# mkpkg

*Release 8.0.1*

**Gene C**

**Jan 03, 2026**

# CONTENTS

CHAPTER

ONE

# MKPKG

## 1.1 Overview

Tool to rebuild Arch packages based on dependency triggers.

- All git tags will be signed by <arch@sapience.com>. Public key is available via WKD or download from website: https://www.sapience.com/tech After key is on keyring use the PKGBUILD source line ending with *?signed* or manually verify using *git tag -v <tag-name>*

## 1.2 New / Interesting

**8.0.0**

- Switch python packaging from hatch to uv
- License is GPL-2.0-or-later
- Small source code tree reorg.

**7.8.0**

- New feature: Support for uninstalled package dependency getting version from program.

  There are cases when building a package which depends on another uninstalled package.

  One example is pigeonhole which depends on dovecot. At build time, dovecot version being used to build against is not yet installed.

  Solved by providing a program (like bash script) which returns the version of the package name provided as an argument.

  The new PKGBUILD variable, _dep_vers_prog is a bash associative array which uses the package name as the key and the script that returns the version as the value.

  Example is provided below.

**Older**

- Use run_prog() from pyconcurrent module if it is available, otherwise use a local copy.
- Fixed issue where build subprocesses that generate very large amounts of data on stdout/stderr could occasionally lead to blocked IO when data exceeded python IO.DEFAULT_BUFFER_SIZE. Symptom is that the build hangs waiting for IO to get unblocked. Fixed by enhancing run_prog() to use non-blocking I/O.
- Immproved code

  PEP-8, PEP-257, PEP-484 and PEP-561 Refactor & clean ups.

- Improved handling of split packages.

  Now checks every packages for any being missing or out of date.

- soname logic updated.

  Default is now 'keep' which only rebuilds of a soname is no longer available. This is in line with how sonames are typically used where soname only changes when ABI changes.

- Major update: soname handling has been re-written from scratch and improved substantially.

  It now identifies every soname versioned library in elf executables along with their full path. It also properly handles executables built with *–rpath* loader options.

  Previous versions relied on makepkg soname output which, unfortunately, only lists sonames if they are also listed as a PKGBUILD dependency. We need every soname versioned library to ensure we do the right thing and rebuild when needed. So it was a mistake to rely on this.

  Can also specify how to handle version comparisons similar to the way package version comparisons are done (e.g. soname > major)

  If you're interested, the soname info is saved into the file *.mkp_dep_soname*

  **N.B.**
  that the build must be run at least once with this new version to generate the soname info (mkpkg -f forces a fresh build)

CHAPTER

TWO

# MKPKG APPLICATION

## 2.1 Overview of mkpkg

Building an Arch package requires invoking *makepkg* with a *PKGBUILD* file. PKGBUILD file contains a *depends* variable which lists those packages that are needed to use tool provided by the package.

It also has a 'makedepends' variable which is a list of other packages that are needed to build the package. makepkg also assumes that any package listed in the *depends* variable must also be present to build the package.

However, once a package has been built, then the only thing which causes a rebuild is a change to the actual package version itself. This can be either because the version of the tool itself changed or because the packager manually changed the release version, thereby forcing a rebuild.

If you have ever needed to rebuild a package by manually bumping the release version, then something is not ideal. If something requires a rebuild, other than the package itself having an update, it would be far better if this is done automatically rather than by hand.

This is what mkpkg does. It automates rebuilds when they are needed for some reason other than the tool / package version itself being newer. As a simple example, if something depends on openssl and the last build was against 3.0.0, then it can be set to rebuild if openssl has been installed more recently version than when the last package build was done. It could also be set to rebuild if openssl has a minor version update like 3.1.x.

Triggers are discussed in detail below, *mkpkg-triggers*, but we'll provide a short summary here.

## 2.2 Triggering Rebuilds Summary

To accomplish this mkpkg allows you to define a set of *triggers* that will cause a rebuild. These are packages, or files, that trigger a rebuild whenever they change in a specified way. Straightforward concept.

The packager is responsible for providing the list of appropriate triggers.

The way to provide the these triggers is by adding a PKGBUILD array variable to provide the list of conditions that should trigger a rebuild.

### 2.2.1 Package Trigger

*_mkpkg_depends* variable is this list of such triggers

There are 2 ways a package can trigger a rebuild.

1. **A package name**

   Rebuilds if the install date is more recent than the last build time e.g.

```
_mkpkg_depends=('openssl' 'systemd')
```

2. **A package name plus a version condition**

   It can be an explicit version or a key word such as *major* which would then only trigger a rebuild when the major version of that package was greater than that at the last build. More details and the different options are detailed below. e.g.

```
_mkpkg_depends=('openssl>3.0.6' 'systemd')
```

### 2.2.2 File Trigger

It can also use any file to trigger a build using *_mkpkg_depends_file*. When a file in this list is newer than the lsat build, it triggers a rebuild.

A typical use case for these file triggers are files provided by the packager, rather than the source, and include things such as systemd unit files or pacman hook files or other package related items. e.g.:

```
_mkpkg_depends_file=('xxx.service')
```

This is useful to ensure packages build and work when conditions are met by other packages being updated.

It is certainly helpful for packages which statically link in libraries, or when core build tools change and it's important to rebuild with the newer versions. Do we really need to rebuild a package when tool chain changes? Sometimes yes; for example, whenever the compiler toolchain is updated, I always rebuild my kernel packages and test.

The majority of compiled packages are built against shared libraries and this can be helpful in this case too; there are additional comments on this topic below.

As another example, I rebuild my python applications when python's major.minor is larger than what was used for previous build.

An additional little benefit, if packages are up to date then running mkpkg is significantly faster than makepkg; can be something like 10x faster or even more.

## 2.3 Background Motivation

mkpkg has one run-time dependency, python.

It uses makepkg to perform the actual package builds in the usual way. That said, makepkg is a part of pacman which is always installed and thus not a *dependency* as far as PKGBUILD is concerned.

When a tool chain used to build a package is updated, it's good practice, IMHO, to rebuild packages which use that tool chain. For example, when gcc, cargo, binutils et al are updated packages using those tools should also be updated. As mentioned above, whenever compiler/binutils tool chain changes, I always rebuild and test my kernel packages. This not only ensures that things compile and work properly with the new build tools but can also be key to reducing the attack surface. One recent (as of time of writing) little example, not to pick on cargo, is CVE-2022-36113

Of course this would require a case where cargo is actually downloading something which should never be permitted; still, it's a conceivable danger.

While static linked libraries surely don't demand a rebuild to function, obviously, because the older library is part of the binary itself, it's still a good idea to rebuild it. This will pick up bug fixes, including security related ones, as well as improvements. Of course, it's always sensible to confirm that an application properly builds and works with the newer tool or library as well.

Here's an example. The *refind* boot manager statically links against gnu-efi. So when gnu-efi is updated, refind should be rebuilt as well even though the previous one will continue to work just fine.

Recently, arch started switching many packages to be compiled with lto. The gnu-efi package was subsequently compiled with * -flto -ffat-lto-object*. The refind boot manager statically links gnu-efi. At this point, refind itself had not changed and so it's up to date as far standard approach is concerned.

However, I would like to know as early as possible that refind builds and runs with the the new gnu-efi library that was updated. In fact, unfortunately perhaps, this build failed and refind not longer builds with the updated gnu-efi library due to lto changes. Good to know.

You could of course have waited until refind itself gets an update and then discover - oh no it no longer builds. But, by doing this early and in this case knowing refind itself has not changed, I know with certainty that this problem stems from the gnu-efi rebuild and not from a refind change - without even looking at any refind source changes.

Given the large number of packages I build I doubt I'd remember what trigger packages are approprate for every package anyway. Computers are good at automating repetitive tasks after all and are much quicker at identifying the trigger packages.

mkpkg was created to address this need. It automates this for you and rebuilds packages when needed. This allows for early detection of problems or confirmation that things are actually fine.

A small comment on shared libraries. While these are generally not a problem, there is an assumption that the library itself still functions the same for whatever part of it the tool is using.

The majority of providers are careful with *sonames* as well, so most of the time that's likely true, however, the cautious among us may want to run regression tests even in this case.

Certainly for mission critical tools. Bugs happen, and it's good to learn of any issues as soon as possible.

But there are indeed some shared library packages, some with dynamically loaded libraries (plugins) that may also be trigger packages. One symptom of that need are those packages that are manually rebuilt by forcing a release version bump typically with a comment such as *rebuilt with latest …* - we certainly see plenty of that happening.

CHAPTER

THREE

# USING MKPKG

## 3.1 Getting Started

Edit the PKGBUILD and add a *_mkpkg_depends* variable with a list of triggers that should cause a rebuild when the condition is met. Triggers are discussed in in detail (*mkpkg-triggers*) below, but a simple example is:

```
_mkpkg_depends=('python>major', 'python-foo')
```

This would trigger a package rebuild if a version of *python-foo* is installed more recently than the last package build or if *python* has a major version which is larger than that used when package was last built.

With the trigger conditions in the PKGBUID, then simply call mkpkg instead of makepkg. Couldn't be simpler. Options for mkpkg are those before any double dash –. Any options following – are passed through to *makepkg*[2].

## 3.2 Options

The options currently supported by mkpkg are:

- (**-v, –verb**)

  Show (stdout) output of makepkg. Default is not to show it.

- (**-f, –force**)

  Force a makepkg run even if not needed. Bump the package release and rebuild

- (**-r, –refresh**)

  Attempts to update saved metadata files. Faster, if imperfect, alternative to rebuild. If there is no saved metadata, and build is up to date, will try refresh the build info. Files updated are *.mkp_dep_vers* and *.mkp_dep_soname*.

  Note that *sonames* are found by examining any executables in the *pkg* directory. If the *pkg* directory is empty, the refresh will not find any sonames.

- (**so-comp, –soname-comp**)

  How to handle automatic soname changes. Default value is *keep* - only rebuilds if soname is no longer available.

  - *newer* : if soname is newer then reubild (time based)
  - *keep* : if soname library is still available, then dont rebuild even if newer version(s) are available
  - *vcomp* : rebuild if soname version is greater than the *vcomp* version. *vcomp* is one of *major*, *minor*, *patch*, *extra* or *last* - same as for regular depenencies.
  - *neverever* : Developer option - will not rebuild even if the soname library is no longer available.

[2] The older style options using *–mkp-* are now deprecated.

- (–)

  All options following this are passed to makepkg

**Config file**

Configs are looked for in first in /etc/mkpkg/config and then in ~/.config/mkpkg/config. Config files are in TOML format. e.g. to change the default soname rebuild compare option from default of *last*:

```
soname_comp = "newer"
```

## 3.3 How mkpkg works

Outline of what it does

- If PKGBUILD has a pkgver() function, check if the pkgver variable matches its output
- If the 2 pkgver match or if there is no pkgver() function then check if a matching package exists
- If package not up to date, then run makepkg build.
- If package seems otherwise up to date, then check if any of the conditions given by *mkpkg_depends* or *mkpkg_depends_files* triggers a build. If a build is called for, then bump the pkgrel and rebuild.
- If the package is out of date, as there is newer version then reset pkgrel back to "1" and build.

So, if a package builds and gets larger package release number, it was because of some trigger package dependency; absent manual modification. If package release is "1" - then you know its a fresh package version.

I use separate tool to run all my package builds so I prefer the output to be easily parseable and provide simple and clear information to feed the builder too.

mkpkg thus prints a line of the form:

```
*mkp-status: <status> <package-version>*
```

Where status is one of :

- **current** -> package is up to date
- **success** -> package was built successfully
- **error** -> problem occurred.

Obviously, package-version is what is sounds like.

It is possible for mkpkg itself to fail for some reason, in which case the *mkp-status:* line could be absent. This is also simple to detect programatically.

## 3.4 Triggering Rebuilds Details

### 3.4.1 _mkpkg_depends

There are 2 kinds of triggers. A trigger based on package and a trigger based on file changed. Each is set using the PKGBUILD variable with a an array of triggers. The variables used are:

- **_mkpkg_depends**

This variable provides a list of packages to trigger a rebuild. Each item in the list can be in one of 2 forms:

1. *name*

   The item is the name of the package then this will trigger a rebuild if the install time of a listed package is newer than the time of the last build.

2. *package_name compare-op vers_trigger*

   This provides semantic version triggers. Package versions are taken to be of the form 'major.minor.patch' or more generally 'elem1.elem2.elem3….' White space around the comparison operator is optional.

- *compare-op*

  is one of : **>**, **>=** or **<**

- *vers_trigger*

  Based on comparing the first [N] elems of the version or the entire version.

  - First_[N] : rebuild if first [N] elems of package version greater than when last built
  - major : alias for First_1 (rebuild if major > last_build)
  - minor : alias for First_2 (rebuild if major.minor > last_build)
  - patch : alias for First_3 (if major.minor.patch > last_build)
    - micro : another name for patch
  - extra : alias for First_4 (major.minor.patch.extra)
    - releaselevel : alias for extra
  - serial : alias for First_5 (major.minor.patch.extra.serial)
  - last : rebuild if package version > last_build version.

*last* is very similar to a time based trigger but based on version instead of time.

For example if the expression is

```
'pkg_name>First_2'
```

or equivalently:

```
'pkg_name>minor'
```

and the current package version is 1.2.3, while the version when last built was 1.2.0 then the versions being compared would be

```
'1.2' > '1.2' which is false.
```

Whereas if the expression was:

```
'pkg_name>First_3'
```

then the comparison would be

```
'1.2.3' > '1.2.0'
```

which is true

N.B. The package must be built at least once using mkpkg so it can save the dependent package versions used. So if a version trigger is added, then this triggers a rebuild as it treats this as if the dependent package version is greater than last

used (which is not known at this point). On subsequent builds the last built version of each dependent package is then known.

Unlike the standard *makedepends* variable, this allows one to not include things that are required to build the package but don't have any affect on the tool function. For example 'git' - which while required to build will not generally change the tool.

Another example, if python was version 3.10 when the package was last built and we have::

```
_mkpkg_depends=('python>minor' 'python-dnspython')
```

Then a rebuild will be done if python is greater than or equal to 3.11.x or if python-dnspython was installed more recently than the last build. This will not trigger a rebuild if python is updated from 3.10.7 to 3.10.8, since this is a patch update not a minor or major update.

Why support '<' you may ask. The only sensible use for less than operator would be to provide a mechanism to trigger a rebuild when a package gets downgraded. This would be accomplished using

```
pkg_name < last
```

### 3.4.2 _mkpkg_depends_files

- *_mkpkg_depends_files*

  This variable can be used to provide a list of files that should trigger a rebuild. The files are relative to the directory containing PKGBUILD.

This might be useful, for example, if the source for some daemon doesn't provide a systemd service file, and the packager adds the file. Adding the file to this list would now trigger rebuilds should there be changes to the service file. An alternative would be to put these files into a git repo and just using the git version. For a small number of files this may be more convenient/simpler.

These variables offer considerable control over what can be used to trigger rebuilds.

### 3.4.3 Variable _dep_vers_prog

In cases where a dependent package being used to build against is not yet installed, then it's version cannot be extracted by calling *pacman -Qi*. Instead, a script can be provided in PKGBUILD which returns the version of the package name provided as an argument.

For example if we have the following dependencies where *foo* and *goo* are used but not yet installed on the build machine (at least not the version being built against):

```
_mkpkg_depends=('python>minor', 'openssl>3.0', 'foo', 'goo')

declare -A _dep_vers_prog
_dep_vers_prog['foo']='./dep-vers'
_dep_vers_prog['goo']='./dep-vers'
```

where the script *./dep-vers* returns the version of it's one argument - where arguments can be *foo* or *goo*.

One example is pigeonhole and dovecot. This are in separate git repos and built as separate packages. pigeonhold is built against the just compiled version of dovecot. In this case a script which extracts *pkgver* from the .PKGINFG file either in the dovecot *pkg* directory, or by extracting it from the just built package does the trick.

## 3.5 Discussion and Next Steps

### 3.5.1 Possible future enhancement

While mkpkg works for all the packages I build, I am more than happy to take enhancement requests - and, of course, to fix bugs!

As mentioned earlier, it's pretty useful to run regression tests after run-time dependencies change. For example shared libraries or other programs used by the tool. To handle this case we might consider adding a separate variable - such as *mkpkg_test_depends* which lists these kind of dependencies.

We note that *checkdepends* vartiable is quite different in intent, as it is used to identify those packages needed to do testing but NOT for things which could impact the outcome of running the tool.

CHAPTER

FOUR

# APPENDIX

## 4.1 mkpkg Source

The source is kept in the github repository Github-mkpkg.

## 4.2 Installation

**Available on**

- Github-mkpkg
- Archlinux AUR

On Arch you can build using the provided PKGBUILD in the packaging directory or from the AUR. All git tags are signed with arch@sapience.com key which is available via WKD or download from https://www.sapience.com/tech. Add the key to your package builder gpg keyring. In PKGBUILD use source= line with *?signed* at the end. You can also manually verify the signature

To build manually, clone the repo and :

```
rm -f dist/*
/usr/bin/python -m build --wheel --no-isolation
root_dest="/"
./scripts/do-install $root_dest
```

When running as non-root then set root_dest a user writable directory

## 4.3 Dependencies

- Run Time: - python (3.9 or later) - pyalpm
- Building Package : - git - build aka python-build - intaller aka python-installer - wheel aka python-wheel - poetry aka python-poetry - rsync
- Optional for building docs:
  - sphinx
  - texlive-latexextra (archlinux packaguing of texlive tools)

## 4.4 Philosophy

We follow the *live at head commit* philosophy as recommended by Google's Abseil team[1]. This means we recommend using the latest commit on git master branch.

## 4.5 License

Created by Gene C. and licensed under the terms of the MIT license.

- SPDX-License-Identifier: MIT
- Copyright (c) 2022-2023 Gene C

## 4.6 Some history

### 4.6.1 Version 6.0.0

- soname rewrite

  New argument for how soname changes are treated : *-so-comp, –soname-comp*.

  Can be *<compare>*, *newer*, *never* or key how to compare the soname versions. The comparison types are the same as for package dependencies described above. Default is *last* which means the entire soname version will be compared to whats available and rebuild will be triggered if a later version now available.

  *<compare>* e.g. *>major* or *>minor*' or *last* etc. If the last built soname was 5.1, and now available is 5.2 then *minor* and *last* will trigger rebuild while *major* would not. *newer* triggers if the last modify time of the library is newer.

  Previous version used sonmaes produced by makepkg - however this only generates sonames if they are listed as dependencies. We want to get every soname - so we started over from scratch. By using our own soname generate we catch every soname and its absolute path - this enables us to correctly treat soname changes. This approach will also correctly deal with any *rpath* loader flags causing executable to use shared library from path(s) specified at compile time.

### 4.6.2 Version 4.1.0

- Arguments

  Change in argument handling. Arguments to be passed to *makepkg* must now follow –. Arguments before the double dash are used by mkpkg itself. To keep backward compatibility the older *–mkp-* style arguments are honored, but the newer simpler ones are preferred. e.g. *-v, –verb* for verbose. Help availble via *-h*.

- Config file now available.

  Configs are looked for in /etc/mkpkg/config then ~/.config/mkpkg/config. It should be in TOML format. e.g. to change the default soname rebuild option:

```
soname_comp = "newer"
```

### 4.6.3 Version 4.0.0

- Soname drive rebuilds.

  Adds support for detecting missing soname libraries, and triggering rebuild. If soname is found then no rebuild is done. Typically happens when older soname is deprecated.

[1] https://abseil.io/about/philosophy#upgrade-support

- Adds new option *–mkp-refresh*.

  Attempts to update saved metadata files. Faster, if imperfect, alternative to rebuild.

### 4.6.4 Older

Adds support for epoch.

Version 2.x.y brings fine grain control by allowing package dependences to trigger builds using semantic version. For example 'python>minor' will rebuild only if a new python package has it's major.minor greater than what it was when package was last built. See *_mkpkg_depends* below for more detail.

The source has been reorganized and packaged using poetry which simplifies installation. The installer script, callable from package() function in PKGBUILD has been updated accordingly. Ther build() function uses python build module to generate the wheel package, as outlined above.

Changed the PKGBUILD variables to have underscore prefix to follow Arch Package Guidelines. Variables are now: *_mkpkg_depends* and *_mkpkg_depends_files*. The code is backward compatible and supports the previous variable names without the leading "_" as well as the ones with the "_".

Now also available on aur.

CHAPTER

FIVE

# LICENSE

mkpkg is a an Archlinux package build tool.