

# Packaging shared libraries

Josselin Mouette



19 may 2006

# A Debian package from a shared library

- `./configure --prefix=/usr`
- `make`
- `make install DESTDIR=debian/tmp`
- The `libfoo3` package contains:
  - the shared library `/usr/lib/*.so.*`.
- The `libfoo-dev` package contains:
  - the `/usr/lib/*.so` symbolic link;
  - the static library `/usr/lib/*.a`;
  - the headers in `/usr/include`;
  - the libtool file, `/usr/lib/*.la`;
  - the pkgconfig file, `/usr/lib/pkgconfig/*.pc`.

That's all.

That's all. Or not?

# Outline

- 1 Basic library packaging
- 2 Common developer mistakes
- 3 Symbol versioning
- 4 The inter-library dependency hell
- 5 Conclusion

# What is a shared library after all?

- Shared objects: code that can be loaded at runtime
- Shared libraries: automatically loaded at startup time

# What is a shared library after all?

- Shared objects: code that can be loaded at runtime
- Shared libraries: automatically loaded at startup time
  
- **API**: interface description for the programmer.  
API breaks → sources need to be changed.
- **ABI**: low-level interface between the binary and the library.  
ABI breaks → binaries need a rebuild.
- **SONAME**: canonical name for a given ABI.  
ABI breakage imply a SONAME change.

# Updating a libray package

General rule: the package manager has to know when an interface changes.

- 1 No ABI change (bugfix) → no change in packaging.
- 2 Backwards compatible ABI change (new symbols) → inform the *shlibs* system.

```
dh_makeshlibs -V'libfoo3 (>= 3.1.0)'
```

- 3 Symbols removed or their meaning changed (ABI breakage)  
→ **change the SONAME.**

Change the library package name: `libfoo3` → `libfoo4`.

- 4 API breakage → change the development package name:  
`libfoo3-dev` → `libfoo4-dev`.

# Library transitions

- A transition starts when the ABI breaks.
- All packages using the library have to be rebuilt.
- They all have to go to *testing* together.

→ Ask the release team before uploading.

# Library transitions

- A transition starts when the ABI breaks.
- All packages using the library have to be rebuilt.
- They all have to go to *testing* together.

→ Ask the release team before uploading.

Possible courses of action:

- 1 Upload right to *unstable* and trigger binary NMUs for all reverse dependencies.
- 2 Upload to *experimental* and rebuild depending packages in *experimental*.
- 3 Keep several sources at once. See *gnutls* or *libpng*.

## Case study: gconf2

Some libraries are more than a .so and some headers. Our example is split into several packages:

- `libgconf2-4` contains the shared library;
- `gconf2-common` contains the configuration and architecture-independent data;
- `libgconf2-dev` contains the development library and documentation;
- `gconf2` contains support binaries like `gconftool`.

## Case study: gconf2

Some libraries are more than a .so and some headers. Our example is split into several packages:

- `libgconf2-4` contains the shared library;
- `gconf2-common` contains the configuration and architecture-independent data;
- `libgconf2-dev` contains the development library and documentation;
- `gconf2` contains support binaries like `gconftool`.

But there is something else...

## Case study: gconf2

What to do with the daemon, `/usr/lib/gconf2/gconfd-2`, which is spawned by programs using the library?

## Case study: gconf2

What to do with the daemon, `/usr/lib/gconf2/gconfd-2`, which is spawned by programs using the library?

- In `gconf2`, circular dependency with `libgconf2-4` (RC bug).
- In `libgconf2-4`, goes against policy and breaks *multiarch*.

## Case study: gconf2

What to do with the daemon, `/usr/lib/gconf2/gconfd-2`, which is spawned by programs using the library?

- In `gconf2`, circular dependency with `libgconf2-4` (RC bug).
- In `libgconf2-4`, goes against policy and breaks *multiarch*.
- Solution: relocate the daemon to `/usr/lib/libgconf2-4/` and include it in `libgconf2-4`.

General rule about circular dependencies: a circular dependency that can't be removed means the two pieces of software should belong to the same package.

# Outline

- 1 Basic library packaging
- 2 Common developer mistakes**
- 3 Symbol versioning
- 4 The inter-library dependency hell
- 5 Conclusion

- Dynamic loading: the code isn't loaded at a given address.
- Code must not depend on its position.
  - Build with `-fPIC`.
- Insufficient for assembly code: has to be written with position independence in mind.
- Run `lintian`. Twice.

# Unexpected ABI breakage

- New upstream version with incompatible ABI but the same SONAME.
- Sometimes upstream acknowledges the error but doesn't come back on an already released package.
  - Random and various breakages in related packages.

# Unexpected ABI breakage

- New upstream version with incompatible ABI but the same SONAME.
- Sometimes upstream acknowledges the error but doesn't come back on an already released package.
  - Random and various breakages in related packages.
- Small issues of the kind can be solved with package conflicts.
- If not, the package is renamed: `libfoo3` → `libfoo3a`.
- A library transition starts.

# Unstable ABI

ABI breakage happens for all versions and the SONAME remains.

- HDF5: developers don't know what the SONAME means.
- Mozilla suite: overly complicated framework with unstable interfaces.
- ...
  - handle the SONAME specifically in Debian.

# Unstable ABI

ABI breakage happens for all versions and the SONAME remains.

- HDF5: developers don't know what the SONAME means.
- Mozilla suite: overly complicated framework with unstable interfaces.
- ...
  - handle the SONAME specifically in Debian.
- Breaks binary compatibility with other distributions

# Unstable ABI

ABI breakage happens for all versions and the SONAME remains.

- HDF5: developers don't know what the SONAME means.
- Mozilla suite: overly complicated framework with unstable interfaces.
- ...
  - handle the SONAME specifically in Debian.
- Breaks binary compatibility with other distributions. . .  
What compatibility?

# Unstable ABI

ABI breakage happens for all versions and the SONAME remains.

- HDF5: developers don't know what the SONAME means.
- Mozilla suite: overly complicated framework with unstable interfaces.
- ...
  - handle the SONAME specifically in Debian.
- Breaks binary compatibility with other distributions...

What compatibility?

- `libxpcor.so` → `libxpcor.so.0d`
- `libhdf5.so.0` → `libhdf5-1.6.5.so.0`

```
libtool --mode=link -release $(VERSION) ...
```

# Exporting private symbols

- Non-`static` functions and global variables are all exported.
- Developers only include useful functions in the public API.
- Other functions are exported but their ABI is not stable.
- Not part of the public API: not a problem?

# Exporting private symbols

- Non-static functions and global variables are all exported.
- Developers only include useful functions in the public API.
- Other functions are exported but their ABI is not stable.
- Not part of the public API: not a problem?
- Application developers write broken software using these functions.
- Libtool comes to the rescue:  

```
libSDL_mixer_la_LDFLAGS += -export-symbols-regex Mix_.*
```
- With a symbol list: use libtool's `-export-symbols` or a version script.

# Namespace conflicts

- GNU ld loads all symbols in the global symbol table.
- Exporting private symbols also leads to namespace conflicts if a program has a local function of the same name.
- For libraries, this happens occasionally.

# Namespace conflicts

- GNU ld loads all symbols in the global symbol table.
- Exporting private symbols also leads to namespace conflicts if a program has a local function of the same name.
- For libraries, this happens occasionally.
- Library plugins: code that can be added to the context of any application using the library.
- Example GTK+ theme engines or input modules.
- Applications are not tested against all plugins.
- Application code can override plugin code.

Solution: always build plugins with `-Wl,-Bsymbolic`.

# Outline

- 1 Basic library packaging
- 2 Common developer mistakes
- 3 Symbol versioning**
- 4 The inter-library dependency hell
- 5 Conclusion

# The problem

Consider a (not so hypothetical) package: a picture viewer using GTK+:

- uses libgtk for the GUI;
- uses libpng to read PNG files;
- libgtk already depends on libpng.

# The problem

Consider a (not so hypothetical) package: a picture viewer using GTK+:

- uses libgtk for the GUI;
- uses libpng to read PNG files;
- libgtk already depends on libpng.

Now libpng changes its ABI: `libpng.so.2` → `libpng.so.3`

- The package is rebuilt against `libpng3`.
- Upon startup, libgtk still brings `libpng.so.2`.
- All symbols are in the global symbol table.
- No way to tell the symbols from both libpng versions.

→ Crash!

# The solution

```
libpng12_la_LDFLAGS += -Wl,--version-script=libpng.vers
```

The *version script* looks like:

```
PNG12_0 {  
*; };
```

This gives a version names PNG12\_0 to the symbols in `libpng.so.3`.

For `libpng.so.2` (version 1.0.x), version PNG10\_0 is used.

# The result

```
$ objdump -T libpng.so.2 | grep png_init_io
00006260 g    DF .text 00000011 PNG10_0    png_init_io
$ objdump -T libpng.so.3 | grep png_init_io
000067a0 g    DF .text 00000011 PNG12_0    png_init_io
```

# The result

```
$ objdump -T libpng.so.2 | grep png_init_io
00006260 g      DF .text 00000011 PNG10_0      png_init_io
$ objdump -T libpng.so.3 | grep png_init_io
000067a0 g      DF .text 00000011 PNG12_0      png_init_io
```

When a binary is linked against this library, it now retains the needed version for each symbol:

```
$ objdump -T pngtest | grep png_init_io
00000000      DF *UND* 00000011 PNG12_0      png_init_io
```

If two symbols with the same name appear in the global symbol table, the dynamic linker will choose the one with the PNG12\_0 version.

- 1 All packages have to be rebuilt.
  - For libpng, fixed packages uploaded early in the sarge development process.
  - All depending packages have been rebuilt before the sarge release.
  - After the sarge release, libpng 1.0.x was removed.→ A clean transition takes a whole release to complete.
- 2 Such changes cannot be easily reverted.
  - Warning when a package requires symbol versions the library hasn't.
  - Failure when a package requires different symbol versions from the ones found.
  - If another distribution starts using different symbol versions, they become incompatible.→ Forward changes upstream.  
→ Ensure they are accepted as is: remember libmysqlclient.

# Versioned symbols + export restriction

Generally it is a good idea to have both.  
The version script looks like:

```
PNG12_0 { global:  
png_init_io;  
png_read_image;  
[...]  
local: *; };
```

# Outline

- 1 Basic library packaging
- 2 Common developer mistakes
- 3 Symbol versioning
- 4 The inter-library dependency hell**
- 5 Conclusion

# Libtool

- Tool to build libraries in a standard way, integrated with autoconf/automake.
- Builds a `libfoo.la` to include in the development package.
- Among other things, contains the dependency information necessary to build static libraries.
- At link time, `.la` files are read recursively to link in all dependencies.

- Tool to build libraries in a standard way, integrated with autoconf/automake.
- Builds a `libfoo.la` to include in the development package.
- Among other things, contains the dependency information necessary to build static libraries.
- At link time, `.la` files are read recursively to link in all dependencies.
- Not necessary for shared libraries, which already include a dependency system.
- In the context of a moving distribution this is harmful:

- Tool to build libraries in a standard way, integrated with autoconf/automake.
- Builds a `libfoo.la` to include in the development package.
- Among other things, contains the dependency information necessary to build static libraries.
- At link time, `.la` files are read recursively to link in all dependencies.
- Not necessary for shared libraries, which already include a dependency system.
- In the context of a moving distribution this is harmful:

*Dependencies:*      `libfoo1 → libbar3 → libbaz0`

- Tool to build libraries in a standard way, integrated with autoconf/automake.
- Builds a `libfoo.la` to include in the development package.
- Among other things, contains the dependency information necessary to build static libraries.
- At link time, `.la` files are read recursively to link in all dependencies.
- Not necessary for shared libraries, which already include a dependency system.
- In the context of a moving distribution this is harmful:

libfoo1 → libbar3 → libbaz0  
Added by libtool:           libfoo1 → libbaz0



- Tool to build libraries in a standard way, integrated with autoconf/automake.
- Builds a `libfoo.la` to include in the development package.
- Among other things, contains the dependency information necessary to build static libraries.
- At link time, `.la` files are read recursively to link in all dependencies.
- Not necessary for shared libraries, which already include a dependency system.
- In the context of a moving distribution this is harmful:

libfoo1 → libbar3 → libbaz1  
libfoo1 → libbaz0

Now libfoo1 has to be rebuilt against libbaz1!

# Relibtoolizing packages

Debian's libtool is fixed: dependencies are only used for static linking.  
→ Patch packages to use this libtool instead.

```
libtoolize --force --copy ; aclocal ;  
automake --force-missing --add-missing --foreign --copy ;  
autoconf ; rm -rf autom4te.cache
```

- Must be done for each new upstream release.
- Generally insufficient to get rid of indirect dependencies.

# Pkgconfig

- Tool to provide and access metadata concerning libraries.
- CFLAGS, LDFLAGS and dependencies.
- Dependencies are recursed.
  - Another source of unneeded indirect dependencies.

- Tool to provide and access metadata concerning libraries.
- CFLAGS, LDFLAGS and dependencies.
- Dependencies are recursed.
  - Another source of unneeded indirect dependencies.
- Recent changes allow `Requires.private` and `Libs.private` in `.pc` files.
- Private dependencies are only used for static linking.
- Fix your `.pc` files!

# GNU linker magic

- When everything else fails,
- when there is no more hope,

# GNU linker magic

- When everything else fails,
- when there is no more hope, GNU ld comes to the rescue.
- with the `--as-needed` option, it will skip unused dependencies it is passed.

```
LDFLAGS="-Wl,--as-needed" ./configure --prefix=/usr [...]
```

# GNU linker magic

- When everything else fails,
- when there is no more hope, GNU ld comes to the rescue.
- with the `--as-needed` option, it will skip unused dependencies it is passed.

```
LDFLAGS="-Wl,--as-needed" ./configure --prefix=/usr [...]
```

- A change in libtool reorders arguments.
- `--as-needed` becomes a dummy option.
- Patch available for `ltmain.sh`.

# Outline

- 1 Basic library packaging
- 2 Common developer mistakes
- 3 Symbol versioning
- 4 The inter-library dependency hell
- 5 Conclusion**

# Final words

- No magic recipes.
- Many known solutions to many problems.
- Release management is impossible without cooperation from library maintainers
- Forward patches upstream.
- Help upstream.
- Kick upstream.

- No magic recipes.
- Many known solutions to many problems.
- Release management is impossible without cooperation from library maintainers
- Forward patches upstream.
- Help upstream.
- Kick upsteam.

*Any questions?*