Debug Packages in OpenBSD

Marc Espie <espie@openbsd.org>, <espie@lse.epita.fr>

Epita Research and Development Laboratory

September 18, 2021
Like most things in OpenBSD, it started at a hackathon,
the p2k19 hackathon, organized by Paul Irofti in Bucarest.
... and the original idea was Paul’s as well:
"hey, there’s this command we can use to split off the debug info from an executable. How about we make some debug packages."
Specifically

1. `objcopy --only-keep-debug program .debug/program.dbg`
2. `strip -d program`
3. `objcopy --add-gnu-debug-link=program .debug/program.dbg`
And that’s about all folks!

Wait what?

Of course there are fun details to take care of!
A more elaborate plan

- Add the debug package to "visible packages"?
- This requires a lot of changes, so let's not
- Debug packages should be "fantom packages that don’t really exist"
- As for update use the exact same update signature as the normal package
At that point, make package is more or less make package-cookies.
so we just need to make extra cookies for the debug packages.
Naming convention: add debug- to the front (to avoid cluttering the normal list).
Do the debug part manually at first: for each DEBUG_FILES, we apply the magic objcopy transformation.
... and write the packing-list (manifest) manually, at first.
This allowed us to check that egdb was happy with it.
• generation of DEBUG_PACKAGES is triggered by the presence of DEBUG_FILES
• we add extra DEBUG_CONFIGURE_ARGS to configure
• ... show that to crash-test developers (we’re at a hackathon, remember ?). Many thanks to all the folks who tested !
Very fast turn-around

- Turns out most people are interested in debug packages!
- ... so I got a lot of developers to test things and offer suggestions!
Some code:

```
# [...]  

DEBUG_PACKAGES ?=  
DEBUG_FILES ?=  
DEBUG_CONFIGURE_ARGS ?=  

.for _S in `${MULTI_PACKAGES}`  
    _PKGFILE${_S} = `${FULL_PKGNAME}${_S}`.tgz  
    _DBG_PKGFILE${_S} = debug-`${_PKGFILE${_S}}`  
    .if ${PKG_ARCH${_S}} == `*` && ${NO_ARCH} != `${MACHINE_ARCH}/all`  
        _PACKAGE_COOKIE${_S} = `${PACKAGE_REPOSITORY}/${NO_ARCH}/${_PKGFILE${_S}}`  
    .else  
        _PACKAGE_COOKIE${_S} = `${PACKAGE_REPOSITORY}/${MACHINE_ARCH}/all/${_PKGFILE${_S}}`  
        _DBG_PACKAGE_COOKIE${_S} = `${PACKAGE_REPOSITORY}/${MACHINE_ARCH}/all/${_DBG_PKGFILE${_S}}`  
    .endif  
.endfor
```
# [...]  

```bash
if !empty(DEBUG_PACKAGES) || !empty(DEBUG_FILES)
INSTALL_STRIP =
DEBUG_FLAGS = -g
CONFIGURE_ARGS += ${DEBUG_CONFIGURE_ARGS}
else
DEBUG_FLAGS =
endif

for _S in ${MULTI_PACKAGES}
PKG_ARGS${_S} += -A='${PKG_ARCH${_S}}'

_create_pkg${_S} = \\ 
    tmp=${_TMP_REPO}${_PKGFILE${_S}} pkgname=${_PKGFILE${_S}} && \\
    ${_PBUILD} ${_PKG_CREATE} -DPORTSDIR="${PORTSDIR}" \\
        $$deps ${PKG_ARGS${_S}} $$tmp && \\
    $$check_lib_depends $$tmp && \\"
Some code III

```bash
${_register_plist${_S}} $$tmp && \n${_checksum_package}

_move_tmp_pkg${_S} = ${_PBUILD} mv ${_TMP_REPO}${_PKGFILE${_S}} ${_PACKAGE_COOKIE${_S}}
$tmp_pkg${_S} = ${_TMP_REPO}${_PKGFILE${_S}}

  if ${DEBUG_PACKAGES:M${_S}}
  _DBG_PKG_ARGS${_S} := ${PKG_ARGS${_S}}
  _DBG_PKG_ARGS${_S} += -P${FULLPKGPATH${_S}}:${FULLPKGNAME${_S}}:${FULLPKGNAME${_S}}
  _DBG_PKG_ARGS${_S} += -DCOMMENT="debug info for ${FULLPKGNAME${_S}}"
  _DBG_PKG_ARGS${_S} += -d"-debug info for ${FULLPKGNAME${_S}}"
  _DBG_PKG_ARGS${_S} += -DFULLPKGPATH=debug/${FULLPKGPATH${_S}}
  _DBG_PKG_ARGS${_S} += -f ${PLIST${_S}}-debug
  _create_pkg${_S} += && \n    tmp=${_TMP_REPO}${_DBG_PKGFILE${_S}} pkgname=${_DBG_PKGFILE${_S}} && \n    ${_PBUILD} ${_PKG_CREATE} -DPORTSDIR="${PORTSDIR}" \n      $$deps ${_DBG_PKG_ARG${_S}} $$tmp && \n```
The code snippet provided includes some commented-out sections and conditional logic. Here is a plain text representation of the relevant parts:

```bash
${_check_lib_depends} $$tmp && \
${_register_plist}{$_S} $$tmp && \
${_checksum_package}
_move_tmp_pkg${_S} += && ${_PBUILD} mv ${_TMP_REPO}${_DBG_PKGFILE${_S}} ${_DBG_PACKAGE_COOKIE${_S}}
_tmp_pkg${_S} += ${_TMP_REPO}${_DBG_PKGFILE${_S}}
. endif

# [...] _copy_debug_info:
.for P in ${DEBUG_FILES:N*.a}
    @dbgpath=${PREFIX}/${P:H}/.debug; \\
    dbginfo=${dbgpath}/${P:T}.dbg; \\
    p=${PREFIX}/${P}; \\
    ${INSTALL_DATA_DIR} $$dbgpath; \\
    echo "> move debug info from $$p into $$dbginfo"; \\
    objcopy --only-keep-debug $$p $$dbginfo; \\
```

This code appears to be part of a script that handles file dependencies, checksums, and file movement, as well as copying debug information from specific files into a debug directory.
Some code V

69    objcopy --strip-debug $$p; \\
70    objcopy --add-gnu-debuglink=$${dbginfo} $$p
71    .endfor
72    .for P in ${DEBUG_FILES:M*.a}
73       @dbgpath=${PREFIX}/${P:H}/.debug; \\
74       dbginfo=$${dbgpath}/${P:T}; \\
75       p=${PREFIX}/${P}; \\
76       ${INSTALL_DATA_DIR} $${dbgpath}; \\
77       echo "> copy debug info from $$p into $${dbginfo}"; \\
78       cp $$p $${dbginfo}; \\
79       strip $$p
80    .endfor
Introducing build-debug-info

This just reuses an existing framework
- we read existing packing-lists in *update-plist*
- let’s just do the same to create the debug-plists
- ... except we generate temporary information
- and generate a `DEBUG_FILES` equivalent dynamically
- **update-plist** is fully OO
- The "parsing the existing lists part" is just common code with **pkg_create**
- There's a common class that parses parameters, with a derived subclass for **update-plist**
- This class requires very few changes to generate a **build-debug-info** tool
cat ${_WRKDEBUG}/debug-info| \ 
  while read dbgpath p dbginfo; do \ 
    ${INSTALL_DATA_DIR} $${dbgpath}; \ 
    echo "> copy debug info from $$p into $${dbginfo}"; \ 
    case $$p in *.a) \ 
      cp $$p $${dbginfo}; \ 
      strip $$p;; \ 
    *) \ 
      objcopy --only-keep-debug $$p $${dbginfo}; \ 
      objcopy --strip-debug $$p; \ 
      objcopy --add-gnu-debuglink=$${dbginfo} $$p; \ 
    esac; done
Feedback from friends

Stuart
- debug packages might be big
- ... so we do them opt-in
- ... also make this arch-dependent: amd64 first then we’ll figure out other architectures
- 32 bit arches are likely to be out in the cold

Antoine
- Hey, it doesn’t work with some python packages!
- That’s because of hardlinks.
- Obviously, if you have two links to the same binary, objcopy --split-debug-info will only work once!
A better picture

Packing-list data

pkg_create args

build-debug-info

Debug packing-lists

Debug files list

fake / staging area data
(to identify links)
All about links

- nothing to do with most binaries
- however the debug link does not have a path
- so we still need to do something for different directories
- most annoying part was tests!
we’re actually in multi-packages land...

So we set DEBUG_PACKAGES to the subpackages for which we want debug packages.

This will get trimmed automatically depending on architecture (exactly like for "normal" multi-packages: don’t deal with stuff that’s NOT_FOR_ARCHS. and don’t do debug for arch independent stuff: a subpackage that does not contain binaries does NOT need a debug-package
we set up MULTI_PACKAGES = -a -b -c bsd.port.arch.mk generates BUILD_PACKAGES by possibly trimming it according to pseudo-flavors and NOT_FOR_ARCHS/ONLY_FOR_ARCHS.

so we just need to set DEBUG_PACKAGES = ${BUILD_PACKAGES}.

A small piece of code in bsd.port.mk will strip PKG_ARCH=*
When we update a port, we need to run update-plist, but in order to do that, we need for fake to finish...

Current "wedge" for debug packages looks like this:

- set `DEBUG_PACKAGES=${BUILD_PACKAGES}` (trimmed through bsd.port.arch.mk)
- trim it through `PKG_ARCH != *`
- if it's not empty use possible `DEBUG_CONFIGURE_ARGS` during configure, build as usual
- at the end of "fake", we run an extra `_copy_debug_info` target
- that target runs `build_debug_info`
- ... and then either links `.dbg` or create `.dbg` through `objcopy`
- make package iterates over normal subpackage with `pkg_create`...
  if there is a debug-subpackage, we also call `pkg_create` on the sly for the debug subpackage
  so make fake can't fail, we just warn in case of issues.
We may run build-debug-info during make package
but it doesn’t extract the debugging information, it creates a Makefile that does that!
so make package depends on that Makefile, and it depend on fake being finished.
and each file is handled independently to extract debug info just once
Hindsight is 20/20. It was obvious we might debug-info extraction to be applied several times, so doing that with real dependencies solves our problem.
For instance I

# Makefile generated by build-debug-info $OpenBSD: build-debug-info,v 1.38 2020/12/04 15:04:52 sthen Exp $

# No serviceable parts

# Intended to run under the stage area after cd ${WRKINST}

OBJCOPY_RULE = ${INSTALL_DATA_DIR} ${@D} && \\
perm=`stat -f "%-p" $?` && chmod u+rw $? && \\
echo "> Extracting debug info from $?" && \\
if readelf 2>/dev/null -wi $?|cmp -s /dev/null -; then \\
   echo "Warning: no debug-info in $?"; \\
fi && \\
objcopy --only-keep-debug $? $@ && \\
${DWZ} $@ && \\
strip -d $? && \\
objcopy --add-gnu-debuglink=$@ $? && \\
chmod $$perm $? && \\
touch $@
For instance II

```bash
LINK_RULE = ${INSTALL_DATA_DIR} ${@D} && \n    echo "> Link debug info from $? to @$" && ln $? $@

all:
  .PHONY: all

all: /vide/build/usr/ports/pobj/libarchive-3.5.1/fake-amd64/usr/local/bin/.debug/bsdcat
    @${OBJCOPY_RULE}

all: /vide/build/usr/ports/pobj/libarchive-3.5.1/fake-amd64/usr/local/bin/.debug/bsdcpio
    @${OBJCOPY_RULE}

all: /vide/build/usr/ports/pobj/libarchive-3.5.1/fake-amd64/usr/local/bin/.debug/bsdtar
    @${OBJCOPY_RULE}
```
For instance III

all: /vide/build/usr/ports/pobj/libarchive-3.5.1/fake-amd64/usr/local/lib/.debug/libarchive.so.11.2

@${OBJCOPY_RULE}
Workflow for people

Old process
- run fake (which generates SOME debug info which may or may not be accurate).
- Run update-plist (which might invalidate the meta info necessary for which file to debug),
- so needs to make `clean=fake` before packaging.

New process
- make fake (no debug info involved)
- update-plist
- create debug info
- and package.
If we did the debug info by accident, we can run it again and again WITHOUT needing to wipe the fake stage, because it’s a Makefile!

(actually \_copy-debug-info is re-run doing EACH packaging step: it depends on the fake dir being up-to-date and the generated debug Makefile, and it just rechecks all .dbg files are accounted for)

Since the debug Makefile doesn’t have "simple" dependencies, we just wipe it at the end of update-plist.
And here is the full process in a nutshell. Highlights:

- need to declare which packages we want debug stuff for (so that the repo doesn’t grow too much).
- declaring is (mostly) \( \text{DEBUG\_PACKAGES} = \{\text{BUILD\_PACKAGES}\} \) get stripped automatically)
- configuring will automatically handle INSTALL\_STRIP/DEBUG in most cases, plus adding \( \text{DEBUG\_CONFIGURE\_ARGS} \) to \( \text{CONFIGURE\_ARGS} \). In a few ports, a bit more glue will be needed.
- the whole magic happens after fake... we use the existing packing-lists
- the fake data is used to generate plists for the debug-packages AND run objcopy
- the debug packages are "fantom" packages that depend on the main package, no independent registration.
- dwz (dwarf compress), imported and maintained by bcallah@, in order to make the debug packages a bit smaller.
- So every debug package BUILD_DEPENDS on devel/dwz, except for devel/dwz which uses the just build binary (and for mozilla which has weird debug info ??? or dwz which is subtly broken)
- Turns out to be a really nice stress-test for debug-info!
The pkg_add side

- initial usage was simply to manually pkg_add debug-* stuff
- SHEARING

- run into a bug on an installed package, and find out the debugging package is obsolete: debug info has a kind of signature which must match the debugging binary EXACTLY.
Two ways around that

- add an option (-d) to `pkg_add` to (silently/automatically) install/update debug packages when available (a bit of a size hog)

- configure `pkg_add` through an env variable to keep a stash of debug packages for installed packages (`DEBUG_PKG_CACHE`), so that you always have an up-to-date debug package.
open issues

- some frameworks make things more complicated (cmake!)
- how do the debug options change things?
- what platforms want debug packages
That’s about all. Many thanks to my fellow developers!