicación en el desarrollo de Ubuntu . . . 28
- 2.2. Descripción general básica del Directorio `debian/` . . . 28
- 2.3. ubuntu-dev-tools: Tools for Ubuntu developers . . . 34
- 2.4. autopkgtest: pruebas automáticas para paquetes . . . 36
- 2.5. Usar chroots . . . 39
- 2.6. Setting up sbuild . . . 40
- 2.7. Empaquetado de KDE . . . 42

3. Lecturas adicionales **44**

Welcome to the Ubuntu Packaging and Development Guide! We are currently developing codename Bionic Beaver, which is to be released in April 2018 as Ubuntu 18.04 LTS.

This is the official place for learning all about Ubuntu Development and packaging. After reading this guide you will have:

- Heard about the most important players, processes and tools in Ubuntu development,
- Your development environment set up correctly,
- A better idea of how to join our community,
- Fixed an actual Ubuntu bug as part of the tutorials.

Ubuntu no solo un sistema operativo libre, gratuito y de código abierto, su plataforma también es abierta y está desarrollada de forma transparente. El código fuente para cada uno de los componentes se puede obtener fácilmente y todos los cambios realizados a la plataforma de Ubuntu pueden ser revisados.

Esto significa que puede involucrarse activamente en mejorarlo y que la comunidad de desarrolladores de la plataforma de Ubuntu está siempre interesada en ayudar a las personas que comienzan.

Ubuntu es además una comunidad de personas estupendas que creen en el software libre y que éste debería estar accesible para todos. Sus miembros son amigables y quieren que se una. Queremos que se involucre, que realice preguntas, que ayude a hacer Ubuntu mejor junto a nosotros.

Si se topa con un problema: ¡no se preocupe! Consulte el *artículo sobre comunicación* y sabrá cómo puede contactar a otros desarrolladores fácilmente.

La guía está separada en dos secciones:

- Una lista de artículos basados en tareas, cosas que puede desear que se hagan.
- Un conjunto de artículos de la base de conocimientos que profundizan en partes especificas de las herramientas y flujos de trabajo.

CAPÍTULO 1

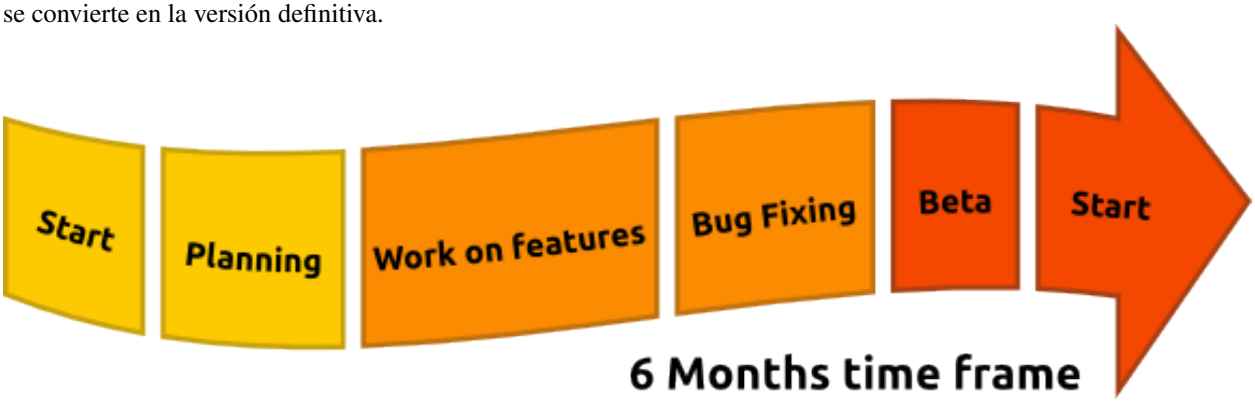
Artículos

1.1 Introducción al desarrollo de Ubuntu

Ubuntu está formado por miles de componentes distintos, escritos en muchos lenguajes de programación diferentes. Cada componente, ya sea una biblioteca software, una herramienta o una aplicación gráfica, está disponible en forma de paquete fuente. Los paquetes fuente, en la mayoría de los casos, constan de dos partes: el código fuente en sí, y los metadatos. Los metadatos incluyen las dependencias del paquete, los derechos de autor e información sobre la licencia e instrucciones sobre cómo compilar el paquete. Una vez que el paquete está compilado, el proceso de construcción proporciona los paquetes binarios, que son los archivos .deb que el usuario puede instalar.

Every time a new version of an application is released, or when someone makes a change to the source code that goes into Ubuntu, the source package must be uploaded to Launchpad's build machines to be compiled. The resulting binary packages then are distributed to the archive and its mirrors in different countries. The URLs in `/etc/apt/sources.list` point to an archive or mirror. Every day images are built for a selection of different Ubuntu flavours. They can be used in various circumstances. There are images you can put on a USB key, you can burn them on DVDs, you can use netboot images and there are images suitable for your phone and tablet. Ubuntu Desktop, Ubuntu Server, Kubuntu and others specify a list of required packages that get on the image. These images are then used for installation tests and provide the feedback for further release planning.

El desarrollo de Ubuntu depende en gran medida de la fase actual del ciclo de publicación. Se publica una nueva versión de Ubuntu cada seis meses, lo que solo es posible porque se han establecido fechas estrictas de congelación. Con cada fecha de congelación que se va alcanzando, los desarrolladores esperan hacer menos cambios y menos intrusivos. La congelación de funciones («feature freeze») es la primera gran fecha de congelación después de haber pasado la primera mitad del ciclo. En esta fase, las funciones deben estar ya prácticamente implementadas. El resto del ciclo se supone que se centrará en corregir errores. Después de que la interfaz de usuario, la documentación, el núcleo, etc. se congelan, se publica una versión beta que pasa gran cantidad de pruebas. De la versión beta en adelante, solo se corrigen errores críticos y se publica la versión candidata, la cual, si no contiene ningún problema serio, es la que se convierte en la versión definitiva.



Miles de los paquetes fuente, miles de millones de líneas de código, cientos de contribuyentes requieren de mucha comunicación y planificación para mantener altos estándares de calidad. Al principio y hacia la mitad de cada ciclo de emisión tiene lugar la «Ubuntu Developer Summit» (cumbre de desarrolladores de Ubuntu) en la que los desarrolladores y contribuyentes se juntan para planificar las características de las próximas versiones. Cada característica es discutida por sus accionistas y se escribe una especificación que contiene información detallada sobre sus supuestos, implementación, cambios necesarios en otras partes, cómo probarla y así sucesivamente. Todo esto se hace de forma abierta y transparente, así que puede participar remotamente y escuchar una transmisión de vídeo, chalar con los asistentes y suscribirse a cambios en las especificaciones, de forma que siempre esté al día.

Sin embargo, no todos los cambios pueden ser discutidos en una reunión, particularmente porque Ubuntu depende de cambios que se hacen en otros proyectos. Es por esto que los contribuyentes se mantienen en contacto permanente. La mayoría de los equipos o proyectos usan listas de correo dedicadas para evitar demasiado ruido no relacionado. Para una coordinación más inmediata, los desarrolladores y contribuyentes usan charlas en IRC («Internet Relay Chat»). Todas las discusiones son abiertas y públicas.

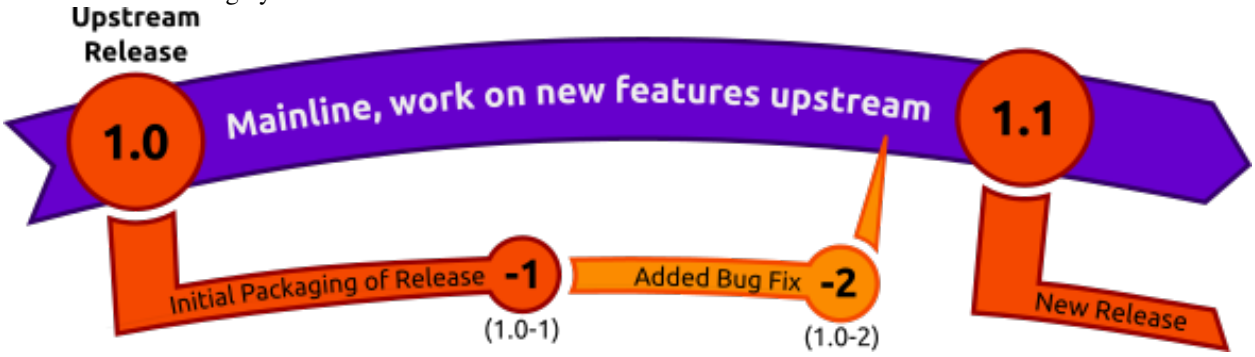
Otra herramienta importante relacionada con la comunicación son los informes de errores. Siempre que se encuentra un error en un paquete o en un parte de la infraestructura, se rellena un informe de error en Launchpad. Se recoge toda la información en dicho informe y se actualiza cuando sea necesario su importancia, estado y desarrollador asignado. Esto lo convierte en una herramienta efectiva para mantenerse al día de los errores de un paquete o proyecto y para organizar la carga de trabajo.

La mayoría del software disponible a través de Ubuntu no está escrito por los propios desarrolladores de Ubuntu. La mayoría está escrito por otros desarrolladores de otros proyectos de código abierto que luego son integrados en Ubuntu. Estos proyectos se denominan «upstreams» (aguas arriba), porque su código fuente fluye a Ubuntu, donde «simplemente» se integra. La relación con los proyectos «upstream» tiene una importancia crítica para Ubuntu. No es solo código lo que Ubuntu recoge aguas arriba, sino que los proyectos «upstream» obtienen también usuarios, informes de error y parches de Ubuntu (y de otras distribuciones).

El proyecto «upstream» más importante para Ubuntu es Debian. Debian es la distribución en la que se basa Ubuntu y muchas de las decisiones de diseño relativas a la infraestructura de empaquetado se hacen allí. Traicionalmente, Debian ha tenido siempre mantenedores dedicados para cada paquete individual o equipos dedicados de mantenimiento. En Ubuntu hay equipos que tienen interés en un subconjunto de paquetes también y, naturalmente, cada desarrollador tiene un área de especialización, pero la participación (y permisos de subida) está generalmente abierta a cualquiera que demuestre capacidad y voluntad.

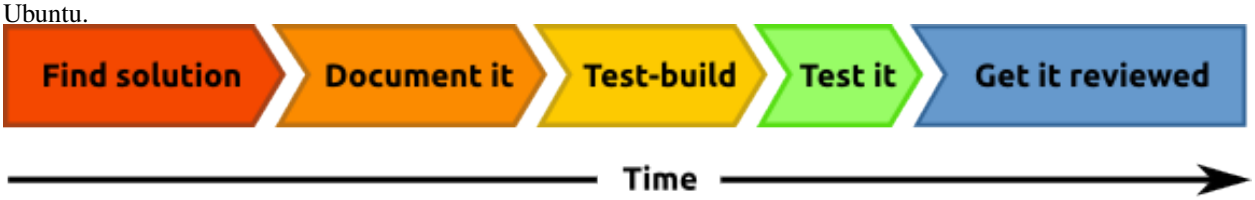
Conseguir un cambio en Ubuntu como un nuevo contribuyente no es tan desalentador como parece y puede resultar una experiencia muy gratificante. No se trata únicamente de aprender algo nuevo y excitante, sino también de compartir la solución y resolver un problema para millones de otros usuarios.

Los desarrollos de código abierto se producen en un mundo distribuido con distintos objetivos diferentes y con áreas de interés diferentes. Por ejemplo, podría darse el caso de que alguien aguas arriba («upstream») esté interesado en trabajar en una nueva funcionalidad importante mientras que Ubuntu, debido a su estricta planificación de emisiones, está interesado en distribuir una versión sólida solo con algunas correcciones de errores. Es por ello que se usa el «desarrollo distribuido», en el que se trabaja en distintas ramas de código que son combinadas con las demás después de revisiones de código y suficientes discusiones.



En el ejemplo mencionado anteriormente tendría sentido emitir Ubuntu con una versión existente del proyecto, añadir la corrección del error, integrarla aguas arriba («upstream») para su próxima emisión y distribuirla (si fuera conveniente) en la próxima emisión de Ubuntu. Sería el mejor compromiso posible y una situación en la que todo el mundo gana.

Para arreglar un error en Ubuntu, primero debería obtener el código fuente del paquete, después trabajar en la solución, documentarla de forma que sea fácil de entender por otros desarrolladores y usuarios y luego compilar el paquete para probarlo. Una vez lo haya probado, puede proponer fácilmente que el cambio se incluya en la publicación actualmente en desarrollo de Ubuntu. Un desarrollador con permisos para subir el código lo revisará y hará que se integre en Ubuntu.



Cuando intente encontrar una solución, habitualmente es una buena idea comprobar los proyectos aguas arriba («upstreams») para ver si el problema (o una posible solución) es conocido y, si no, hacer lo posible para que la solución sea un esfuerzo concertado.

Podrían ser necesarios pasos adicionales para hacer que el cambio se incluya en una versión antigua, aunque todavía soportada de Ubuntu, y enviarlo aguas arriba («upstream»).

Los requisitos más importantes para tener éxito en el desarrollo de Ubuntu son: tener una habilidad especial para «hacer que las cosas funcionen de nuevo», no tener miedo a leer documentación y a hacer preguntas, ser un jugador de equipo y disfrutar con algo de trabajo detectivesco.

Good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on `freenode`.. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

1.2 Fase de preparación

Hay una serie de cosas que necesita hacer para poder empezar a desarrollar para Ubuntu. Este artículo está diseñado para que pueda configurar su equipo de forma que pueda comenzar a trabajar con paquetes y subirlos a Launchpad, la plataforma de hospedaje de Ubuntu. Los temas cubiertos son:

- Instalar software relacionado con el empaquetado. Incluye:
  - Utilidades de empaquetado específicas de Ubuntu
  - Software de cifrado, para poder verificar que el trabajo se hizo por el usuario indicado
  - Software de cifrado adicional para poder enviar archivos de forma segura
- Crear y configurar una cuenta en Launchpad
- Configurar un entorno de desarrollo para ayudarle compilar paquetes localmente, interactuar con otros desarrolladores y proponer sus cambios en Launchpad.

Nota: Se recomienda hacer el trabajo de empaquetado usando la versión actual en desarrollo de Ubuntu. Esto le permitirá probar los cambios en el mismo entorno en el que serán realmente aplicados y usados.

Don't want to install the latest development version of Ubuntu? Spin up an LXD container.

1.2.1 Instalar software básico de empaquetado

There are a number of tools that will make your life as an Ubuntu developer much easier. You will encounter these tools later in this guide. To install most of the tools you will need run this command:

```
$ sudo apt install gnupg pbuilder ubuntu-dev-tools apt-file
```

Esta orden instalará el software siguiente:

- `gnupg` – GNU Privacy Guard contains tools you will need to create a cryptographic key with which you will sign files you want to upload to Launchpad.
- `pbuilder` – una herramienta para realizar compilaciones reproducibles de una paquete en un entorno limpio y aislado.
- `ubuntu-dev-tools` (y `devscripts`, una dependencia directa) – una colección de herramientas que hace muchas tareas del empaquetado más sencillas.
- `apt-file` proporciona una forma sencilla de encontrar el paquete binario que contiene un archivo determinado.

Crear su clave GPG

GPG stands for GNU Privacy Guard and it implements the OpenPGP standard which allows you to sign and encrypt messages and files. This is useful for a number of purposes. In our case it is important that you can sign files with your key so they can be identified as something that you worked on. If you upload a source package to Launchpad, it will only accept the package if it can absolutely determine who uploaded the package.

Para generar una nueva clave GPG, ejecute:

```
$ gpg --gen-key
```

GPG primero le preguntará qué tipo de clave desea generar. Elegir el tipo por defecto (RSA y DSA) es correcto. Después le preguntará por el tamaño de la clave. El predeterminado (actualmente 2048) es adecuado, pero 4096 es más seguro. Posteriormente le preguntará si desea que la clave caduque en algún momento. Es seguro decir «0», lo que significa que nunca expirará. Las últimas preguntas serán sobre su nombre y dirección de correo electrónico. Aquí simplemente elija los que quiere usar para el desarrollo de Ubuntu, y puede añadir direcciones de correo electrónico adicionales posteriormente. No es necesario añadir un comentario. Luego tendrá que establecer una frase de paso, elija una segura (una frase de paso es simplemente una contraseña en la que se permite la inclusión de espacios).

Ahora GPG creará la clave, lo que le puede llevar un tiempo; necesita algunos bytes aleatorios, así que si le da al sistema algún trabajado para hacer, mejor. Mueva el cursor, escriba algunos párrafos de texto aleatorio o cargue alguna página web.

Una vez hecho esto, recibirá un mensaje similar a:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

En este caso `43CDE61D` es el *ID de clave*.

Después necesitará cargar la parte pública de su clave a un servidor de claves, de forma que el resto del mundo pueda identificar los mensajes y archivos como suyos. Para hacerlo, escriba:

```
$ gpg --send-keys --keyserver keyserver.ubuntu.com <KEY ID>
```


Esto mandará su clave al servidor de claves de Ubuntu, pero una red de servidores de claves la sincronizarán automáticamente entre ellos. Una vez que se complete esta sincronización, su clave pública firmada estará lista para verificar sus contribuciones por todo el mundo.

Crear su clave SSH

SSH viene de *Secure Shell* (intérprete de órdenes seguro) y es un protocolo que le permite intercambiar datos de una forma segura en una red. Es habitual usar SSH para acceder a un intérprete de órdenes en otro equipo y usarlo para transferir archivos de forma segura. Para nuestros propósitos, usaremos SSH básicamente para subir paquetes fuentes a Launchpad.

Para generar una clave SSH, escriba:

```
$ ssh-keygen -t rsa
```

El nombre de archivo por defecto normalmente tiene sentido, así que puede dejarlo tal cual. Por motivos de seguridad, se recomienda encarecidamente que use una frase de paso.

Configurar pbuilder

`pbuilder` le permite compilar paquetes localmente en su equipo. Sirve para un par de propósitos:

- La compilación se hará en un entorno mínimo y limpio. Esto le ayuda a asegurarse de que sus compilaciones se completan de una forma reproducible, pero sin modificar su sistema local.
- No es necesario instalar todas las *dependencias de compilación* necesarias de forma local.
- Puede configurar varias instancias para distintas versiones de Ubuntu y Debian.

Configurar `pbuilder` es muy fácil, ejecute:

```
$ pbuilder-dist <release> create
```

where <release> is for example *xenial*, *zesty*, *artful* or in the case of Debian maybe *sid* or *buster*. This will take a while as it will download all the necessary packages for a "minimal installation". These will be cached though.

1.2.2 Preparar una configuración para trabajar con Launchpad

Con una configuración local básica establecida, el siguiente paso será configurar el sistema para trabajar con Launchpad. Esta sección se centrará en los siguientes temas:

- Qué es Launchpad y la creación de una cuenta de Launchpad
- Subir claves GPG y SSH a Launchpad
- Configure your shell to recognize you (for putting your name in changelogs)

Acerca de Launchpad

Launchpad es la pieza central de la infraestructura usada en Ubuntu. No sólo almacena todos sus paquetes y código, sino también cosas tales como traducciones, informes de errores e información sobre la gente que trabaja en Ubuntu y su pertenencia a equipos. También usará Launchpad para publicar sus propuestas de solución y hacer que otros desarrolladores de Ubuntu las revisen y esponsoricen.

Necesitará registrarse en Launchpad y proporcionar una cantidad mínima de información. Esto le permitirá descargar y subir código, enviar informes de error y más cosas.

Además de hospedar a Ubuntu, Launchpad puede hospedar a cualquier otro proyecto de software libre. Para más información véase la Launchpad Help wiki.

Obtener una cuenta de Launchpad

If you don't already have a Launchpad account, you can easily create one. If you have a Launchpad account but cannot remember your Launchpad id, you can find this out by going to https://launchpad.net/~ and looking for the part after the ~ in the URL.

El proceso de registro de Launchpad le pedirá que elija un nombre de usuario. Se recomienda que use su nombre real de forma que los colegas desarrolladores de Ubuntu lleguen a conocerle mejor.

Cuando registre una nueva cuenta, Launchpad le enviará un correo con un enlace que deberá abrir en su navegador para verificar su dirección de correo electrónico. Si no lo recibe, compruebe su carpeta de correo no deseado (spam).

The new account help page on Launchpad has more information about the process and additional settings you can change.

Subir su clave GPG a Launchpad

First, you will need to get your fingerprint and key ID.

Para buscar su huella de GPG, ejecute:

```
$ gpg --fingerprint email@address.com
```

y le mostrará algo como:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

Then run this command to submit your key to Ubuntu keyserver:

```
$ gpg --keyserver keyserver.ubuntu.com --send-keys 43CDE61D
```

where `43CDE61D` should be replaced by your key ID (which is in the first line of output of the previous command). Now you can import your key to Launchpad.

Diríjase a https://launchpad.net/~/+editpgpkeys y copie la huella de la clave («Key fingerprint») en la casilla de texto. En el caso anterior sería `5C28 0144 FB08 91C0 2CF3 37AC 6F0B F90F 43CD E61D`. Ahora pulse en «Import Key» (importar clave).

Launchpad usará la huella para comprobar su clave en el servidor de claves de Ubuntu. Si tiene éxito, le enviará un correo electrónico cifrado, pidiéndole que confirme la clave importada. Compruebe su cuenta de correo electrónico y lea el mensaje que le ha enviado Launchpad. *Si su cliente de correo electrónico soporta el cifrado de OpenPGP, le pedirá la contraseña que eligió para la clave cuando la generó GPG. Escriba la contraseña, y luego pulse en el enlace que confirma que esa clave es suya.*

Launchpad encrypts the email, using your public key, so that it can be sure that the key is yours. If you are using Thunderbird, the default Ubuntu email client, you can install the Enigmail plugin to easily decrypt the message. If your email software does not support OpenPGP encryption, copy the encrypted email's contents, type `gpg` in your terminal, then paste the email contents into your terminal window.

De vuelta a la web de Launchpad, use el botón «Confirm» (confirmar) y Launchpad completará la importación de su clave OpenPGP.

Encuentre más información en https://help.launchpad.net/YourAccount/ImportingYourPGPKey

Subir su clave SSH a Launchpad

Abra https://launchpad.net/~/+editsshkeys en su navegador web y abra también `~/.ssh/id_rsa.pub` en un editor de textos. Esta es la parte pública de su clave SSH, así que es seguro compartirla en Launchpad. Copie el contenido del archivo y péguelo en la casilla de texto de la página web que dice «Add an SSH key» (añadir una clave SSH). Ahora pulse «Import Public Key» (importar clave pública).

For more information on this process, visit the creating an SSH keypair page on Launchpad.

Configurar el intérprete de órdenes

The Debian/Ubuntu packaging tools need to learn about you as well in order to properly credit you in the changelog. Simply open your *~/.bashrc* in a text editor and add something like this to the bottom of it:

```
export DEBFULLNAME="Bob Dobbs"
export DEBEMAIL="subgenius@example.com"
```

Ahora guarde el archivo y reinicie su terminal o ejecute:

```
$ source ~/.bashrc
```

(si no usa el intérprete de órdenes por defecto, que es *bash*, edite el archivo de configuración de dicho intérprete como corresponda).

1.3 Corregir un fallo en Ubuntu

1.3.1 Introducción

Si se han seguido las instrucciones de *ponerse en marcha con el desarrollo de Ubuntu*, debería estar preparado para comenzar.

Como se puede observar en la imagen superior, no hay sorpresas en el proceso de corrección de fallos en Ubuntu: se encuentra un problema, se descarga el código fuente, se trabaja en la solución, se prueba, se envían los cambios a Launchpad y se pide que sea revisado e integrado. Durante esta guía se pasará por todos estos pasos, uno a uno.

1.3.2 Encontrar el problema

Hay muchas formas de encontrar cosas en las que trabajar. Puede ser reportando un error que usted mismo haya encontrado (lo que le da una buena oportunidad de probar la solución), o un problema que haya encontrado en otro lugar, tal vez en un informe de error.

Take a look at the bitesize bugs in Launchpad, and that might give you an idea of something to work on. It might also interest you to look at the bugs triaged by the Ubuntu One Hundred Papercuts team.

1.3.3 Averiguar qué se debe corregir

Si no se conoce el paquete fuente que contiene el código con el problema, pero sí la ruta al programa afectado en su sistema, puede encontrar el paquete en el que necesitará trabajar.

Let's say you've found a bug in Bumprace, a racing game. The Bumprace application can be started by running `/usr/bin/bumprace` on the command line. To find the binary package containing this application, use this command:

```
$ apt-file find /usr/bin/bumprace
```

Esto mostrará:

```
bumprace: /usr/bin/bumprace
```

Note that the part preceding the colon is the binary package name. It's often the case that the source package and binary package will have different names. This is most common when a single source package is used to build multiple different binary packages. To find the source package for a particular binary package, type:

```
$ apt-cache showsrc bumprace | grep ^Package:
Package: bumprace
$ apt-cache showsrc tomboy | grep ^Package:
Package: tomboy
```

`apt-cache` es parte de la instalación estándar de Ubuntu.

1.3.4 Confirmar el problema

Once you have figured out which package the problem is in, it's time to confirm that the problem exists.

Digamos que el paquete `bumprace` no tiene una página web en su descripción. Como primer paso podría comprobar que el problema no esté ya resuelto. Es fácil de hacer, bien mirando al Centro de software o ejecutando:

```
apt-cache show bumprace
```

La salida debería ser algo parecido a esto:

```
Package: bumprace
Priority: optional
Section: universe/games
Installed-Size: 136
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XNBC-Original-Maintainer: Christian T. Steigies <cts@debian.org>
Architecture: amd64
Version: 1.5.4-1
Depends: bumprace-data, libc6 (>= 2.4), libsdl-image1.2 (>= 1.2.10),
         libsdl-mixer1.2, libsdl1.2debian (>= 1.2.10-1)
Filename: pool/universe/b/bumprace/bumprace_1.5.4-1_amd64.deb
Size: 38122
MD5sum: 48c943863b4207930d4a2228cedc4a5b
SHA1: 73bad0892be471bbc471c7a99d0b72f0d0a4babc
SHA256: 64ef9a45b75651f57dc76aff5b05dd7069db0c942b479c8ab09494e762ae69fc
Description-en: 1 or 2 players race through a multi-level maze
 In BumpRacer, 1 player or 2 players (team or competitive) choose among 4
 vehicles and race through a multi-level maze. The players must acquire
 bonuses and avoid traps and enemy fire in a race against the clock.
 For more info, see the homepage at http://www.linux-games.com/bumprace/
Description-md5: 3225199d614fba85ba2bc66d5578ff15
```



```
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Origin: Ubuntu
```

Un contraejemplo sería `gedit`, el cual tiene definida una página web:

```
$ apt-cache show gedit | grep ^Homepage
Homepage: http://www.gnome.org/projects/gedit/
```

A veces se encontrará con que un problema concreto que estaba mirando ya ha sido corregido. Para evitar desperdiciar esfuerzos y duplicar trabajos tiene sentido realizar antes cierto trabajo de investigación.

1.3.5 Investigar la situación de un error

Primero debería comprobar si ya existe un informe de error para el problema en Ubuntu. Quizá ya haya alguien trabajando en una corrección o podamos contribuir de alguna forma a la solución. Para Ubuntu echamos un vistazo rápido a https://bugs.launchpad.net/ubuntu/+source/bumprace y vemos que no hay un error abierto aquí para nuestro problema.

Nota: Para Ubuntu la URL `https://bugs.launchpad.net/ubuntu/+source/<package>` siempre debería llevar la página de error del paquete fuente en cuestión.

Para Debian, que es la fuente principal de paquetes para Ubuntu, echamos un vistazo a http://bugs.debian.org/src:bumprace y tampoco pudimos encontrar un informe de error sobre nuestro problema.

Nota: Para Debian la URL `http://bugs.debian.org/src:<package>` siempre debería llevar la página de error del paquete fuente en cuestión.

El problema en el que estamos trabajando es especial ya que sólo afecta a parte relacionada con el empaquetado de `bumprace`. Si hubiera un problema en el código fuente sería útil comprobar también el registro de errores aguas arriba. Desafortunadamente este es a menudo diferente para cada paquete que mire, pero si lo busca en la web, en la mayoría de los casos debería encontrarlo fácilmente.

1.3.6 Ofrecer ayuda

Si encuentra un error abierto que no está asignado a nadie y está en situación de corregirlo, debería añadir un comentario con su solución. Asegúrese de incluir tanta información como sea posible: ¿bajo qué circunstancias ocurre el error? ¿cómo ha corregido el problema? ¿ha probado la solución?

Si no se ha rellenado ningún informe de error, puede crear uno. Lo que debe tener en cuenta es lo siguiente: ¿es la incidencia tan pequeña que simplemente pedir que alguien la confirme es suficiente? ¿ha podido únicamente corregir parcialmente la incidencia y desea al menos compartir su parte?

Es bueno poder ofrecer ayuda y con toda seguridad será apreciada.

1.3.7 Obtener el código

Once you know the source package to work on, you will want to get a copy of the code on your system, so that you can debug it. The ubuntu-dev-tools package has a tool called `pull-lp-source` that a developer can use to grab the source code for any package. For example, to grab the source code for the tomboy package in `xenial`, you can type this:

```
$ pull-lp-source bumprace xenial
```


If you do not specify a release such as `xenial`, it will automatically get the package from the development version.

Once you've got a local clone of the source package, you can investigate the bug, create a fix, generate a debdiff, and attach your debdiff to a bug report for other developers to review. We'll describe specifics in the next sections.

1.3.8 Trabajar en una solución

Se han escrito libros completos sobre cómo encontrar errores, cómo arreglarlos, cómo probarlos, etc. Si completamente novato en programación, intente primero encontrar errores simples como erratas obvias en los textos. Intente mantener tan reducidos como sea posible y documente claramente tanto los cambios como los supuestos que haga.

Antes de comenzar a trabajar en una solución, asegúrese de investigar si alguien más ya lo ha solucionado o está trabajando en una solución. Algunos buenos lugares para buscar son:

- El sistema de seguimiento de errores (abiertos y cerrados) del proyecto original (y de Debian)
- El historial de cambios del proyecto original (o una versión más reciente) podría haber corregido el problema.
- La subida de errores o paquetes de Debian u otras distribuciones

You may want to create a patch which includes the fix. The command `edit-patch` is a simple way to add a patch to a package. Run:

```
$ edit-patch 99-new-patch
```

Esto copiara el paquete a un directorio temporal. Ahora ya puede editar los archivos con un editor de textos o aplicar parches desde el proyecto original, por ejemplo:

```
$ patch -p1 < ../bugfix.patch
```

Completada la edición escriba `exit` o pulse `Ctrl+D` para abandonar el intérprete de órdenes temporal. El nuevo parche se habrá añadido a `debian/patches`.

You must then add a header to your patch containing meta information so that other developers can know the purpose of the patch and where it came from. To get the template header that you can edit to reflect what the patch does, type this:

```
$ quilt header --dep3 -e
```

This will open the template in a text editor. Follow the template and make sure to be thorough so you get all the details necessary to describe the patch.

In this specific case, if you just want to edit `debian/control`, you do not need a patch. Put `Homepage: http://www.linux-games.com/bumprace/` at the end of the first section and the bug should be fixed.

Documentar la solución

Es muy importante documentar con suficiente detalle los cambios que se hacen, de tal forma que los desarrolladores que revisen el código en el futuro no tengan que imaginar su razonamiento y sus supuestos cuando lo hizo. Cada paquete fuente de Debian y Ubuntu contiene un archivo `debian/changelog` donde se mantienen los cambios de cada paquete subido.

La manera más fácil de actualizar esto es ejecutar:

```
$ dch -i
```

Esto añadirá una entrada modelo al registro de cambios y lanzará un editor en el que poder rellenar los campos vacíos. Un ejemplo podría ser:


```
specialpackage (1.2-3ubuntu4) trusty; urgency=low

  * debian/control: updated description to include frobnicator (LP: #123456)

 -- Emma Adams <emma.adams@isp.com>  Sat, 17 Jul 2010 02:53:39 +0200
```

`dch` debería rellenar por usted la primera y última línea de la entrada en el registro de cambios. La línea 1 contiene el nombre del paquete fuente, la número de versión, la emisión de Ubuntu a la que va dirigida, la urgencia (que casi en todos los casos es «low», baja). La última línea siempre contiene el nombre, la dirección de correo y la fecha y hora (en formato **RFC 5322**) del cambio.

Realizado todo esto, céntrese en la propia entrada del registro de cambios: es muy importante documentar:

1. Where the change was done.
2. What was changed.
3. Where the discussion of the change happened.

En este (muy sencillo) ejemplo el último punto se cubre por `(LP: #123456)` que se refiere al reporte 123456 en Launchpad. Los reportes de errores, discusiones en listas de correos y documentos de especificaciones son buenas fuentes de información que justifican las razones por las cuales se hacen los cambios. Como ventaja adicional, si se usa la notación `LP: #<número>` para de los errores de Launchpad, dichos errores se cerrarán automáticamente cuando los cambios se suban a Ubuntu.

In order to get it sponsored in the next section, you need to file a bug report in Launchpad (if there isn't one already, if there is, use that) and explain why your fix should be included in Ubuntu. For example, for tomboy, you would file a bug here (edit the URL to reflect the package you have a fix for). Once a bug is filed explaining your changes, put that bug number in the changelog.

1.3.9 Probar la solución

Para crear un paquete de prueba con los cambios, ejecute estas órdenes:

```
$ debuild -S -d -us -uc
$ pbuilder-dist <release> build ../<package>_<version>.dsc
```

This will create a source package from the branch contents (`-us -uc` will just omit the step to sign the source package and `-d` will skip the step where it checks for build dependencies, pbuilder will take care of that) and `pbuilder-dist` will build the package from source for whatever `release` you choose.

Nota: If `debuild` errors out with "Version number suggests Ubuntu changes, but Maintainer: does not have Ubuntu address" then run the `update-maintainer` command (from ubuntu-dev-tools) and it will automatically fix this for you. This happens because in Ubuntu, all Ubuntu Developers are responsible for all Ubuntu packages, while in Debian, packages have maintainers.

In this case with bumprace, run this to view the package information:

```
$ dpkg -I ~/pbuilder/*_result/bumprace_*.deb
```

As expected, there should now be a `Homepage:` field.

Nota: En muchos casos tendrá que instalar de verdad el paquete para asegurarse de que funciona como se espera. En nuestro caso es mucho más fácil. Si la compilación tuvo éxito, encontrará los paquetes binarios en `~/pbuilder/<release>_result`. Instálelos mediante `sudo dpkg -i <package>.deb` o haciendo doble clic sobre ellos en el administrador de archivos.

1.3.10 Submitting the fix and getting it included

With the changelog entry written and saved, run `debuild` one more time:

```
$ debuild -S -d
```

and this time it will be signed and you are now ready to get your diff to submit to get sponsored.

In a lot of cases, Debian would probably like to have the patch as well (doing this is best practice to make sure a wider audience gets the fix). So, you should submit the patch to Debian, and you can do that by simply running this:

```
$ submittodebian
```

Esto le llevará por una serie de pasos para asegurarse de que el error termina en el lugar adecuado. Asegúrese de revisar de nuevo el archivo de differencias (diff) para tener certeza de que no incluye cambios aleatorios que haya realizado anteriormente.

La comunicación es importante, así que cuando añada algo más de descripción a la petición de inclusión, sea amigable y explíquelo bien.

Si todo ha ido bien debería recibir un correo desde el sistema de registro de errores de Debian con más información. Esto podría llevar unos minutos.

It might be beneficial to just get it included in Debian and have it flow down to Ubuntu, in which case you would not follow the below process. But, sometimes in the case of security updates and updates for stable releases, the fix is already in Debian (or ignored for some reason) and you would follow the below process. If you are doing such updates, please read our *Security and stable release updates* article. Other cases where it is acceptable to wait to submit patches to Debian are Ubuntu-only packages not building correctly, or Ubuntu-specific problems in general.

But if you're going to submit your fix to Ubuntu, now it's time to generate a "debdiff", which shows the difference between two Debian packages. The name of the command used to generate one is also `debdiff`. It is part of the `devscripts` package. See `man debdiff` for all the details. To compare two source packages, pass the two dsc files as arguments:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc
```

In this case, `debdiff` the dsc you downloaded with `pull-lp-source` and the new dsc file you generated. This will generate a patch that your sponsor can then apply locally (by using `patch -p1 < /path/to/debdiff`). In this case, pipe the output of the debdiff command to a file that you can then attach to the bug report:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc > 1-1.0-1ubuntu1.debdiff
```

The format shown in `1-1.0-1ubuntu1.debdiff` shows:

1. `1-` tells the sponsor that this is the first revision of your patch. Nobody is perfect, and sometimes follow-up patches need to be provided. This makes sure that if your patch needs work, that you can keep a consistent naming scheme.
2. `1.0-1ubuntu1` shows the new version being used. This makes it easy to see what the new version is.
3. `.debdiff` is an extension that makes it clear that it is a debdiff.

While this format is optional, it works well and you can use this.

Next, go to the bug report, make sure you are logged into Launchpad, and click "Add attachment or patch" under where you would add a new comment. Attach the debdiff, and leave a comment telling your sponsor how this patch can be applied and the testing you have done. An example comment can be:

```
This is a debdiff for Artful applicable to 1.0-1. I built this in pbuilder
and it builds successfully, and I installed it, the patch works as intended.
```


Make sure you mark it as a patch (the Ubuntu Sponsors team will automatically be subscribed) and that you are subscribed to the bug report. You will then receive a review anywhere between several housr from submitting the patch to several weeks. If it takes longer than that, please join `#ubuntu-motu` on `freenode` and mention it there. Stick around until you get an answer from someone, and they can guide you as to what to do next.

Once you have received a review, your patch was either uploaded, your patch needs work, or is rejected for some other reason (possibly the fix is not fit for Ubuntu or should go to Debian instead). If your patch needs work, follow the same steps and submit a follow-up patch on the bug report, otherwise submit to Debian as shown above.

Remember: good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on freenode. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

1.3.11 Consideraciones adicionales:

Si encuentra un paquete y existen un par de cosas triviales que puede corregir al mismo tiempo, hágalo. Esto acelerará su revisión e inclusión.

Si hay varias cosas importantes que quiera arreglar, sería recomendable enviar parches individuales o peticiones de integración. Si se han rellenado errores independientes para las incidencias, lo hace incluso más fácil.

1.4 Empaquetando nuevo software

A pesar de que hay miles de paquetes en el repositorio de Ubuntu, todavía hay muchos que nadie ha conseguido. Si hay una nueva y excitante porción de software que siente que necesita una exposición más amplia, quizá quiera intentar crear un paquete para Ubuntu o un PPA. Esta guía le acompañará a través de los pasos del empaquetado de nuevo software.

Probablemente quiera leer el artículo *Getting Set Up* antes para preparar su entorno de desarrollo.

1.4.1 Comprobar el programa

La primera fase del empaquetado es obtener el archivo tar liberado aguas arriba (llamamos a los autores de las aplicación aguas arriba, «upstream») y comprobar que compila y se ejecuta.

Esta guía le llevará a través del empaquetado de una aplicación sencilla llamada GNU Hello, la cual ha sido publicada en GNU.org.

Download GNU Hello:

```
$ wget -O hello-2.10.tar.gz "http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz"
```

Now uncompress it:

```
$ tar xf hello-2.10.tar.gz
$ cd hello-2.10
```

Esta aplicación usa el sistema de compilación autoconf, así que debemos ejecutar «./configure» para preparar la compilación.

Esto comprobará las dependencias de compilación necesarias. Como `hello` es un ejemplo sencillo, `build-essential` debería proporcionar todo lo que necesitamos. Para programas más complejos, la orden fallará si no tiene las bibliotecas y archivos de desarrollo necesarios. Instale los paquetes necesarios y repita hasta que la orden se ejecute correctamente.:


```
$ ./configure
```

Ahora puede compilar el fuente:

```
$ make
```

Si la compilación finaliza con éxito puede instalar y ejecutar el programa:

```
$ sudo make install
$ hello
```

1.4.2 Iniciar un paquete

`bzr-builddeb` includes a plugin to create a new package from a template. The plugin is a wrapper around the `dh_make` command. Run the command providing the package name, version number, and path to the upstream tarball:

```
$ sudo apt-get install dh-make bzr-builddeb
$ cd ..
$ bzr dh-make hello 2.10 hello-2.10.tar.gz
```

Cuando pregunte por el tipo de paquete escriba `s` para binario simple. Esto importará el código en u n rama y añadirá el directorio de empaquetado `debian/`. Eche un vistazo al contenido. La mayoría de los archivos que añade son sólo necesarios para paquetes especialistas (como los módulos de Emacs) así que puede comenzar por eliminar los archivos de ejemplo opcionales:

```
$ cd hello/debian
$ rm *ex *EX
```

Ahora debería personalizar cada uno de los archivos.

In `debian/changelog` change the version number to an Ubuntu version: `2.10-0ubuntu1` (upstream version 2.10, Debian version 0, Ubuntu version 1). Also change `unstable` to the current development Ubuntu release such as `trusty`.

Mucho del trabajo de compilación de paquetes se hace mediante una serie de scripts llamados `debhelper`. El comportamiento exacto de `debhelper` cambia con las nuevas versiones mayores y el archivo de compatibilidad instruye a `debhelper` sobre la versión como la que debe comportarse. Generalmente querrá establecerla a la versión más reciente, que es la `9`.

`control` contiene todos los metadatos del paquete. El primer párrafo describe el paquete fuente. El segundo y siguientes describen los paquetes binarios a construir. Necesitaremos añadir los paquetes necesarios para compilar la aplicación a `Build-Depends:`. Para `hello`, asegúrese de que incluye al menos:

```
Build-Depends: debhelper (>= 9)
```

También necesitará rellenar la descripción del programa en el campo `Description:`.

Debe rellenar `copyright` para que siga la licencia de los fuentes aguas arriba. Según el archivo hello/COPYING es GNU GPL 3 o superior.

`docs` contiene cualquier archivo de documentación de aguas arriba que piense que debería ser incluido en el paquete final.

`README.source` y `README.Debian` solo son necesarios si su paquete tiene características no estándar. No las tiene, así que se pueden borrar.

`source/format` se puede dejar como está. Describe el formato de la versión del paquete fuente y debería ser `3.0 (quilt)`.

`rules` es el archivo más complejo. Es un Makefile que compila el código y lo convierte en un paquete binario. Afortunadamente la mayor parte del trabajo hoy en día se realiza automáticamente por `debhelper 7` así que el objetivo Makefile `%` universal simplemente ejecuta el script `dh` que a su vez ejecutará todo lo que haga falta.

Todos estos archivos se explican con más detalle en el artículo *resumen del directorio debian*.

Finalmente confirme el código en su rama de empaquetado:

```
$ bzr add debian/source/format
$ bzr commit -m "Initial commit of Debian packaging."
```

1.4.3 Construir el paquete

Ahora necesitamos comprobar que el empaquetado compila con éxito el paquete y genera el paquete binario .deb:

```
$ bzr builddeb -- -us -uc
$ cd ../../
```

`bzr builddeb` es una orden para compilar el paquete en su ubicación actual. Los parámetros `-us -uc` le indicarán que no es necesario firmar con GPG el paquete. El resultado se dejará en `...`

Puede ver el contenido del paquete con:

```
$ lesspipe hello_2.10-0ubuntu1_amd64.deb
```

Install the package and check it works (later you will be able to uninstall it using `sudo apt-get remove hello` if you want):

```
$ sudo dpkg --install hello_2.10-0ubuntu1_amd64.deb
```

You can also install all packages at once using:

```
$ sudo debi
```

1.4.4 Siguientes pasos

Even if it builds the .deb binary package, your packaging may have bugs. Many errors can be automatically detected by our tool `lintian` which can be run on the source .dsc metadata file, .deb binary packages or .changes file:

```
$ lintian hello_2.10-0ubuntu1.dsc
$ lintian hello_2.10-0ubuntu1_amd64.deb
```

To see verbose description of the problems use `--info` lintian flag or `lintian-info` command.

For Python packages, there is also a `lintian4python` tool that provides some additional lintian checks.

Después de realizar una corrección al empaquetado puede reconstruirlo usando la opción `-nc` («no clean», sin limpiar) para no tener que compilarlo desde el principio:

```
$ bzr builddeb -- -nc -us -uc
```

Después de comprobar que el paquete se compila en local debería asegurarse de que lo hace también en un sistema limpio usando `pbuilder`. Puesto que se va a subir en breve a un PPA (archivo de paquetes personal), esta proceso de subida deberá ser *firmado* para permitir a Launchpad que verifique que la carga proviene de usted (puede saber que la carga se firmará porque no se pasan los marcadores `-us` y `-uc` a `bzr builddeb` como se hizo antes). Para firmar su trabajo necesita haber configurado GPG. Si no ha configurado todavía `pbuilder-dist` o GPG todavía, *do so now*:


```
$ bzr builddeb -S
$ cd ../build-area
$ pbuilder-dist trusty build hello_2.10-0ubuntu1.dsc
```

Cuando esté satisfecho con su paquete deseará que otros lo revisen. Puede subirlo la rama a Launchpad para su revisión:

```
$ bzr push lp:~<lp-username>/+junk/hello-package
```

Al subirlo a un PPA se asegurará de que compila y le proporcionará una manera fácil de probar los paquetes binarios para usted y para otros. Necesitará configurar un PPA en Launchpad y luego cargarlo con `dput`:

```
$ dput ppa:<lp-username>/<ppa-name> hello_2.10-0ubuntu1.changes
```

You can ask for reviews in `#ubuntu-motu` IRC channel, or on the MOTU mailing list. There might also be a more specific team you could ask such as the GNU team for more specific questions.

1.4.5 Enviar para su inclusión

Existe varios caminos por los que un paquete puede entrar en Ubuntu. En la mayoría de los casos, ir antes por Debian puede ser la mejor alternativa. De esta forma se asegura de que su paquete llegará al mayor número de usuarios, ya que también estará disponible no solo para Debian y Ubuntu, sino para todos sus derivados también. Estos son algunos enlaces útiles para enviar nuevos paquetes a Debian:

- Debian Mentors FAQ - debian-mentors is for the mentoring of new and prospective Debian Developers. It is where you can find a sponsor to upload your package to the archive.
- Work-Needing and Prospective Packages - Information on how to file “Intent to Package” and “Request for Package” bugs as well as list of open ITPs and RFPs.
- Debian Developer’s Reference, 5.1. New packages - The entire document is invaluable for both Ubuntu and Debian packagers. This section documents processes for submitting new packages.

In some cases, it might make sense to go directly into Ubuntu first. For instance, Debian might be in a freeze making it unlikely that your package will make it into Ubuntu in time for the next release. This process is documented on the “New Packages” section of the Ubuntu wiki.

1.4.6 Screenshots

Una vez que se ha cargado el paquete a Debian, se recomienda añadir capturas de pantalla para que el usuario pueda ver el aspecto del programa. Estas imágenes pueden cargarse en http://screenshots.debian.net/upload .

1.5 Actualizaciones de seguridad y de versiones estables

1.5.1 Corregir un error de seguridad en Ubuntu

Introducción

Fixing security bugs in Ubuntu is not really any different than *fixing a regular bug in Ubuntu*, and it is assumed that you are familiar with patching normal bugs. To demonstrate where things are different, we will be updating the dbus package in Ubuntu 12.04 LTS (Precise Pangolin) for a security update.

Obtener el código fuente

In this example, we already know we want to fix the dbus package in Ubuntu 12.04 LTS (Precise Pangolin). So first you need to determine the version of the package you want to download. We can use the `rmadison` to help with this:

```
$ rmadison dbus | grep precise
dbus | 1.4.18-1ubuntu1   | precise          | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-security | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-updates  | source, amd64, armel, armhf, i386, powerpc
```

Typically you will want to choose the highest version for the release you want to patch that is not in -proposed or -backports. Since we are updating Precise's dbus, you'll download 1.4.18-1ubuntu1.4 from precise-updates:

```
$ bzr branch ubuntu:precise-updates/dbus
```

Parchear el código fuente

Now that we have the source package, we need to patch it to fix the vulnerability. You may use whatever patch method that is appropriate for the package, but this example will use `edit-patch` (from the ubuntu-dev-tools package). `edit-patch` is the easiest way to patch packages and it is basically a wrapper around every other patch system you can imagine.

Para crear un parche usando `edit-patch`:

```
$ cd dbus
$ edit-patch 99-fix-a-vulnerability
```

Esto aplicará los parches existentes y dejará el empaquetado en un directorio temporal. Ahora edite los archivos necesarios para solucionar la vulnerabilidad. Frecuentemente se habrá proporcionado un parche desde aguas arriba de forma que pueda aplicarlo:

```
$ patch -p1 < /home/user/dbus-vulnerability.diff
```

Después de hacer los cambios necesarios, simplemente pulse Ctrl+D o escriba «exit» para dejar el intérprete de órdenes temporal.

Formatear el registro de cambios («changelog») y los parches

After applying your patches you will want to update the changelog. The `dch` command is used to edit the `debian/changelog` file and `edit-patch` will launch `dch` automatically after un-applying all the patches. If you are not using `edit-patch`, you can launch `dch -i` manually. Unlike with regular patches, you should use the following format (note the distribution name uses precise-security since this is a security update for Precise) for security updates:

```
dbus (1.4.18-2ubuntu1.5) precise-security; urgency=low

  * SECURITY UPDATE: [DESCRIBE VULNERABILITY HERE]
    - debian/patches/99-fix-a-vulnerability.patch: [DESCRIBE CHANGES HERE]
    - [CVE IDENTIFIER]
    - [LINK TO UPSTREAM BUG OR SECURITY NOTICE]
    - LP: #[BUG NUMBER]
...
```

Actualice su parche para usar las etiquetas de parche adecuadas. Su parche debería tener por lo menos las etiquetas «Origin», «Description» y «Bug-Ubuntu». Por ejemplo, edite debian/patches/99-fix-a-vulnerability.patch para que tenga es siguiente aspecto:


```
## Description: [DESCRIBE VULNERABILITY HERE]
## Origin/Author: [COMMIT ID, URL OR EMAIL ADDRESS OF AUTHOR]
## Bug: [UPSTREAM BUG URL]
## Bug-Ubuntu: https://launchpad.net/bugs/[BUG NUMBER]
Index: dbus-1.4.18/dbus/dbus-marshal-validate.c
...
```

Muchas vulnerabilidades se puede solucionar en la misma carga de seguridad, pero asegúrese de usar parches distintos para las diferentes vulnerabilidades.

Probar y enviar el trabajo

En este punto el proceso es el mismo que para *arreglar un bug normal de Ubuntu*. Más concretamente, deseará:

1. Construir el paquete y comprobar que compila sin errores y sin ningún aviso del compilador añadido.
2. Actualizar a la nueva versión del paquete desde la versión anterior
3. Probar que el nuevo paquete corrige la vulnerabilidad y no introduce ninguna regresión
4. Enviar el trabajo mediante una propuesta de integración de Launchpad y rellenar un error en Launchpad asegurándose de marcar el error como un fallo de seguridad y de suscribirse a `ubuntu-security-sponsors`.

Si la vulnerabilidad de seguridad no es pública todavía no presente una propuesta de integración y asegúrese de marcar el error como privado.

El presentado debe incluir un caso de prueba, es decir, un comentario que indique claramente cómo recrear el error ejecutando la versión antigua y luego cómo asegurarse de que el error no existe en la nueva versión.

The bug report should also confirm that the issue is fixed in Ubuntu versions newer than the one with the proposed fix (in the above example newer than Precise). If the issue is not fixed in newer Ubuntu versions you should prepare updates for those versions too.

1.5.2 Actualizaciones de versiones estables

También se permite actualizaciones de emisiones en las que un paquete tiene un error de gran impacto, como por ejemplo una regresión severa de una emisión anterior o un error que podría causar pérdida de datos. Debido al potencia de que esas actualizaciones a su vez introduzcan errores solo se permiten cuando los cambios pueden ser fácilmente comprendidos y verificados.

El proceso de actualizaciones de versiones estables es el mismo que el proceso para errores de seguridad excepto que debería suscribir `ubuntu-sru` al error.

The update will go into the `proposed` archive (for example `precise-proposed`) where it will need to be checked that it fixes the problem and does not introduce new problems. After a week without reported problems it can be moved to `updates`.

See the Stable Release Updates wiki page for more information.

1.6 Parches a los paquetes

A veces, los mantenedores de paquetes de Ubuntu tienen que cambiar el código fuente recibido desde aguas arriba para que funcione correctamente en Ubuntu. Ejemplos de esto incluyen parches aguas arriba que no han llegado todavía a la versión emitida, o cambios al sistema de compilación aguas arribas que son necesarios únicamente para compilarlos en Ubuntu. Se podría cambiar el código recibido de aguas arriba directamente, pero al hacerlo que dificulta la eliminación de los parches posteriormente cuando se hayan incorporado los cambios aguas arriba, o extraer el cambio para enviarlo

al proyecto aguas arriba. En su lugar, se mantienen los cambios como parches independientes, en forma de archivos de diferencias (diff).

Hay varias formas diferentes de gestionar los parches de paquetes de Debian, aunque afortunadamente se está estandarizando en un único sistema, Quilt, que se usa ya para la mayoría de los paquetes.

Let's look at an example package, `kamoso` in Trusty:

```
$ bzr branch ubuntu:trusty/kamoso
```

Los parches se mantienen en `debian/patches`. Este paquete tiene un parche `kubuntu_01_fix_qmax_on_armel.diff` para arreglar un fallo de compilación en ARM. Al parche se le ha dado un nombre descriptivo de lo que hace, un número para mantener los parches ordenados (dos parches se puede solapar si cambian el mismo archivo) y en este caso el equipo de Kubuntu añade su propio prefijo para mostrar que el parche proviene de ellos en lugar de venir de Debian.

El orden de los parches a aplicar se mantiene en `debian/patches/series`.

1.6.1 Parches con Quilt

Antes de comenzar a trabajar con Quilt necesita indicarle dónde encontrar los parches. Añádalo a su archivo `~/.bashrc`:

```
export QUILT_PATCHES=debian/patches
```

Y como fuente el archivo para aplicar la nueva exportación:

```
$ . ~/.bashrc
```

Por defecto todos los parches ya están aplicados a las extracción UDD o a los paquetes descargados. Puede comprobarlo con:

```
$ quilt applied
kubuntu_01_fix_qmax_on_armel.diff
```

Si quería eliminar el parche debería ejecutar `pop`:

```
$ quilt pop
Removing patch kubuntu_01_fix_qmax_on_armel.diff
Restoring src/kamoso.cpp

No patches applied
```

Y para aplicar un parche use `push`:

```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp

Now at patch kubuntu_01_fix_qmax_on_armel.diff
```

1.6.2 Añadir un nuevo parche

Para añadir un nuevo parche necesita indicarle a Quilt que cree un nuevo parche, qué archivos debería cambiar ese parche, editar los archivos y refrescar el parche:


```
$ quilt new kubuntu_02_program_description.diff
Patch kubuntu_02_program_description.diff is now on top
$ quilt add src/main.cpp
File src/main.cpp added to patch kubuntu_02_program_description.diff
$ sed -i "s,Webcam picture retriever,Webcam snapshot program,"
src/main.cpp
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

El paso `quilt add` es importante, si lo olvida los archivos no terminarán en el parche.

El cambio estará ahora en `debian/patches/kubuntu_02_program_description.diff` y en el archivo `series` se habrá añadido el parche. Ahora debería añadir el archivo al empaquetado:

```
$ bzr add debian/patches/kubuntu_02_program_description.diff
$ bzr add .pc/*
$ dch -i "Add patch kubuntu_02_program_description.diff to improve the program description"
$ bzr commit
```

Quilt mantiene sus metadatos en el directorio `.pc/`, así que actualmente necesita añadirlo también al empaquetado. Esto debería mejorarse en el futuro.

Como regla general debería tener cuidado al añadir parches a programas a menos que lleguen desde aguas arriba, frecuentemente hay una buena razón para que el cambio no se haya hecho todavía. El ejemplo anterior cambia una cadena de la interfaz de usuario por ejemplo, así que invalidaría todas las traducciones. Si duda, pregunte al autor aguas arriba antes de añadir el parche.

1.6.3 Cabeceras de parche

Le recomendamos que etiquete cada parche con cabeceras DEP-3 añadiéndolas al comienzo de cada archivo de parche. Estas son algunas de las cabeceras que puede usar:

Description Description of what the patch does. It is formatted like `Description` field in `debian/control`: first line is short description, starting with lowercase letter, the next lines are long description, indented with a space.

Author Quién escribió el parque (por ejemplo, «Juana Nadie <empaquetador@ejemplo.com>").

Origin De dónde proviene el parche (por ejemplo, «upstream», aguas arriba), cuando no se indica el autor (*Author*).

Bug-Ubuntu Un enlace al error en Launchpad, prefiriéndose una forma corta (como *https://bugs.launchpad.net/bugs/XXXXXXX*). Si existen tambien errores aguas arriba o , añadir cabeceras *Bug* o *Bug-Debian*.

Forwarded Si el parche se reenvió aguas arriba. Uno de «yes» (sí), «no» (no) o «not-needed» (no necesario).

Last-Update Fecha de la última revisión (en formato «YYYY-MM-DD»).

1.6.4 Actualizar a nuevas versiones de aguas arriba

Para actualizar a la nueva versión, puede usar la orden `bzr merge-upstream`:

```
$ bzr merge-upstream --version 2.0.2 https://launchpad.net/ubuntu/+archive/primary/+files/kamoso_2.0
```


Cuando ejecute esa orden, se des-aplicarán todos los parches, porque pueden quedarse obsoletos. Podría ser necesario refrescarlos para que se adapten al nuevo código fuentes aguas arriba o deban ser eliminados completamente. Para comprobar posibles problemas, aplique los parches uno a uno:

```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp
Hunk #1 FAILED at 398.
1 out of 1 hunk FAILED -- rejects in file src/kamoso.cpp
Patch kubuntu_01_fix_qmax_on_armel.diff can be reverse-applied
```

Si puede ser aplicado a la inversa significa que el parche ya ha sido aplicado aguas arriba, de forma que ya se puede borrar:

```
$ quilt delete kubuntu_01_fix_qmax_on_armel
Removed patch kubuntu_01_fix_qmax_on_armel.diff
```

Luego continúe:

```
$ quilt push
Applied kubuntu_02_program_description.diff
```

Es una buena idea hacer un refresco, lo que actualizará el parche relativo a los fuentes cambiados aguas arriba:

```
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

Luego confirme como siempre:

```
$ bzr commit -m "new upstream version"
```

1.6.5 Hacer que un paquete use Quilt

Los paquetes modernos usan Quilt de forma predeterminada, están incluido en el formato de empaquetado. Compruebe `debian/source/format` para asegurarse que pone `3.0 (quilt)`.

Los paquetes más antiguos que usen el formato fuente 1.0 necesitarán usar Quilt explícitamente, normalmente incluyendo un archivo makefile en `debian/rules`.

1.6.6 Configurando Quilt

Puede usar el archivo `~/.quiltrc` para configurar quilt. Estas son algunas opciones que pueden resultar útiles para usar quilt con paquetes debian:

```
# Set the patches directory
QUILT_PATCHES="debian/patches"
# Remove all useless formatting from the patches
QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
# The same for quilt diff command, and use colored output
QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
```

1.6.7 Otros sistemas de parches

Otros sistemas de parches que se usan por los paquetes incluyen `dpatch` y `cdbs simple-patchsys`, los cuales funcionan de forma similar a Quilt, manteniendo los parches en `debian/patches`, pero emplean órdenes distintas

para aplicar, des-aplicar o crear parches. Puede averiguar qué sistema de parches se usa por un paquete mediante la orden `what-patch` (del paquete `ubuntu-dev-tools`). Puede usar `edit-patch`, como se ha mostrado en *capítulos previos*, como una manera fiable de trabajar con todos los sistemas.

En paquetes todavía más antiguos los cambios se incluirán directamente en los fuentes y los mantendrán el archivo fuente `diff.gz`. Esto complica la actualización a nuevas versiones de aguas arribas o distinguir entre parches por lo que lo mejor es evitarlo.

No cambie el sistema de parches de un paquete sin haberlo discutido con el mantenedor de Debian o el equipo pertinente de Ubuntu. Si no existe un sistema parches no tenga problemas en añadir Quilt.

1.7 Corrección de paquetes que no compilan a partir del código fuente (FTBFS)

Antes de que un paquete se pueda usar en Ubuntu, tiene que compilarse a partir del código fuente. Si esto falla, probablemente espere en -proposed (propuesto) y no estará disponible en los repositorios de Ubuntu. Puede encontrar una lista completa de los paquetes para los que falla la compilación desde el código fuente en http://qa.ubuntuwire.org/ftbfs/. Se muestran cinco categorías principales en esta página:

- Paquetes que fallan la compilación (F): algo falló en el proceso de compilación.
- Compilaciones canceladas (X): la compilación se ha cancelado por algún motivo. Estas probablemente deberían evitarse para empezar.
- El paquete está esperando a otro paquete (M): este paquete está esperando a que otro paquete se compile, se actualice o (si el paquete está en «main») una de sus dependencias está en la parte equivocada del repositorio.
- Fallo en el «chroot» (C): la parte del «chroot» ha fallado, lo que muy probablemente se arregle mediante una recompilación. Pida a un desarrollador que recompile el paquete y esto debería solucionarlo.
- Falló al subirlo (U) el paquete no se pudo subir. Generalmente es simplemente un caso en el que pedir una recompilación, pero compruebe antes el registro de la compilación.

1.7.1 Primeros pasos

Lo primero que querrá hacer es ver si puede reproducir el problema de compilación por usted mismo. Obtenga el código ejecutando `bzr branch lp:ubuntu/PAQUETE` y después obteniendo el tarball o ejecute *dget PAQUETE_DSC‘* sobre el archivo .dsc de la página de launchpad. Una vez lo tenga, compílelo en un «schroot».

Debería ser capaz de reproducir el error de compilación. Si no es así, compruebe si la compilación está descargando una dependencia faltante, lo que quiere decir que simplemente necesita convertirla en una dependencia de compilación en debian/control. Compilar el paquete en local también puede ayudar a saber si el problema está causado por una dependencia que falta, no listada (se compila localmente, pero falla en un «schroot»).

1.7.2 Comprobando Debian

Una vez que haya reproducido la incidencia, será el momento de buscar una solución. Si el paquete está también en Debian, puede comprobar si compila allí yendo a http://packages.qa.debian.org/PAQUETE. Si Debian cuenta con una versión más moderna, debería fusionarla. Si no es así, compruebe los registros de compilación y los errores enlazados desde esa página para conseguir información adicional sobre los problemas de compilación o parches. Debian también mantiene una lista de órdenes que fallaron al compilar y cómo corregirlas, la cual se puede encontrar en https://wiki.debian.org/qa.debian.org/FTBFS, donde podrá buscar soluciones.

1.7.3 Otras causas para para que un paquete falle para compilarse desde el código fuente (FTBFS).

Si un paquete está en «main» y le falta una dependencia que no está en «main», tendrá que rellenar un error de MIR. En https://wiki.ubuntu.com/MainInclusionProcess se explica el procedimiento para hacerlo.

1.7.4 Corregir la incidencia

Una vez que haya encontrado una solución al problema, siga el mismo proceso para cualquier otro error. Cree un parche, añádalo a una rama bzr o a un error, suscríbase a ubuntu-sponsors y luego intente que lo incluyan agua arriba o en Debian.

1.8 Bibliotecas compartidas

Las bibliotecas compartidas es código compilado que se supone que será compartido por diferentes programas. Se distribuyen como archivos `.so` en `/usr/lib/`.

Una biblioteca exporta símbolos que son versiones compiladas de funciones, clases y variables. Una biblioteca tiene un nombre denominado SONAME que incluye un número de versión. Esta versión de SONAME no tiene por qué coincidir con el número de versión de emisión pública. Un programa se compila contra una versión determinada de SONAME de la biblioteca. Si cualquiera de los símbolos es eliminado o modificado, entonces el número de versión debe ser cambiado lo que fuerza que que cualquier paquete que use la biblioteca sea recompilado contra la nueva versión. Los números de versiones normalmente son establecidos aguas arriba y los mantenemos en los nombres de los paquetes binarios, llamado número ABI, pero en ocasiones aguas arriba no usan números de versiones razonables y los empaquetadores tienen que mantener números de versiones separados.

Las bibliotecas se distribuyen normalmente aguas arriba como emisiones independientes. Algunas veces se distribuyen como partes de un programa. En este caso se pueden incluir en el paquete binario junto con el programa (esto se denomina «bundling», atado) si no espera que ningún otro programa use la biblioteca, pero lo más frecuente es que sean separadas en paquetes binarios independientes.

Las propias bibliotecas se meten en un paquete binario llamado `libfoo1` donde `foo` es el nombre de la biblioteca y `1` es la versión del SONAME. Los archivos de desarrollo del paquete, como los archivos de cabecera, necesitan compilar programas contra la biblioteca que se ha metido en el paquete llamado `libfoo-dev`.

1.8.1 Un ejemplo

Usaremos libnova como un ejemplo:

```
$ bzr branch ubuntu:trusty/libnova
$ sudo apt-get install libnova-dev
```

Para encontrar el SONAME de una biblioteca ejecute:

```
$ readelf -a /usr/lib/libnova-0.12.so.2 | grep SONAME
```

El SONAME es `libnova-0.12.so.2`, lo que coincide con el nombre del archivo (es lo normal, pero no siempre ocurre). En este caso aguas arriba han puesto el número de versión como parte del SONAME y le han dado una versión ABI de `2`. Los nombres de paquetes de bibliotecas deberían seguir el SONAME de la biblioteca que contienen. El paquete binario de la biblioteca se llama `libnova--0.12-2`, donde `libnova-0.12` es el nombre de la biblioteca y `2` es nuestro número ABI.

SI aguas arriban hacen cambios incompatible a su biblioteca tendrá que revisionar su SONAME y nosotros tendremos que cambiar de nombre a nuestra biblioteca. Cualquier otro paquete que use nuestra biblioteca deberá ser recompilado contra la nueva versión, esto es lo que se llama una transición y puede suponer cierto esfuerzo. Afortunadamente nuestro número ABI seguirá coincidiendo con el SONAME de aguas arriba, pero en ocasiones se introducen incompatibilidades sin cambiar el número de versión lo que nos fuerza a hacer cambios en el nuestro.

Si miramos en debian/libnova-0.12-2.install vemos que incluye estos dos archivos:

```
usr/lib/libnova-0.12.so.2
usr/lib/libnova-0.12.so.2.0.0
```

El último es la biblioteca en sí, completada con el número de menor de la versión y punto. El primero es un enlace simbólico que apunta a la biblioteca real. El enlace simbólico es lo que buscarán los programas que empleen la biblioteca, ya que los programas que se ejecutan no se preocupan por el número de menor de la versión.

`libnova-dev.install` incluye todo los archivos necesarios para compilar un programa con esta biblioteca. Los archivo de cabeceras, una configuración binaria, el archivo `.la` de libtool y `libnova.so` que es otro enlace simbólico apuntando a la misma biblioteca, programas compilados contra la biblioteca no se preocupan del número mayor de la versión (aunque el binario en el que se compilarán sí lo hará).

`.la` libtool files are needed on some non-Linux systems with poor library support but usually cause more problems than they solve on Debian systems. It is a current Debian goal to remove .la files and we should help with this.

1.8.2 Bibliotecas estáticas

El paquete -dev también incluye `usr/lib/libnova.a`. Es una biblioteca estática, una alternativa a las librerías compartidas. Cualquier programa compilado contra una biblioteca estática incluya el directorio de código dentro de sí mismo. Esto evita tener que preocuparse sobre la compatibilidad binaria de la biblioteca. Sin embargo, también significa que cualquier error, incluyendo las incidencias de seguridad, no serán actualizados junto con la biblioteca hasta que se recompile el programa. Por esta razón se desaconseja el uso de programas que usen librerías estáticas.

1.8.3 Archivos de símbolos

Cuando un paquete se compila contra una biblioteca el mecanismo de `shlibs` añadirá una dependencia del paquete a esa biblioteca. Este es el motivo de que la mayoría de los programas tengan `Depends: ${shlibs:Depends}` en el archivo `debian/control`. Eso se sustituye por las dependencias de la biblioteca en tiempo de compilación. Sin embargo, shlibs solo puede hacerla depender del número mayor de versión ABI, `2` en el ejemplo, así que si se añaden nuevos símbolos a libnova 2.1 un programa que use estos símbolos podría todavía seguir instalado contra libnova ABI 2.0, lo que podría producir un fallo.

Para hacer más precisas las dependencias de las bibliotecas matenemos archivos `.symbols` que listan todos los símbolos de una librería y la versión en la que aparecen.

libnova no tiene archivo de símbolos así que podemos crear uno. Comience por compilar el paquete:

```
$ bzr builddeb -- -nc
```

La opción `-nc` hará que se finalice al completar la compilación sin que se eliminen los archivos compilados. Cámbiese al directorio compilado y ejecute `dpkg-gensymbols` para el paquete de la biblioteca:

```
$ cd ../build-area/libnova-0.12.2/
$ dpkg-gensymbols -plibnova-0.12-2 > symbols.diff
```

Esto genera un archivo de diferencias (diff) que puede aplicar:

```
$ patch -p0 < symbols.diff
```


Lo que creará un archivo con un nombre similar a `dpkg-gensymbolsnY_WWI` que lista todos los símbolos. También lista la versión de paquete actual. Podemos eliminar la versión de empaquetado que se lista en el archivo de símbolos porque generalmente no se añaden nuevos símbolos por versiones más modernas de empaquetado, sino por los desarrolladores aguas arriba:

```
$ sed -i s,-0ubuntu2,, dpkg-gensymbolsnY_WWI
```

Ahora mueva el archivo a su ubicación, confirme y haga una prueba de compilación:

```
$ mv dpkg-gensymbolsnY_WWI ../../libnova/debian/libnova-0.12-2.symbols
$ cd ../../libnova
$ bzr add debian/libnova-0.12-2.symbols
$ bzr commit -m "add symbols file"
$ bzr builddeb
```

Si compila con éxito es que el archivo de símbolos es correcto. Con la siguiente versión aguas arriba de libnova podría ejecutar de nuevo dpkg-gensymbols y obtendrá un archivo de diferencias para actualizar el archivo de símbolos.

1.8.4 Archivos de símbolos de bibliotecas de C++

C++ has even more exacting standards of binary compatibility than C. The Debian Qt/KDE Team maintain some scripts to handle this, see their Working with symbols files page for how to use them.

1.8.5 Lecturas adicionales

Junichi Uekawa's Debian Library Packaging Guide goes into this topic in more detail.

1.9 Adaptar actualizaciones de software a versiones anteriores

Sometimes you might want to make new functionality available in a stable release which is not connected to a critical bug fix. For these scenarios you have two options: either you upload to a PPA or prepare a backport.

1.9.1 Archivo de paquetes personal («Personal Package Archive», PPA)

Usar un PPA tiene una serie de ventajas. Es bastante directo, no necesita la aprobación de nadie, pero la desventaja es que sus usuarios tendrán que activarlo manualmente. Es un origen de software no estándar.

The PPA documentation on Launchpad is fairly comprehensive and should get you up and running in no time.

1.9.2 Adaptaciones oficiales a versiones anteriores de Ubuntu

El proyecto Backports (adaptaciones a versiones antiguas) es un medio de proporcionar nuevas funcionalidades a los usuarios. Debido al riesgo inherente de afectar a la estabilidad al adaptar los paquetes a versiones antiguas, los usuarios no obtienen estos paquetes sin algún tipo de acción explícita por su parte. Esto generalmente convierte a las adaptaciones en camino poco adecuado para corregir errores. Si un paquete en una emisión de Ubuntu tiene un error, debería ser corregido mediante los procesos descritos en *Security Update or the Stable Release Update process*, según sea corresponda.

Una vez haya determinado que quiere adaptar un paquete a una versión estable, necesitará hacer una compilación de prueba y probarla sobre dicha versión estable. `pbuilder-dist` (del paquete `ubuntu-dev-tools`) es una herramienta muy práctica para hacerlo fácilmente.

Para reportar una petición de adaptación y hacer que el equipo de Backporters la procese, puede usar la herramienta `requestbackport` (también del paquete `ubuntu-dev-tools`). Esta herramienta determinará las emisiones intermedias a las que el paquete necesita ser adaptado, listará todas las dependencias inversas y rellenará la petición de adaptación. También incluirá una lista de comprobación en el error.

CAPÍTULO 2

Base de conocimiento

2.1 Comunicación en el desarrollo de Ubuntu

En un proyecto donde se modifican miles de líneas de código, se toman muchas decisiones y miles de personas interactúan a diario, es importante comunicarse eficazmente.

2.1.1 Listas de correo

Las listas de correo son una herramienta importante si quiere comunicar ideas a un equipo más grande y asegurarse de llegar a todos, incluso en distintas zonas horarias.

En términos de desarrollo, estos son los más importantes:

- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel-announce (Solo anuncios, los anuncios más importantes sobre desarrollo van aquí)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel (discusión general de desarrollo de Ubuntu)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-motu (Discusión del equipo MOTU, para obtener ayuda con el empaquetado)

2.1.2 Canales IRC

Para discusiones en tiempo real, conéctese a irc.freenode.net y únase a alguno de estos canales:

- #ubuntu-devel (para discusiones generales sobre desarrollo)
- #ubuntu-motu (para discusiones del equipo MOTU y solicitar ayuda generalmente)

2.2 Descripción general básica del Directorio `debian/`

En este artículo explicaremos brevemente los diferentes archivos que son importantes para el empaquetado de paquetes Ubuntu los cuales están en el directorio `debian/`. Los más importantes son `changelog`, `control`, `copyright` y `rules`. Estos son requeridos por todos los paquetes. Un conjunto de archivos adicionales en `debian/` puede que sean usados para personalizar y configurar el comportamiento del paquete. Sobre algunos de ello se habla en este articulo, pero esto no pretende ser una lista completa.

2.2.1 El registro de cambios

Este archivo es, como su nombre indica, un listado de los cambios realizados en cada versión. Tiene un formato especifico que da el nombre del paquete, versión, distribución, cambios, y quién realizó estos cambios en un determinado momento. Si tiene una clave GPG (vea: *Getting set up*), asegúrese de usar el mismo nombre y dirección de correo electrónico en `changelog`. tal y cómo lo tiene en la clave. Lo siguiente es una plantilla de `changelog`:

```
package (version) distribution; urgency=urgency

 * change details
  - more change details
 * even more change details

-- maintainer name <email address>[two spaces]  date
```

El formato (especialmente el de la fecha) es importante. La fecha debe estar en formato **RFC 5322**, que puede obtenerse mediante la orden `date -R`. Por conveniencia, la orden `dchx` se puede usar para editar el archivo changelog. Se actualizará la fecha de forma automática.

Los puntos de segundo nivel se indican con un guión «-», los de primer nivel usan un asterisco «*».

Si está empaquetando desde cero, `dch --create` (`dch` se encuentra en el paquete `devscripts`) creará un archivo `debian\changelog` estándar.

Este es un ejemplo del archivo `changelog` para hello:

```
hello (2.8-0ubuntu1) trusty; urgency=low

  * New upstream release with lots of bug fixes and feature improvements.

-- Jane Doe <packager@example.com>  Thu, 21 Oct 2013 11:12:00 -0400
```

Observe que el paquete tiene un `-0ubuntu1` anexado a el, este representa la versión en la distribución y se usa para que el paquete pueda ser actualizado (para solucionar errores por ejemplo) con nuevas subidas usando la misma versión principal con la que se liberó.

Ubuntu y Debian tienen esquemas de versión de paquetes ligeramente diferentes para evitar conflictos en los programas con la misma versión principal. Si una paquete de Debian sufre un cambio en Ubuntu, se le agrega el `ubuntuX` (donde `X` es el número de revisión en Ubuntu) al final de la versión de Debian. Si el paquete hello `2.6-1` de Debian es modificado en Ubuntu, la versión quedaría `2.6-1ubuntu1`. Si el paquete no existiera en Debian, entonces la revisión ahí sería `0` (y quedaría `2.6-0ubuntu1`).

For further information, see the changelog section (Section 4.4) of the Debian Policy Manual.

2.2.2 El archivo de control

El archivo `control` contiene la información que usan administradores de paquetes (como `apt-get`, `synaptic` y `adept`), dependencias de compilación, información del mantenedor y mucho más.

Para el paquete `hello` de Ubuntu, el archivo `control` tiene el siguiente aspecto:

```
Source: hello
Section: devel
Priority: optional
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XSBC-Original-Maintainer: Jane Doe <packager@example.com>
Standards-Version: 3.9.5
Build-Depends: debhelper (>= 7)
Vcs-Bzr: lp:ubuntu/hello
```



```
Homepage: http://www.gnu.org/software/hello/

Package: hello
Architecture: any
Depends: ${shlibs:Depends}
Description: The classic greeting, and a good example
 The GNU hello program produces a familiar, friendly greeting. It
 allows non-programmers to use a classic computer science tool which
 would otherwise be unavailable to them. Seriously, though: this is
 an example of how to do a Debian package. It is the Debian version of
 the GNU Project's 'hello world' program (which is itself an example
 for the GNU Project).
```

El primer párrafo describe el paquete fuente incluyendo la lista de paquetes requeridos para construirlo desde sus fuentes en el campo `Build-Depends`. También contiene alguna meta-información como el nombre del mantenedor, la versión de la política de Debian que cumple el paquete, la ubicación del repositorio de control de versiones del paquete y la página principal para enviar el parche.

Note that in Ubuntu, we set the `Maintainer` field to a general address because anyone can change any package (this differs from Debian where changing packages is usually restricted to an individual or a team). Packages in Ubuntu should generally have the `Maintainer` field set to `Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>`. If the Maintainer field is modified, the old value should be saved in the `XSBC-Original-Maintainer` field. This can be done automatically with the `update-maintainer` script available in the `ubuntu-dev-tools` package. For further information, see the Debian Maintainer Field spec on the Ubuntu wiki.

Cada párrafo adicional describe un paquete binario a ser construido.

For further information, see the control file section (Chapter 5) of the Debian Policy Manual.

2.2.3 El archivo copyright

This file gives the copyright information for both the upstream source and the packaging. Ubuntu and Debian Policy (Section 12.5) require that each package installs a verbatim copy of its copyright and license information to `/usr/share/doc/$(package_name)/copyright`.

Por lo general, la información sobre los derechos de autor se encuentra en el archivo `COPYING` dentro del directorio de código fuente del programa. Este archivo debe contener información tal los nombres de los autores, del creador del paquete, la URL de donde salió el código fuente y una línea de Copyright con el año, el titular de los derechos de autor y el propio texto del copyright. Una plantilla de ejemplo sería:

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: Hello
Source: ftp://ftp.example.com/pub/games

Files: *
Copyright: Copyright 1998 John Doe <jdoe@example.com>
License: GPL-2+

Files: debian/*
Copyright: Copyright 1998 Jane Doe <packager@example.com>
License: GPL-2+

License: GPL-2+
 This program is free software; you can redistribute it
 and/or modify it under the terms of the GNU General Public
 License as published by the Free Software Foundation; either
```

```
version 2 of the License, or (at your option) any later
version.
.
This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU General Public License for more
details.
.
You should have received a copy of the GNU General Public
License along with this package; if not, write to the Free
Software Foundation, Inc., 51 Franklin St, Fifth Floor,
Boston, MA  02110-1301 USA
.
On Debian systems, the full text of the GNU General Public
License version 2 can be found in the file
`/usr/share/common-licenses/GPL-2'.
```

This example follows the Machine-readable debian/copyright format. You are encouraged to use this format as well.

2.2.4 El archivo de reglas

El último archivo que se necesita estudiar es el archivo `rules`. Este archivo hace todo el trabajo para crear el paquete. Es un Makefile con secciones para compilar e instalar el programa, y después crear el archivo `.deb` a partir de los archivos instalados. También tiene una sección para limpiar todos los archivos generados por la compilación, con el fin de volver a quedarse únicamente con el paquete fuente.

Esta es una versión simplificada del archivo de reglas creado por `dh_make` (el cual se puede encontrar en el paquete `dh-make`):

```
#!/usr/bin/make -f
# -*- makefile -*-

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

%:
	dh  $@
```

Analicemos con más detalle este archivo. Lo que hace es pasar cada sección de compilación invocada por `debian/rules` como un parámetro a `/usr/bin/dh`, que a su vez ejecutará todos los comandos `dh_*` necesarios.

`dh` ejecuta una secuencia de órdenes de debhelper. Las secuencias soportadas son las que corresponden a las secciones del archivo `debian/rules`: «build», «clean», «install», «binary-indep», y «binary». Para saber cuales son las órdenes que se ejecutan en cada sección, ejecute:

```
$ dh binary-arch --no-act
```

A las órdenes de la secuencia binary-indep se incluye la opción «-i» para asegurar que solo funcionen en paquetes binarios independientes, mientras que a las órdenes de la secuencia binary-arch se les agrega una opción «-a» para asegurar que solo funcionen para paquetes independientes de la arquitectura.

Cada orden debhelper registrará cuando se ha ejecutado correctamente en «debian/package.debhelper.log» (archivo que dh_clean elimina). De esta manera dh puede saber qué órdenes ya han sido ejecutadas, para qué paquetes y evitar ejecutar esas órdenes de nuevo.

Cada vez que dh se ejecuta, comprueba el registro y busca la última orden registrada que está en la secuencia especificada. Continúa entonces con la siguiente orden de la secuencia. Las opciones --until, --before, --after y --remaining pueden modificar este comportamiento.

Si el archivo debian/rules contiene una sección con el nombre override_dh_command, cuando se llegue a esa orden en la secuencia, dh ejecutará esa sección en lugar de la orden original. La sección sobrescrita puede ejecutar entonces la orden con opciones adicionales, o ejecutar órdenes completamente diferentes en su lugar (téngase en cuenta que para usar esta característica, se debería construir las dependencias con debhelper 7.0.50 o superior).

Véase /usr/share/doc/debhelper/examples/ y man dh para más ejemplos. Véase también la sección sobre el archivo de reglas (sección 4.9) del manual de normas de Debian.

2.2.5 Archivos adicionales

El archivo de instalación

El archivo install lo usa dh_install para instalar archivos en el paquete binario. Tiene dos casos de uso estándar:

- Para instalar archivos en el paquete que no son manejados por el sistema de compilación del proyecto original.
- Partir paquete fuente en un único fichero grande en varios paquetes binarios.

En el primer caso, el archivo install debería contener solo una linea por cada archivo instalado, indicando tanto el archivo como el directorio de destino. Por ejemplo, el siguiente archivo install instalaría el script foo que se encuentra en la raíz del paquete en usr/bin y el archivo «desktop» del directorio debian en usr/share/applications:

```
foo usr/bin
debian/bar.desktop usr/share/applications
```

Cuando un paquete fuente crea varios paquetes binarios, dh instala los archivos en debian/tmp en lugar de debian/<paquete>. Los archivos instalados en debian/tmp pueden entonces moverse a diferentes paquetes binarios usando varios archivos $nombre_paquete.install. Esto se hace frecuentemente para separar grandes cantidades de datos que son independientes de la arquitectura de paquetes que dependen de la misma, llevándolos a paquetes Architecture: all. En este caso, solo se necesita el nombre de los archivos (o directorios) a instalar, sin el directorio de instalación. Por ejemplo, un archivo foo.install que solo contenga archivos dependientes de la arquitectura se vería así:

```
usr/bin/
usr/lib/foo/*.so
```

Por el contrario, un archivo foo-common.install que solo contenga archivos que son independientes de la arquitectura se vería así:

```
/usr/share/doc/
/usr/share/icons/
/usr/share/foo/
/usr/share/locale/
```

Esto crearía dos paquetes binarios, foo y foo-common. Ambos necesitarían su propia sección en el archivo debian/control.

Véase man dh_install y la sección sobre el archivo «install» (Section 5.11) de la guía del nuevo mantenedor de Debian para obtener más información.

El archivo de avisos («watch»)

El archivo debian/watch permite comprobar automáticamente si existe una nueva versión del proyecto original con la herramienta uscan, localizada en el paquete devscripts. La primera línea del archivo «watch» debe describir la versión del formato (3, en el momento de escribir esto), mientras que el resto de líneas contienen las URLs a analizar. Por ejemplo:

```
version=3

http://ftp.gnu.org/gnu/hello/hello-(.*).tar.gz
```

Ejecutando uscan en el directorio raíz del paquete comparará el número de versión original que se encuentra en debian/changelog con el último disponible en el proyecto original. Si se encuentra una versión más moderna del proyecto original, se descargará automáticamente. Por ejemplo:

```
$ uscan
hello: Newer version (2.7) available on remote site:
  http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz
  (local version is 2.6)
hello: Successfully downloaded updated package hello-2.7.tar.gz
    and symlinked hello_2.7.orig.tar.gz to it
```

If your tarballs live on Launchpad, the debian/watch file is a little more complicated (see Question 21146 and Bug 231797 for why this is). In that case, use something like:

```
version=3
https://launchpad.net/flufl.enum/+download http://launchpad.net/flufl.enum/.*/flufl.enum-(.+).tar.gz
```

Para más información, véase man uscan y la sección sobre el archivo «watch» (sección 4.11) del manual de normas de Debian.

Para obtener una lista de los paquetes de los que el archivo watch informa no estar sincronizados con las versiones originales véase la página Ubuntu External Health Status (estado de salud externa de Ubuntu, en inglés).

El archivo de formato («source/format»)

Este archivo indica el formato fuente del paquete. Solo debería contener una línea indicando el formato deseado:

- 3.0 (native) para paquetes nativos de Debian (sin versión en upstream)
- 3.0 (quilt) para paquetes con un archivo tar de upstream independiente.
- 1.0 para paquetes que desean declarar explícitamente el formato por defecto

Actualmente, el formato fuente del paquete será por defecto 1.0 si no existe este archivo. Se puede establecer esto explícitamente en el archivo «source/format». Si elige no usar este archivo para definir el formato, Lintian mostrará una advertencia sobre su ausencia. La advertencia es de carácter informativo y puede ser ignorada.

Se le anima a utilizar el nuevo formato fuente 3.0 . Proporciona una serie de nuevas características:

- Soporte para formatos de compresión adicionales: bzip2, lzma y xz
- Soporte para varios archivos tar de upstream
- No es necesario volver a empaquetar los archivos tar de upstream para eliminar el directorio debian
- Los cambios específicos de Debian ya no se guardan en un único archivo .diff.gz, sino en varios parches compatibles con quilt en debian/patches/

https://wiki.debian.org/Projects/DebSrc3.0 summarizes additional information concerning the switch to the 3.0 source package formats.

See `man dpkg-source` and the source/format section (Section 5.21) of the Debian New Maintainers' Guide for additional details.

2.2.6 Recursos adicionales

In addition to the links to the Debian Policy Manual in each section above, the Debian New Maintainers' Guide has more detailed descriptions of each file. Chapter 4, "Required files under the debian directory" further discusses the control, changelog, copyright and rules files. Chapter 5, "Other files under the debian directory" discusses additional files that may be used.

2.3 ubuntu-dev-tools: Tools for Ubuntu developers

`ubuntu-dev-tools` package is a collection of 30 tools created for making packaging work much easier for Ubuntu developers. It's similar in scope to Debian `devscripts` package.

2.3.1 Setting up packaging environment

`setup-packaging-environment` command allows to interactively set up packaging environment, including setting environment variables, installing required packages and ensuring that required repositories are enabled.

2.3.2 Environment variables

Introducing yourself

`ubuntu-dev-tools` configurations can be set using environment variables. It's used for example in changelogs. For example, to set e-mail address (and full name), use `UBUMAIL` variable. It overrides the `DEBEMAIL` and `DEBFULLNAME` variables used by `devscripts`. To learn `ubuntu-dev-tools` about you, open *~/.bashrc* in text editor and add something like this:

```
export UBUMAIL="Marcin Mikołajczak <marcin@example.org>"
```

Now, save this file and restart your terminal or use `source ~/.bashrc`.

Changing preferred builder

Default builder is specified by `UBUNTUTOOLS_BUILDER` variable. To set between *pbuilder* (default), *pbuilder-dist*, and *sbuild*, change this variable. Example:

```
export UBUNTUTOOLS_BUILDER=sbuild
```

Save file and restart terminal.

You can also check whether to update the builder every time before building, by changing `UBUNTUTOOLS_UPDATE_BUILDER` from `no` (default) to `yes`.

2.3.3 Downloading source packages

`ubuntu-dev-tools` comes with `pull-lp-source` command, allowing to download source packages from Launchpad. Its usage is simple. To download latest source package for ubuntu-settings, use:


```
$ pull-lp-source ubuntu-settings
```

You can also specify release from which you want to download source or specify version of source package. `-d` option allows to download source package without extracting. A slightly more complex example would look like this:

```
$ pull-lp-source brisk-menu 0.5.0-1 -d
```

`pull-debian-source` package allows to do the same for Debian source packages. It has similar syntax.

2.3.4 Backporting packages

`ubuntu-dev-tools` provides `backportpackage` allowing us to backport a package from specified release of Ubuntu or Debian. For example, to backport `bzr` package from latest development release for your installed Ubuntu version, simply:

```
$ backportpackage -w . bzr
```

This command allows to use more options. To specify Ubuntu release for which you are going to backport a package, use `-d dest` or `--destination=DEST` parameter, where `DEST` is Ubuntu release, for example `xenial`. You can specify more than one destination. In turn, `-s SOURCE` and `--source=SOURCE` specifies the Ubuntu or Debian release from which you are going to backport a package. `-w DIR` and `--workdir=DIR` specifies directory, where package files will be downloaded, unpacked and built. By default, it will create temporary directory that will be automatically deleted. `-U` or `--update` allows to update build environment before building package. `-u` or `--upload` allows to upload package after building (for example to PPAs) using `dput`.

2.3.5 Requesting backports

`requestbackport` command makes creating backports through Launchpad bugs much easier. It creates testing checklist that will be included in the bug. For example, to request backporting libqt5webkit5 from latest development branch to current stable release (without optional parameters):

```
$ requestbackport libqt5webkit5
```

You should fulfill the checklist if you have already tested the backport.

Additional options allows to specify destination of backport and its source, by using `-d DEST` or `--destination=DEST` and `s SRC` or `--source=SRC`.

2.3.6 Other simple commands

`ubuntu-dev-tools` also includes small utilities allowing to do simple tasks like checking whether .iso file is an Ubuntu installation media.

ubuntu-iso

To do this, use `ubuntu-iso <pathtoiso>`, for example:

```
$ ubuntu-iso ~/Downloads/ubuntu.iso
```


bitesize

"Bitesize" tag is used on Launchpad to describe tasks that are suitable for begineers who want to contribute to one of the projects. `bitesize` command allows to add "bitesize" tag to Launchpad bug with just simple command, by providing its number, like:

```
$ bitesize 1735410
```

404main

`404main` allows to check whether all of package build dependencies are included in main repository of specified Ubuntu distribution. Example:

```
$ 404main libqt5webkit5 xenial
```

If any of the required packages isn't part of Ubuntu main repository, you can check whether the package fulfill Ubuntu main inclusion requirements and request it.

Further reading

`ubuntu-dev-tools` manpages are covering more about usage of this package.

2.4 autopkgtest: pruebas automáticas para paquetes

The DEP 8 specification defines how automatic testing can very easily be integrated into packages. To integrate a test into a package, all you need to do is:

- añadir un archivo de nombre `debian/tests/control` que especifique los requisitos para el banco de pruebas,
- añadir las pruebas en `debian/tests/`.

2.4.1 Requisitos del banco de pruebas

In `debian/tests/control` you specify what to expect from the testbed. So for example you list all the required packages for the tests, if the testbed gets broken during the build or if `root` permissions are required. The DEP 8 specification lists all available options.

Más adelante vamos a ver el paquete fuente `glib2.0`. En un caso muy simple el archivo podría tener la siguiente forma:

```
Tests: build
Depends: libglib2.0-dev, build-essential
```

Para la prueba de `debian/tests/build` esto se aseguraría que los paquetes `libglib2.0-dev` y `build-essential` están instalados.

Nota: Puede usar `@` en la línea `Depends` para indicar que desea que estén instalados todos los paquetes que son compilados por el paquete fuente en cuestión.

2.4.2 La prueba real

La prueba asociada al ejemplo anterior podría ser:

```
#!/bin/sh
# autopkgtest check: Build and run a program against glib, to verify that the
# headers and pkg-config file are installed correctly
# (C) 2012 Canonical Ltd.
# Author: Martin Pitt <martin.pitt@ubuntu.com>

set -e

WORKDIR=$(mktemp -d)
trap "rm -rf $WORKDIR" 0 INT QUIT ABRT PIPE TERM
cd $WORKDIR
cat <<EOF > glibtest.c
#include <glib.h>

int main()
{
    g_assert_cmpint (g_strcmp0 (NULL, "hello"), ==, -1);
    g_assert_cmpstr (g_find_program_in_path ("bash"), ==, "/bin/bash");
    return 0;
}
EOF

gcc -o glibtest glibtest.c `pkg-config --cflags --libs glib-2.0`
echo "build: OK"
[ -x glibtest ]
./glibtest
echo "run: OK"
```

Aquí una porción de código C muy simple se escribe en un directorio temporal. Luego es compilada con las bibliotecas del sistema (usando los marcadores y rutas de bibliotecas proporcionadas por *pkg-config*). Luego el binario compilado, que simplemente prueba algunas partes de la funcionalidad del núcleo de glib, se ejecuta.

While this test is very small and simple, it covers quite a lot: that your -dev package has all necessary dependencies, that your package installs working pkg-config files, headers and libraries are put into the right place, or that the compiler and linker work. This helps to uncover critical issues early on.

2.4.3 Ejecutando la prueba

While the test script can be easily executed on its own, it is strongly recommended to actually use `autopkgtest` from the `autopkgtest` package for verifying that your test works; otherwise, if it fails in the Ubuntu Continuous Integration (CI) system, it will not land in Ubuntu. This also avoids cluttering your workstation with test packages or test configuration if the test does something more intrusive than the simple example above.

The README.running-tests (online version) documentation explains all available testbeds (schroot, LXD, QEMU, etc.) and the most common scenarios how to run your tests with `autopkgtest`, e. g. with locally built binaries, locally modified tests, etc.

The Ubuntu CI system uses the QEMU runner and runs the tests from the packages in the archive, with `-proposed` enabled. To reproduce the exact same environment, first install the necessary packages:

```
sudo apt install autopkgtest qemu-system qemu-utils autodep8
```

Now build a testbed with:


```
autopkgtest-buildvm-ubuntu-cloud -v
```

(Please see its manpage and `--help` output for selecting different releases, architectures, output directory, or using proxies). This will build e. g. `adt-trusty-amd64-cloud.img`.

Then run the tests of a source package like `libpng` in that QEMU image:

```
autopkgtest libpng --- qemu adt-trusty-amd64-cloud.img
```

The Ubuntu CI system runs packages with only selected packages from `-proposed` available (the package which caused the test to be run); to enable that, run:

```
autopkgtest libpng -U --apt-pocket=proposed=src:foo --- qemu adt-release-amd64-cloud.img
```

or to run with all packages from `-proposed`:

```
autopkgtest libpng -U --apt-pocket=proposed --- qemu adt-release-amd64-cloud.img
```

The `autopkgtest` manpage has a lot more valuable information on other testing options.

2.4.4 Más ejemplos

Esta lista no es completa, pero podría ayudarle a hacer una mejor idea de cómo están implementadas y cómo se usan las pruebas automatizadas en Ubuntu.

- The libxml2 tests are very similar. They also run a test-build of a simple piece of C code and execute it.
- The gtk+3.0 tests also do a compile/link/run check in the "build" test. There is an additional "python3-gi" test which verifies that the GTK library can also be used through introspection.
- In the ubiquity tests the upstream test-suite is executed.
- The gvfs tests have comprehensive testing of their functionality and are very interesting because they emulate usage of CDs, Samba, DAV and other bits.

2.4.5 Infraestructura de Ubuntu

Packages which have `autopkgtest` enabled will have their tests run whenever they get uploaded or any of their dependencies change. The output of automatically run autopkgtest tests can be viewed on the web and is regularly updated.

Debian also uses `autopkgtest` to run package tests, although currently only in schroots, so results may vary a bit. Results and logs can be seen on http://ci.debian.net. So please submit any test fixes or new tests to Debian as well.

2.4.6 Llevando la prueba a Ubuntu

El proceso de enviar un autopkgtest para un paquete es muy parecido a *fixing a bug in Ubuntu*. En esencia debe simplemente:

- ejecutar `bzr branch ubuntu:<packagename>`,
- editar el archivo `debian/control` para activar las pruebas,
- añadir el directorio `debian/tests`.
- write the `debian/tests/control` based on the DEP 8 Specification,
- añadir sus casos de prueba a `debian/tests`,

- confirmar los cambios, empujarlos a Launchpad, proponer una integración y hacer que la revisen igual que cualquier otra mejora de un paquete fuente.

2.4.7 Lo que puede hacer

The Ubuntu Engineering team put together a list of required test-cases, where packages which need tests are put into different categories. Here you can find examples of these tests and easily assign them to yourself.

If you should run into any problems, you can join the #ubuntu-quality IRC channel to get in touch with developers who can help you.

2.5 Usar chroots

Si ejecuta una versión de Ubuntu pero trabaja en paquetes de otra versión, puede crear un ambiente para esa otra versión con un `chroot`

Un `chroot` le permite tener un sistema de archivos completo de otra distribución el cual puede funcionar normalmente. Esto evita la sobrecarga de ejecutar una maquina virtual

2.5.1 Crear un chroot

Use la orden `debootstrap` para crear un nuevo chroot:

```
$ sudo debootstrap trusty trusty/
```

This will create a directory `trusty` and install a minimal trusty system into it.

If your version of `debootstrap` does not know about Trusty you can try upgrading to the version in `backports`.

Puede trabajar dentro del chroot:

```
$ sudo chroot trusty
```

Dónde puede instalar o eliminar cualquier paquete que desee sin afectar a su sistema principal.

Quizás quiera copiar sus claves GPG/ssh y su configuración de Bazaar dentro del chroot de manera que pueda acceder y firmar paquetes directamente:

```
$ sudo mkdir trusty/home/<username>
$ sudo cp -r ~/.gnupg ~/.ssh ~/.bazaar trusty/home/<username>
```

Para detener apt y otros programas que se quejan por la perdida de locales, puede instalar el paquete de idioma correspondiente:

```
$ apt-get install language-pack-en
```

Si quiere correr programas X necesita enlazar el directorio /tmp dentro del chroot, desde fuera del chroot ejecute:

```
$ sudo mount -t none -o bind /tmp trusty/tmp
$ xhost +
```

Algunos programas pueden necesitar vincularse a /dev o /proc.

For more information on chroots see our Debootstrap Chroot wiki page.

2.5.2 Alternativas

SBuild is a system similar to PBuilder for creating an environment to run test package builds in. It closer matches that used by Launchpad for building packages but takes some more setup compared to PBuilder. See the Security Team Build Environment wiki page for a full explanation.

Full virtual machines can be useful for packaging and testing programs. TestDrive is a program to automate syncing and running daily ISO images, see the TestDrive wiki page for more information.

Puede configurar pbuilder para hacer una pausa cuando encuentre un error en la construcción. Copie C10shell desde /usr/share/doc/pbuilder/examples en un directorio y use el parámetro `--hookdir=` para apuntar a él.

Amazon's EC2 cloud computers allow you to hire a computer paying a few US cents per hour, you can set up Ubuntu machines of any supported version and package on those. This is useful when you want to compile many packages at the same time or to overcome bandwidth restraints.

2.6 Setting up sbuild

`sbuild` simplifies building Debian/Ubuntu binary package from source in clean environment. It allows to try debugging packages in environment similar (as opposed to `pbuild`) to builders used by Launchpad.

It works on different architectures and allows to build packages for other releases. It needs kernel supporting overlayfs.

2.6.1 Installing sbuild

To use sbuild, you need to install sbuild and other required packages and add yourself to the `sbuild` group:

```
$ sudo apt install debhelper sbuild schroot ubuntu-dev-tools
$ sudo adduser $USER sbuild
```

Create `.sbuildrc` in your home directory with following content:

```
# Name to use as override in .changes files for the Maintainer: field
# (mandatory, no default!).
$maintainer_name='Your Name <user@example.org>';

# Default distribution to build.
$distribution = "bionic";
# Build arch-all by default.
$build_arch_all = 1;

# When to purge the build directory afterwards; possible values are "never",
# "successful", and "always".  "always" is the default. It can be helpful
# to preserve failing builds for debugging purposes.  Switch these comments
# if you want to preserve even successful builds, and then use
# "schroot -e --all-sessions" to clean them up manually.
$purge_build_directory = 'successful';
$purge_session = 'successful';
$purge_build_deps = 'successful';
# $purge_build_directory = 'never';
# $purge_session = 'never';
# $purge_build_deps = 'never';

# Directory for writing build logs to
$log_dir=$ENV{HOME}."/ubuntu/logs";
```



```
# don't remove this, Perl needs it:
1;
```

Replace "Your Name <user@example.org>" with your name and e-mail address. Change default distribution if you want, but remember that you can specify target distribution when executing command.

If you haven't restarted your session after adding yourself to the `sbuild` group, enter:

```
$ sg sbuild
```

Generate GPG keypair for sbuild and create chroot for specified release:

```
$ sbuild-update --keygen
$ mk-sbuild bionic
```

This will create chroot for your current architecture. You might want to specify another architecture. For this, you can use `--arch` option. Example:

```
$ mk-sbuild xenial --arch=i386
```

2.6.2 Using schroot

Entering schroot

You can use `schroot -c <release>-<architecture> [-u <USER>]` to enter newly created chroot, but that's not exactly the reason why you are using sbuild:

```
$ schroot -c bionic-amd64 -u root
```

Using schroot for package building

To build package using sbuild chroot, we use (surprisingly) the `sbuild` command. For example, to build `hello` package from x86_64 chroot, after applying some changes:

```
apt source hello
cd hello-*
sed -i -- 's/Hello/Goodbye/g' src/hello.c   # some
sed -i -- 's/Hello/Goodbye/g' tests/hello-1 #
dpkg-source --commit
dch -i                                      #
update-maintainer                           # changes
sbuild -d bionic-amd64
```

To build package from source package (`.dsc`), use location of the source package as second parameter:

```
sbuild -d bionic-amd64 ~/packages/goodbye_*.dsc
```

To make use of all power of your CPU, you can specify number of threads used for building using standard `-j<threads>`:

```
sbuild -d bionic-amd64 -j8
```


2.6.3 Maintaining schroots

Listing chroots

To get list of all your sbuild chroots, use `schroot -l`. The `source:` chroots are used as base of new schroots. Changes here aren't recommended, but if you have specific reason, you can open it using something like:

```
$ schroot -c source:bionic-amd64
```

Updating schroots

To upgrade the whole schroot:

```
$ sbuild-update -ubc bionic-amd64
```

Expiring active schroots

If because of any reason, you haven't stopped your schroot, you can expire all active schroots using:

```
$ schroot -e --all-sessions
```

2.6.4 Further reading

There is Debian wiki page covering sbuild usage.

Ubuntu Wiki also has article about basics of sbuild.

`sbuild` manpages are covering details about sbuild usage and available features.

2.7 Empaquetado de KDE

Packaging of KDE programs in Ubuntu is managed by the Kubuntu and MOTU teams. You can contact the Kubuntu team on the Kubuntu mailing list and `#kubuntu-devel` Freenode IRC channel. More information about Kubuntu development is on the Kubuntu wiki page.

Our packaging follows the practices of the Debian Qt/KDE Team and Debian KDE Extras Team. Most of our packages are derived from the packaging of these Debian teams.

2.7.1 Política de parches

Kubuntu no añade parches a los programas KDE a menos que vengan de los autores aguas arriba o se hayan enviado aguas arriba con la expectativa de que sean combinados pronto o hayamos consultado la incidencia con los autores aguas arriba.

Kubuntu no cambia la marca de los paquetes excepto cuando aguas arriba esperan que se haga (por ejemplo, el logotipo de la parte inferior izquierda del menú de inicio) o para simplificar (por ejemplo, eliminar las pantallas iniciales).

2.7.2 debian/rules

Los paquetes de Debian incluyen algunos añadidos al uso básico de Debhelper. Estos añadidos se mantienen en el paquete `pkg-kde-tools`.

Los paquetes que usan Debhelper 7 deberían añadir la opción `--with=kde`. Esto asegurará que se usan los marcadores de compilación adecuados y que se añaden opciones como la gestión de los «stubs» de kdeinit y las traducciones:

```
%:
    dh $@ --with=kde
```

Algunos paquetes más modernos de KDE usan el sistema `dhmk`, una alternativa a `dh` desarrollada por el equipo QT/KDE de Debian. Puede leer sobre el mismo en /usr/share/pkg-kde-tools/qt-kde-team/2/README. Los paquetes que lo usen incluirán `include /usr/share/pkg-kde-tools/qt-kde-team/2/debian-qt-kde.mk` en lugar de ejecutar `dh`.

2.7.3 Traducciones

Las traducciones de los paquetes de «main» en se importan en Launchpad y se exportan de Launchpad a los paquetes de idiomas de Ubuntu.

Así que todos los paquetes de «main» de KDE deben generar plantillas de traducción, incluir o hacer disponibles aguas arriba las traducciones y gestionar los archivos `.desktop` de traducciones.

Para generar las plantillas de traducción el paquete debe incluir un archivo `Messages.sh`; quéjese aguas arriba si no lo hace. Puede comprobar si funciona ejecutando `extract-messages.sh` lo que debería producir uno o más archivos `.pot` en el directorio `po/`. Esto se hará automáticamente durante la compilación si usa la opción `--with=kde` de `dh`.

Upstream will usually have also put the translation `.po` files into the `po/` directory. If they do not, check if they are in separate upstream language packs such as the KDE SC language packs. If they are in separate language packs Launchpad will need to associate these together manually, contact David Planella to do this.

Si un paquete se mueve desde «universe» a «main» tendrá que ser vuelto a subir antes de que las traducciones se importen en Launchpad.

Los archivos `.desktop` también necesitan traducciones. Se parchea KDELibs para que lea las traducciones de los archivos `.po` que están apuntados por una línea `X-Ubuntu-Gettext-Domain=` añadida a los archivos `.desktop` en el momento de compilar el paquete. Se genera un archivo .pot para cada paquete en el momento de la compilación y los archivos .po necesarios se descargan desde aguas arriba y se incluyen en el paquete o en los paquetes de idiomas. La lista de archivos .po a descargar de los repositorios de KDE está en `/usr/lib/kubuntu-desktop-i18n/desktop-template-list`.

2.7.4 Símbolos de biblioteca

Library symbols are tracked in `.symbols` files to ensure none go missing for new releases. KDE uses C++ libraries which act a little differently compared to C libraries. Debian's Qt/KDE Team have scripts to handle this. See Working with symbols files for how to create and keep these files up to date.

CAPÍTULO 3

Lecturas adicionales

You can read this guide offline in different formats, if you install one of the binary packages.

Si quiere conocer más sobre la construcción de paquetes Debian, aquí tiene algunos recursos Debian que pueden resultarle útiles:

- How to package for Debian;
- Debian Policy Manual;
- Debian New Maintainers' Guide — available in many languages;
- Packaging tutorial (also available as a package);
- Guide for Packaging Python Modules.

We are always looking to improve this guide. If you find any problems or have some suggestions, please report a bug on Launchpad. If you'd like to help work on the guide, grab the source there as well.