How to make package managers cry
(or)
How to piss off package managers
(pick one)

Kenneth Hoste
kenneth.hoste@ugent.be
GitHub: @boegel
Twitter: @kehoste

FOSDEM 2018
Package Management devroom
Feb 3rd 2018, Brussels (Belgium)
Context

- "package managers" (people) in the broad sense
  - anyone who needs to "install" software every now and then
- slight focus on scientific software
- some personal bias as lead developer of easybuild
  - framework to install (scientific) software on HPC systems
  - http://easybuilders.github.io/easybuild
- disclaimer: most of what I'm showing are not my ideas...
Goals

• present techniques to make software difficult to install

• mention excuses to get away with using them yourself

• how to score bonus points by taking things to the extreme

• examples of projects that have done a (really) good job
WARNING

This talk is meant to have a clear sarcastic tone.

Please do NOT take it too seriously.

It is meant to be an eye-opener regarding "bad" practices in software installation procedures.

Please do NOT interpret the given 'advice' as genuine.

I do NOT want to insult particular people or projects.
Common aspects of mentioned techniques

• create confusion

• surprise people (but not in a good way)

• annoy people

• trigger frustration

• aim for wasting (human) time
Reasons to employ these techniques

• try to get less people to use your software
  • they may find bugs, which you will need to fix
  • they may ask questions, or submit feature requests...

• avoid getting contributions
  • requires reviewing & testing
  • you will need to maintain the features they contribute!

• if they can't install your software, they will give up quickly

• also, motivate more people to use tools like easybuild
I. Creative software versioning & releasing

- don't use semantic versioning (don't see https://semver.org)

- make minor changes to releases, without bumping version

- don't do bugfix releases
  - tell people to check GitHub repository for updates
  - create a webpage with instructions on how to fix known bugs

- total lack of proper releases/versions
  - just a master branch in a GitHub repo, no tags/versions
  - let people come up with their own versioning scheme!

- remove old versions, do not keep an archive of previous releases
I. Creative software versioning & releasing

- excuses you can use:
  - "It was just a really tiny change, no need for a new version"
  - "Versions are not as important as they used to be."
  - "You should always use the latest available 'version'."
  - "Old versions had bugs, so they shouldn't be used anymore."

- bonus points:
  - have very strict version requirements for dependencies
  - clearly motivate your (lack of) versioning policy
OpenFOAM: no more (proper) bugfix releases

Replacing the “dot-1” Release

In past versions of OpenFOAM, we released a “dot-1” version, e.g. OpenFOAM–5.1, a few months after the release of the major version, i.e. 5.0. The rationale was that the dot-1 version contained code fixes to a large number of issues reported following a new version release. The timing of the dot-1 version release was based on issue reports falling to the background level.

Today the development line is maintained publicly to “always-releasable” quality. Issues are continuously reported and resolved, such that we no longer see a sharp rise in reported issues following release of a major version. The dot-1 version is therefore an anachronism, not relevant to our development of OpenFOAM today. Instead, we plan to release updated openfoam5 packs, compiled from the latest OpenFOAM–5.x sources, approximately once per month.

(taken from https://openfoam.org/news/v5-0-patch)
Problem with Morrison Scheme (Posted January 15, 2016)

Problem: When using the Morrison scheme without any cumulus turned on in any of the domains (for example, running a single domain with Morrison scheme only), problems exist because the Morrison scheme uses some tendency arrays from output with a cumulus scheme. However these arrays were not allocated when no cumulus scheme is used.

Solution: If you wish to use this scheme without any cumulus, you must edit Registry.EM_COMMON, and update this line, from:

```plaintext
package morr_two_moment mp_physics==10 - moist:qv,qc,qr,qi,qs,qg;scalar:qni,qns,qnr,qng
```

to

```plaintext
package morr_two_moment mp_physics==10 - moist:qv,qc,qr,qi,qs,qg;scalar:qni,qns,qnr,qng;state:rqrcuten,rqscuten,rqicuten
```

Once you update the file, save the file, and then you will need to go back to the WRFV3/ directory, issue a 'clean -a', reconfigure, and recompile the code.

(taken from http://www2.mmm.ucar.edu/wrf/users/wrfv3.7/known-prob-3.7.1.html)
'releases', no old versions

http://bioconductor.org

- creative interpretation of 'releases'
  - bundle of R packages with a particular release version (e.g. 3.6)
  - versions included in latest release get bumped...
  - ... without bumping the overall version of the bundle
- individual packages are not (always) archived
  - version bump in latest release implies *removing* old version
  - "nobody should use it anymore, it had bugs"
II. Don't provide release notes/changelog

• leave people guessing what has changed

• at the very least make release notes very vague
  • "minor enhancements & bug fixes"

• excuses you can use:
  • "See commit history on GitHub for more details."

• bonus points:
  • mention release notes are "coming soon"
    (and then never provide them)
III. Vendoring dependencies

• ship copies of required dependencies with your software

• excuses you can use:
  • "Makes installation easier."
  • "We know best how dependencies should be installed."

• bonus points
  • postpone updating included dependencies as long as possible
  • make some minor adjustments (and don't contribute back)
  • only include some dependencies
IV. Automagic installation of dependencies

- download & install dependencies during installation process

- excuses you can use:
  - "Makes installation easier."
  - "It's not unfair to assume that internet is reachable."

- bonus points
  - don't properly document dependencies
  - make it difficult to provide dependencies via another way
  - only do this for some dependencies
  - change your mind at some point to surprise people
V. More dependencies is better

- more dependencies implies more stuff to be installed
- try to favour dependencies that are hard to install themselves
- excuses you can use:
  - "I don't want to re-invent the wheel."
- bonus points:
  - mix different programming languages
  - make your software a common dependency, and rule the world
left-pad success story

- https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos

- > 250 JavaScript modules removed from NPM

- including some very popular ones like 'left-pad'
  - a tiny (trivial) module to 'indent' strings

- lots of stuff depended on left-pad, including Node.js

- removing of left-pad from NPM broke half the Internet!
QIIME dependency hell

- bioinformatics software (https://qiime2.org)

- requires Python, Perl, R, Haskell, OCaml, ...

- released as VM, containers (don't do this, let people install)
WARNING

This talk is meant to have a clear *sarcastic* tone.

*Please do **NOT** take it too seriously.*

It is meant to be an eye-opener regarding "bad" practices in software installation procedures.

Please do **NOT** interpret the given 'advice' as genuine.

I do **NOT** want to insult particular people or projects.
• hardcode as much as possible:
  • names of commands, in particular compilers (gcc, g++)
  • compiler options, (no) optimisation flags
    (pro tip: default for GCC is -O0!), ...
  • locations of libraries, header files, even commands!
  • versions of dependencies

• excuses you can use:
  • "We expect a standard environment."
  • "We can't support all possible environments out there."
VII. Choose your tools wisely (or don't choose)

• prefer using tools that people are not familiar with (yet)

• switch to something else when a tool becomes 'mainstream'

• use popular tools that nobody likes

• use tools with 'special' behaviour
  • resetting or taking control of the environment
  • hard to debug/fix when something goes 'wrong'

• or use your own scripts rather than an existing tool
  • or at least create wrappers around tools people know
VII. Choose your tools wisely (or don't choose)

• excuses you can use:
  • "These modern tools are a lot better."
  • "We can't keep living in the past, we need to move forward."
  • "I prefer to use my own scripts."

• bonus points
  • don't use the tools as they're intended to be used
  • require an ancient or very recent version for some reason
  • name your own scripts after existing tools ('./configure')
- http://scons.org

- "a next-generation build tool"

- "improved, cross-platform substitute for classic Make utility"

- resets environment in which commands are executed
  - $PATH is reset to /usr/local/bin:/bin:/usr/bin
  - can't find commands installed in a non-standard location

- (can be controlled via $ENV construction variable, don't tell anyone)
• https://bazel.build

• uses hardcoded locations for compilers, etc.
  • /usr/bin/{ar,cpp,gcc,ld}, /usr/lib/gcc, /usr/include, ...

• takes control over environment (like SCons does)

• confusing command line options:
  '-copt', '-config=opt' and '-c opt' are three different things!

• weird syntax:
  
  bazel build --config=opt //tensorflow/tools/pip_package:build_pip_package

I am not sure what this does, but I going to guess that the double slash is a typo.
• popular configuration & build tool, but nobody really likes it

• OK if all goes well, but if stuff goes wrong you're in trouble

• hard to figure out what's really wrong

• convincing CMake to behave is even more challenging

• excuses:
  • You don't really need one, everybody uses it already!
VIII. Partial installation procedure

• no configuration mechanism (just hardcoding)
• no test suite
• no support for installing build artefacts somewhere else
• excuses
  • "Not really needed, it's pretty trivial."
• bonus points:
  • provide a test suite, but include broken tests!
  • hide build artefacts in multiple (deep) subdirectories
IX. Interactive scripts

• ask questions, only accept specific answers (numbers, words)

• try to make it hard to automate

• provide a fallback "silent" mechanism for your own sanity (but don't document it!)

• excuses you can use:
  • "Interactive scripts are more intuitive."

• bonus points:
  • numbered list of possible answers, change it over releases
• https://www.tensorflow.org

• Python library for machine learning/deep learning

• originally developed by Google Brain team

• most forked GitHub project in 2017 (5th in #contributors)

• very popular in scientific research thanks to deep learning hype

• great performance on GPU \o/
• binary Python 'wheels' are made available via PyPI

• incentive to install it from source for good performance
• interactive "./configure" script (not Autotools as you may expect)
  • also picks up $TF_NEED_* env vars (undocumented)

• uses Bazel as build tool
  • resets environment, hardcodes compiler & co to /usr/...

• auto-installs some dependencies (but not Python, CUDA, cuDNN)

• need to "pip install" self-built Python wheel...
Conclusions

• tons of things you can do to make your software hard to install

• goals: confusion, surprise, annoyance, frustration, wasting time

• people can't complain about software they can't get to run

• lots of projects out there with good ideas, leverage them

• good excuses are not that hard to come up with

• be creative, go for bonus points!
WARNING

This talk is meant to have a clear **sarcastic** tone.

*Please do **NOT** take it too seriously.*

It is meant to be an eye-opener regarding

"bad" practices in software installation procedures.

Please do **NOT** interpret the given 'advice' as genuine.

I do **NOT** want to insult particular people or projects.