Custom **Poudriere Packages** in Your Own Repository

BY BENEDICT REUSCHLING

I'm forever grateful for the people who made ports and package installations on FreeBSD so easy. Whereas other Unix-like systems need to either manually install a bunch of libraries or dependencies, let alone package origins as text files, the BSDs typically don't need any of that. A simple **`pkg install`** foo does all that for me with the end result of having foo installed. Those pre-built packages come from the official FreeBSD package distribution system, and they have been configured with default options that serve most default cases.

In my workplace, unfortunately, the defaults don't fit us. What we need is LDAP support to query our central database to perform authentications. A lot of ports provide LDAP support via "make config", but the packages based on them do not have that option enabled by default. So, when I want to install PostgreSQL 15 with LDAP support, I can't install it via **`pkg install postgresql15-server`**, as the installed binaries have no clue about LDAP authentication in the database.

A first solution is to build my own custom packages. I fetch the latest ports tree with

```
# portsnap auto
```

Then I navigate into the directory of the port in question, which is **`/usr/ports/databases/postgresql15-server`** and run

```
# make config-recursive
```

Then, I carefully select any option that relates to LDAP. If that is too tedious because there are too many dependency packages to go through, I can also add

```
OPTIONS_SET=LDAP
```

to my **`/etc/make.conf`**
which will then automatically select this option (if available) for any ports I build. Then, I can run

```
# make package
```

and wait for the package to be built with those custom options. The resulting package can be found in the **`work/pkg`** subdirectory and installed via **`pkg install ./packagename`**

This is all fine and good, but what about more than one package like this? Or, what if you want to always have the latest package version available to install, but don't want to bother doing that on each individual box you manage? Sooner or later, people want to build their own packages automatically and make them available in their own networks. For those cases, poudriere was developed.

Poudriere (French for powder keg) is a framework to build customized packages in jails, resolve dependencies between them and make them available as a FreeBSD repository.

The ports developers use it to test new versions of ports (hence the powder keg analogy, as such builds can sometimes explode or simply fail). You don't have to be a ports developer or understand any of the building blocks to use poudriere. The reason why each port (even the tiniest dependency) is built in a separate jail (created and deleted automatically) is to have a clean build-environment each time. It also enables parallel building of ports, which is highly beneficial considering the number of packages to build that may not be immediately obvious (dependencies may build other dependent ports first and down the rabbit hole it goes).

You need to have a FreeBSD machine for building those custom packages, ideally with a lot of CPU and memory and good I/O. Poudriere stresses a system because it tries to run a lot of those package builds in parallel, which is why it can also be seen as a system benchmark tool.

Let's get started creating our own poudriere machine. I call my buildbox poudriere (because I'm creative today) and distinguish it from other commands in this article by the prompt: **`poudriere#`** in front.

Poudriere Setup

First, install the poudriere package. There is a **`poudriere-devel`** version which has some fixes that are not yet contained in the regular version. My own experiences have been good so far with both, so I chose this version:

```
poudriere# pkg install poudriere-devel
```

We're taking a closer look at poudriere's config file, located in **`/usr/local/etc/poudriere.conf`** after the installation. We make a couple of changes in there, depending on our system resources and whether or not we are using ZFS (strongly recommended). The comments in this file will help you understand what these options do:

```
ZPOOL=zroot
FREEBSD_HOST=ftp://ftp.freebsd.org
RESOLV_CONF=/etc/resolv.conf
BASEFS=/usr/local/poudriere
POUDRIERE_DATA=/usr/local/poudriere/data
USE_PORTLINT=no
USE_TMPFS=yes
MAX_FILES=2048
DISTFILES_CACHE=/usr/ports/distfiles
CHECK_CHANGED_OPTIONS=verbose
CHECK_CHANGED_DEPS=yes
CCACHE_DIR=/var/cache/ccache
PARALLEL_JOBS=65
PREPARE_PARALLEL_JOBS=5
ALLOW_MAKE_JOBS=yes
KEEP_OLD_PACKAGES=yes
KEEP_OLD_PACKAGES_COUNT=3
```

You need to provide the name of the ZFS pool that poudriere should use. ZFS can clone the build jails quickly and remove them, rather than having to copy them from a base jail for each port. Create the necessary datasets that were defined in **`POUDRIERE_DATA`**, **`DISTFILES_CACHE`** and **`CCACHE_DIR`** if they don't exist yet. Adjust the values for **`MAX_FILES`**, **`PARALLEL_JOBS`** and **`PREPARE_PARALLEL_JOBS`** to fit your system. I recommend to start low at first and then increase them after your first couple poudriere builds. If you max out

these values, a poudriere build may bring your system to its knees, so keep a bit of resource for other tasks (i.e. your ssh login session).

Ccache Setup

This part is entirely optional and poudriere will run just fine without it. It's merely an optimization to speed up future builds. With ccache, compiled binaries are cached (hence the name, compiler cache) and if they have not changed, the cached version is used. This results in enormous build speedups for consecutive builds as compile times are reduced, in some cases dramatically. We've already told poudriere to use ccache in the config file, but we need to install and configure ccache first. If ccache is missing, poudriere will run fine, but won't make use of the compile caching.

Before we run the **`pkg install`** command, we create a separate dataset, as the pkg would otherwise create a directory for it.

```
poudriere# zfs create \
-o mountpoint=/var/cache/ccache \
-o compression=lz4 \
-o recordsize=1M
zroot/var/cache/ccache
```

With those options, I could reduce the on-disk size of the cache by 1/3, but it highly depends on what packages you build and how often.

Let's now install the ccache package:

```
poudriere# pkg install ccache
```

Next, let's edit the ccache config file. It resides (or rather, is searched) in many different directories. We'll create one reference file and simply link to it from the other locations to ease the maintenance burden. Luckily, you'll rarely touch this file, but if you do, the symbolic links ensure each location is changed with it.

```
poudriere# cat << EOF > /var/cache/ccache/ccache.conf
max_size = 0
cache_dir = /var/cache/ccache
base_dir = /var/cache/ccache
hash_dir = false
EOF
```

The **`max_size`** option can limit the cache size to a certain amount, but with 0, it can use all the disk space it requires. I don't worry too much about it, since the ZFS compression is doing nice work here. I can even set a quota on the **`zroot/var/cache/ccache`** dataset if disk space got low.

The **`cache_dir`** and **`base_dir`** define the location of the cache. In our case, they point to our dataset. The **`hash_dir`** option set to false with increase cache hits, but having it activated makes debugging difficult. That's a tradeoff I'm willing to take for better performance. Consult **`ccache(1)`** for details on this and other options. It's discussed in a FreeBSD forums thread: https://forums.freebsd.org/threads/howto-speeding-up-poudriere-build-times.69431/

Let's create the symlinks to this config file in the locations ccache expects to find it.

```
poudriere# ln -s /var/cache/ccache/ccache.conf /root/.ccache/ccache.conf
poudriere# ln -s /var/cache/ccache/ccache.conf /usr/local/etc/ccache.conf
```

That's it already. You can check the status of the cache using

```
poudriere# ccache -s
```

after you've done a couple of builds. Let's return to poudriere now...

Poudriere Configuration

We've already mentioned that poudriere will run jails for each individual port that makes up a package. To do that, poudriere needs to know for which architecture and which FreeBSD release the packages should be built. There are subtle differences between versions, but poudriere can build different versions side by side without them interfering with each other. The jails are kept separate from each other. In addition, you can build packages for your Raspberry Pi on your beefy amd64 server this way.

Let's start with a jail for amd64 and FreeBSD 13.2, since that is the current version at the writing of this article.

```
poudriere# poudriere jail -c -j 132x64 -v 13.2-RELEASE
```

With the **-c** option, we ask poudriere to create a new base jail for the builds and provide the architecture and version that should be used. You can even build packages for unsupported FreeBSD versions (i.e., FreeBSD 12.1) by adding the **-m ftp-archive** option.

You can list the available builder jails with this command:

```
poudriere# poudriere jails -l
JAILNAME   VERSION      ARCH           METHOD TIMESTAMP           PATH
132x64     13.2-RELEASE amd64          http   2023-05-03 07:54:58 /usr/local/poudriere/
                                                                  jails/132x64
```

Your output may be different, and you can give the jail any name you want. I encourage you to include the version and architecture to distinguish it from any other poudriere jails you may have.

A ports tree is required from which to base the packages. It's a straightforward command like:

```
poudriere# poudriere ports -c -p default
```

Again, we can list some details about this ports tree:

```
poudriere# poudriere ports -l
PORTSTREE METHOD    TIMESTAMP           PATH
default   git+https 2023-10-01 12:00:02 /usr/local/poudriere/ports/default
```

We could also have multiple ports trees available. Maybe we want to have a slower pace in updates and only build the ports from the last quarter instead of the latest ones. All possible with poudriere and the **-p** option.

What we need now is a list of packages for poudriere to build. "Oh, I like that port, and I always install this shell, with that editor. And **tmux...**" This may turn up a list quickly, but it won't contain dependencies (packages needed to run or build those listed). Poudriere takes care of resolving those for you.

Put it in **/usr/local/etc/poudriere.d/** as a text file (again, any name you like) with each port and it's category on a separate line.

My list of ports looks like this (and is ever growing the more I start appreciating poudriere):

```
poudriere# cat /usr/local/etc/poudriere.d/pkglist.txt
databases/postgresql15-server
databases/postgresql15-client
databases/postgresql15-contrib
```

These are the ports that need LDAP support, remember? You can start with something simpler, like **games/sl** or **shells/fish** to not spend too much time waiting for poudriere to finish building.

A better way is to let the package system create a list of packages. Log into the system where you have all this software installed and run **pkg_info**. This creates a list that includes the dependencies. Quite a list already! You can always cut it down in case you wanted to delete a package anyway. A package may also be too big to build (libreoffice comes to mind).

Defining which ports to build does not yet configure them. We still need to tell poudriere which options to set for each port. But we need to do this only once and for future builds, poudriere will save our selected options and reuse them, even for future versions of the package. This corresponds to running "make config" in the ports directory as we did at the beginning of the article.

```
poudriere# poudriere options \
-j 132x64 \
-p default \
-f /usr/local/etc/poudriere.d/pkglist.txt
```

This defines options for the whole list of ports. You can also do it for individual ones with this invocation:

```
poudriere# poudriere options \
-j 132x64 \
-p default \
databases/postgresql15-server
```

That completes the preparations for poudriere. We can now start our first package build.

Building and Distributing Packages

To initiate a package build, provide the jail, ports tree and the list of packages to the bulk subcommand of poudriere:

```
poudriere# poudriere bulk \
-j 132x64 \
-p default \
-f /usr/local/etc/poudriere.d/pkglist.txt
```

Sit back and enjoy the automated package build. You can press CTRL-T to see an intermediary output of the build status. I recommend you run this in a **tmux** session so that you can detach from a long build and get back to it later without stopping the whole process when you exit from the shell.

After the build is finished, poudriere will tell you that it has created a repository that contains the packages from your list. To make use of this repository, we need to tell our FreeBSD about its existence by defining a new repository location.

```
poudriere# mkdir -p /usr/local/etc/pkg/repos
```

A file called **local.conf** (again, could be any name, see the theme here?) contains my own repository configuration.

```
poudriere# cat /usr/local/etc/pkg/repos/local.conf
Poudriere: {
        url: "file:///usr/local/poudriere/data/packages/132x64-default",
        priority: 23,
}
```

My repository is called Poudriere (again: your own name, you know the drill) and I defined its location in the url: part. I got the location from poudriere's build output. The priority defines the order in which these repositories are used. By default, the upstream FreeBSD.org repository is used. But since we want our own packages to take precedence, simply give it a higher priority than zero to make it use our own package repository first.

You can also disable the FreeBSD repository completely by adding

```
FreeBSD: {
   enabled: no
}
```

but you can also run both side by side at first.

Check which repos are used with the output of

```
poudriere# pkg -vvv
```

The bottom part should contain our own repository definition.

Let's run pkg update:

```
Updating Poudriere repository catalogue...
Fetching meta.conf: 100%    163 B   0.2kB/s    00:01
Fetching packagesite.pkg: 100%   68 KiB  69.8kB/s    00:01
Processing entries: 100%
Poudriere repository update completed. 232 packages processed.
All repositories are up to date.
```

Notice the number of packages. This is definitely our own local repository as the main FreeBSD repository contains over 31500 at this point, whereas this only has 232. These are the packages that we built based on the list in our own pkglist.txt plus the dependencies to make those packages build and run. Another way to view available repositories is via pkg stats, which produces this output on my system:

```
Local package database:
        Installed packages: 120
        Disk space occupied: 2 GiB

Remote package database(s):
        Number of repositories: 2
        Packages available: 34190
        Unique packages: 34190
        Total size of packages: 117 GiB
```

In this instance, I left the FreeBSD: entry in **/usr/local/etc/pkg/repos/local.conf** at

```
FreeBSD: {
   enabled: yes
}
```

Now both repositories are used, but the local one is preferred because of the higher priority. If the package is not in my local repository, the other repos are checked based on their priority. In this case, we fall back to the official repository if all else fails.

```
poudriere# pkg upgrade
Updating FreeBSD repository catalogue...
FreeBSD repository is up to date.
Updating Poudriere repository catalogue...
Fetching meta.conf: 100%    163 B   0.2kB/s    00:01
Fetching packagesite.pkg: 100%   73 KiB  75.0kB/s    00:01
Processing entries: 100%
Poudriere repository update completed. 249 packages processed.
All repositories are up to date.
Checking for upgrades (40 candidates): 100%
Processing candidates (40 candidates): 100%
The following 1 package(s) will be affected (of 0 checked):

New packages to be INSTALLED:
        gdbm: 1.23 [FreeBSD]

Number of packages to be installed: 1
209 KiB to be downloaded.

Proceed with this action? [y/N]:
```

You can always tell where a certain package is coming from as the repository is listed in brackets behind it. The gdbm package is coming from the official FreeBSD repository. Be careful with this mixture, though. In some cases, your custom-built packages and those from the official FreeBSD repo may not work together, as you may have removed some options that other packages require. Test carefully or decide to go full poudriere. In that case, set

```
 FreeBSD: {
   enabled: no
}
```

or remove that repo definition from **/usr/local/etc/pkg/repos/local.conf** altogether.

What about the rest of your ever-growing fleet of machines running FreeBSD? Especially for VMs or embedded systems, you wouldn't run a separate poudriere builder on each of them. One way could be to share the repo URL via NFS to each machine. A better approach is to configure a central, beefy poudriere build machine to share its packages via http as a repository. Other FreeBSD machines in your network can then add it just like the official FreeBSD repository. The added benefit of that compared to the NFS share is that you can sign those packages. This ensures cryptographic integrity and trust in the source (those packages are really those you custom-built). That way, machines check whether the public keys of the build machine match the records they have to ensure the packages are coming from a genuine source. Let's set this up next.

To serve packages to other machines, we install nginx as the webserver. Other webservers also work if they can share a single URL as a document root to clients. We can also share

the build-logs during a "poudriere bulk" run to see the progress of the current build and debug those ports that failed.

```
poudriere# pkg install nginx
```

Of course, this nginx could also come from our own local repository if we wanted to make any changes to its default package configuration. Note: we're not encrypting the traffic itself with SSL, but simply the package repository itself. Adding SSL for the transit encryption is left as an exercise to the reader, but it's not complicated.

After editing **/usr/local/etc/nginx.conf**, it should have these sections in it:

```
server {
  listen 80 default;
  server_name domain.or.ip.address;
  root /usr/local/share/poudriere/html;

  location /data {
    alias /usr/local/poudriere/data/logs/bulk;
    autoindex on;
  }
  location /packages {
    root /usr/local/poudriere/data;
    autoindex on;
  }
}
```

After saving and exiting the **nginx.conf**, we need to make one change to make our build logs also appear properly in the browser. That means, when a log file (.log extension) gets opened, it will not be offered as a download, but, instead, displayed in the browser right away. To do that, we visit **/usr/local/etc/nginx/mime.types**. Find the line that starts with **text/plain** and change it to include log files as well:

```
text/plain                                txt log;
```

Enable nginx to start during system reboot by doing an entry for it in **/etc/rc.conf** via

```
poudriere# service nginx enable
```

Start the service now:

```
poudriere# service nginx start
```

You can find your build status and logs by pointing your browser to http://server_domain_or_IP.

For the repository signature, we create a couple of directories for the keys and certificates.

```
poudriere# cd /usr/local/etc/ssl

poudriere# mkdir keys certs
```

The keys should not fall into anyone else's hands, so we only allow the root user to poke into this directory:

```
poudriere# chmod 0600 keys
```

Next, we start generating the RSA private key for the poudriere repository.

```
poudriere# openssl genrsa -out keys/poudriere.key 4096
```

Handle that key just like the physical key to your front door: never give it away or let anyone else even see it. If the key is compromised, an attacker could inject any kind of packages into your machines and that will make your day a very unpleasant one.

Next, we create a certificate (public key) based on this private key we just generated:

```
poudriere# openssl rsa -in keys/poudriere.key -pubout -out
certs/poudriere.cert
```

All necessary keys are now available. Next, we need to make poudriere aware of them so that packages being built are signed with them. This is done in **/usr/local/etc/poudriere.conf** which we visited earlier for our first basic poudriere configuration. Find the line starting with **PKG_REPO_SIGNING_KEY** and give it the location of the private key we generated. The end result looks like this:

```
PKG_REPO_SIGNING_KEY=/usr/local/etc/ssl/keys/poudriere.key
```

Another line we want to change for some additional clickable links is the following:

```
URL_BASE=http://my.domain.or.IP
```

The website defined by **URL_BASE** shows a lot of statistics about past and current package builds. During a build, it will refresh itself to show the build status. You can also drill down into each build to see which ports have successfully built, were skipped, or ignored. Very nice!

After adding those lines, poudriere is ready to sign new built packages. And, indeed, after the next

```
poudriere# poudriere bulk \
-j 132x64 \
-p default \
-f /usr/local/etc/poudriere.d/pkglist.txt
```

run, I see these new lines in the output:

```
[132x64-default] [2023-08-06_09h24m25s] [load_priorities:] Queued: 0  Built: 0  Failed:
0  Skipped: 0  Ignored: 0  Fetched: 0  Tobuild: 0   Time: 00:00:08
[00:00:08] Recording filesystem state for prepkg... done
[00:00:08] Creating pkg repository
[00:00:09] Signing repository with key: /usr/local/etc/ssl/keys/poudriere.key
Creating repository in /tmp/packages: 100%
Packing files for repository: 100%
[00:00:13] Signing pkg bootstrap with method: pubkey
[00:00:13] Committing packages to repository: /usr/local/poudriere/data/packages/
132x64-default/.real_1691306678 via .latest symlink
[00:00:13] Removing old packages
```

Time to revisit **/usr/local/etc/pkg/repos/local.conf** and add the certificate next to the signature type. The file should look like this after the edits:

```
Poudriere: {
        url: "file:///usr/local/poudriere/data/packages/132x64-default",
        priority: 23,
        mirror_type: "srv",
        signature_type: "pubkey",
        pubkey: "/usr/local/etc/ssl/certs/poudriere.cert",
        enabled: yes
}
```

Client Configuration

Now it is time we add another machine to use our signed package repository. Log into the FreeBSD that you want your custom, freshly built packages on and create the directory for the poudriere certificate and the repo configuration.

```
clienthost# cd /usr/local/etc/
clienthost# mkdir ssl/certs ssl/keys
clienthost# mkdir pkg/repos
```

Then, securely copy the poudriere.cert from **/usr/local/etc/ssl/certs/** on the build machine into that directory we just created. You can use **scp(1)** for the transfer from the host to the client:

```
poudriere# scp /usr/local/etc/ssl/certs/poudriere.cert
clienthost:/usr/local/etc/ssl/certs/
```

Next, we define a new repository location like the above. The only difference is in the URL. Whereas the build machine could use **file:///** to reference the local file system, remote machines will have to connect via http to our nginx. The **mirror_type** is also different, but other than that, it's the same configuration as follows:

```
clienthost# sudoedit /usr/local/etc/pkg/repos/freebsd.conf

Poudriere: {
        url: "http://my.domain.or.ip/packages/132x64-default",
        mirror_type: "http",
        signature_type: "pubkey",
        pubkey: "/usr/local/etc/ssl/certs/poudriere.cert",
        priority: 23,
        enabled: yes
}
```

```
clienthost# pkg update
pkg update
Updating FreeBSD repository catalogue...
FreeBSD repository is up to date.
Updating Poudriere repository catalogue...
[clienthost] Fetching meta.conf: 100%    163 B   0.2kB/s    00:01
[clienthost] Fetching packagesite.pkg: 100%   73 KiB  75.0kB/s    00:01
Processing entries: 100%
Poudriere repository update completed. 247 packages processed.
All repositories are up to date.
```

The line telling me about 247 packages processed must mean that it could find my other repository and the packages within. Running a

```
clienthost# pkg upgrade
```

confirmed that it works since it was referencing my own custom repository name:

```
Updating FreeBSD repository catalogue...
FreeBSD repository is up to date.
Updating Poudriere repository catalogue...
Poudriere repository is up to date.
All repositories are up to date.
New version of pkg detected; it needs to be installed first.
The following 1 package(s) will be affected (of 0 checked):

Installed packages to be UPGRADED:
        pkg: 1.19.1_1 -> 1.20.5 [Poudriere]

Number of packages to be upgraded: 1

The process will require 2 MiB more space.
9 MiB to be downloaded.

Proceed with this action? [y/N]:
```

It works! The package was downloaded from the build machine and uses the custom configuration that I made for the package. It feels snappy and satisfying to know that I don't have to rebuild llvm on every underpowered FreeBSD VM that I run. That's what beefy build servers are for.

Here is the output from a pkg upgrade with a mixed FreeBSD and custom repository:

```
New packages to be INSTALLED:
        cyrus-sasl: 2.1.28 [Poudriere]
        gdbm: 1.23 [FreeBSD]
        icu: 73.2,1 [FreeBSD]
        openldap26-client: 2.6.5 [Poudriere]
```

Be aware though that mixing those packages may lead to some headaches due to the way these two interact. A package that expects another package to have a certain option set, but is not part of the particular package in the other repo, may cause build or install errors, leading to applications not working as expected. Ideally, one would use a single repository source, either upstream or their own internal one.

Also make sure that if you are building the packages for the "latest" branch and distribute those to your clients, that those clients also are using latest in **`/etc/pkg/freebsd.conf`**. Otherwise, the packages are newer than quarterly. I had one instance where the packages above were happily updated and installed, but once I ran "pkg autoremove" they were listed for removal again. Then I could upgrade them again and the vicious circle was complete.

When you simply add the repository to a machine without the certificate, pkg update will complain about it missing:

```
clienthost# pkg update
Updating FreeBSD repository catalogue...
```

```
FreeBSD repository is up to date.
Updating Poudriere repository catalogue...
Fetching meta.conf: 100%    163 B   0.2kB/s    00:01
Fetching packagesite.pkg: 100%   76 KiB  77.6kB/s    00:01
pkg: openat(/usr/local/etc/ssl/certs/poudriere.cert): No such file or directory
pkg: rsa_verify(cannot read key): No such file or directory
pkg: Invalid signature, removing repository.
Unable to update repository Poudriere
Error updating repositories!
```

One last thing you can do on your build server is create a cron job to update poudriere's ports tree:

```
poudriere# poudriere ports -u -p default
```

We use **-u** this time instead of **-c** to indicate that we want to update an existing ports tree. We could do this once a day or even earlier, depending on how fresh you want your packages to be. Then, let poudriere run the bulk build:

```
poudriere# poudriere bulk \
-j 132x64 \
-p default \
-f /usr/local/etc/poudriere.d/pkglist.txt
```

I run this nightly, so that in the morning, I have freshly built packages waiting for me. I can always check the build status using the poudriere web site that nginx provides.

Poudriere really shines when it builds a lot of different packages for different architectures. It uses qemu to build packages for non-amd64 architectures like my Raspberry Pi. Since all is done in parallel, I can get my own packages built much faster, while the rest can come from the default FreeBSD repository—as I don't need special options set on those. Over time, the list of your own custom packages will certainly grow. The same goes for the number of machines that use this repository provided with your own certificate. You're in total control of when and which packages you update and you can even help build new ports with Poudriere.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.