Divorce Your Linux Admin

Package Management For Lusers

If you run Linux on your own machines, you're used to having `root` and doing what you jolly well like. But, if you've ever spent more than about 10 minutes in a large coporate IT environment, you learn pretty quickly that `root` is hard to get, it takes a ton of paperwork to get anything done, and you usually have to wait forever. I've actually had the experience of waiting for 6 weeks to get permission to install a single symlink ... *and I had* `root`!

There is a good reason for this, of course. Security threats are very real, lawsuits are ominipresent, and the Geniuses In Charge (tm) are writing regulation and audit compliance rules that make root canals seem like fun. Information Security people may feel like they are the IRS of the business, but they perform an important and necessary task: Saying "No".

So ... is there a better way? Is there a way to eliminate the requirement for `root` in most day-to-day things we need to do as users and developers. Is there a way we can comply with the required corporate security constraints, but still run our own happy show? The answer is a qualified "Yes".

Some things do- and always will need `root`: Managing devices, storage, ulimits, and security configuration leap to mind. But, say, all you want is a newer version of `java` on your servers. Or suppose you want a package that isn't part of your standard OS load. `vi` is everywhere, but suppose you want to use `emacs` instead (as you should).

You could, of course, download the source for the programs you want, configure and compile them, and run them, say, out of your home directory. Oops ... standard IT corporate security practice is to never allow a compiler to exist on a production host. There are ways around this, but it's fairly painful to have to do that for every single package you may want. (If you don't think so, I encourage you to try and bootstrap the `gcc` compiler chain from scratch. It's a ton of fun. No, really, it is ...)

Wouldn't it be nice if we could implement package management in userland in a way that is repeatable, can be automated, and gives us control of our own universe without having to beg for `root` changes or have to wait for the vendor to release a new version. Well, Sparky, we have the techology to do just that.

It's worth mentioning that the approach outlined below is especially handy with cloud and on-demand computing. It makes automating a standard toolset deployment pretty simple. It's also actually handy on your own machines when you *do* have `root`. The less you use superuser, the less chance you'll screw something up.

> **Warning**
>
> What follows has been implemented on an experimental basis. It's been tested in only a very limited number of systems but seems to work well. However, you should do your own detailed testing before deploying this into a production environment. Failure to do so may result in broken systems, hallway snickering, hives, and being transferred to your new development shop in Moose Dropping Pass, Alaska.

MacOS Rescues Linux

The approach we're going to describe got started in the Mac OSX world. Back when Apple finally came to their senses, and switched their OS to a Unix-base (FreeBSD 4.4), they only partly implemented the

shell tools everyone had come to know and love. The `homebrew` project got spun up to allow any OSX user to install the command line applications they knew from Unix. `homebrew` is essentially a userland package management system which can be run and modified without superuser power. Many of these packages (these days, perhaps all, I haven't checked) actually download a pre-compiled version under `/usr/local`.

This ended up being pretty popular with advanced Mac users. So much so, that a derivative project, `linuxbrew`, got spun up to take the Mac stuff, but apply it to Linux. That is, give the Linux user userland package management system. It too, has found success among the Linux literati.

But ... there is a fly in the ointment. When I first undertook this project, I thought I could just pick a directory on Linux machine and use `linuxbrew` to install what I wanted. *No habla Senor Frog.* Many Linux binaries are sensitive to where they are installed, where they can find their supporting libraries and a host of other things. So, if I install a binary with `linuxbrew` somewhere other than the default `/home/linuxbrew`, it's likely not going to work. But that ability is exactly what I needed. Each different application, user, or service ID should be free to install their desired tool set wherever they wish.

"So", sez me, "I'll just use `linuxbrew` to automate the download, configuration, compliation, and installation of all the packages." i.e., "I'll automate the build from source." (That roaring laughing you hear is coming from every Linux engineer who ever tried something like this.)

I will spare you sensitive readers the subsequent cursing, whining, begging, crying and caterwauling that ensued. Let's just say that making a position-dependent package management system work in a position-independent way is ... er, non-trivial. In fairness, it's not the fault of the `linuxbrew` people. They were super supportive and helpful with all this. It wasn't their code that was the problem (mostly, I did find a minor bug or two which the `linuxbrew` folks fixed at light speed). Most of the issues had to do with the packages themselves having embedded assumptions about where they can find tools during the compilation phase. That's right, the *source code and configurations* have hardwired assumptions about where they would find things like `perl`.

At this point, the whole process had taken me a few dozen hours and I was sufficiently enraged that I just *had* to figure out. As we'll see shortly, I think I finally got there. But, in the mean time ...

> **Note**
>
> If you write software, config files, makefiles, test cases, or any part of the software delivery ecosystem *with hardwired paths to things emebedded in them*, you are officially a big bozo. Not the fun kind with a red nose and big shoes either. The *only* hardwired path that's OK is `/bin/sh` on a shebang line. But if you do things like this:
>
> ```
> #!/usr/bin/python
> ```
>
> You should be sent to work 1st level phone support on the midnight shift in Somalia until you learn better. Grrrrrr.
> This is the right way to do this is:
>
> ```
> #!/usr/bin/env python
> ```
>
> `env` can reliably be found there and it will "discover" where `python` happens to actually be installed on that machine, so long as it is in `$PATH` somewhere. Similarly, learn to use constructs like:
>
> ```
> DATE=$(which date)
> DATE=${DATE:-/bin/date}
> ```
>
> In short, **NEVER make assumptions where things are**. Always discover it at configuration time.

Preview Of Coming Attractions

What I eventually discovered was that getting this to work required a number of things:

1) Everything has to be built from source *in the directory location being targeted.* The only exception is the `brew` program itself, which is position agnostic. So, if I want to build

a tools tree under `/my/fine/tools`, then I have to clone `linuxbrew` into that directory and do the build from there.

2) The initial build requires the OS compiler chain and related development tools to bootstrap up a minimal `linuxbrew` environment capable of compiling everything else. You can do this on your own machine (not recommended because you shouldn't be fidding around as root there), but a better way is to do it in a VM. In my case, I made it even simpler by doing everything in `docker` containers.

3) Once you have a bootstrapped `linuxbrew` environment running - i.e., One that has a functioning `gcc` and supporting tool chain - you make a `tar` backup of it. You then untar that onto a machine that has (almost) no native OS development tools on it and do the remainder of the installations from there.

   It's "almost" because of the aforementioned dain bramaged open source packages hardwiring of paths. For example, you *have* to have the OS copy of `perl` installed on your build machine. Many open source packages just *insist* that `perl` is always to be found under `/usr/bin`.

4) When you're all done installing and configuring your `linuxbrew` environment, you just `tar` it off somewhere safe. You can then untar it onto any other Linux machine (with a reasonably similar kernel) so long as you do so at the *same directory location under which it was built*.

   This lends itself nicely to automated deploys via tools like `tsshbatch` or `ansible`. You build a master tarball of your "standard" tools tree and then use automated deployment to put it everywhere.

Doing It The `docker` Way

Like I said, you can do this in a VM, but the step-by-step approach below uses `docker` containers which are easy to setup and tear down for testing. More importantly, you can install and remove native system packages as you go without gumming up your host system. I've used this approach extensively over the past several years for another important reason: *I always have root on a container*. That makes it trivial to do the required OS package management (installing- and removing native compilers, for example).

In my test environment, the containers have a number of properties. You don't have to do it this way, of course, but it makes things a lot simpler if you do:

- They run sshd so I can log into them easily from the host system.
- I have the ability to log in as an unprivileged user (`test`) or as `root`. `test` also has the ability to `sudo` to superuser.
- They share a filesystem with the host so that I can read/write files from any running container AND the files I do write persist across container rebuilds.
- The containers get started with the `--security-opt seccomp=unconfined` option. Building `emacs` revealed the need for this. By default, `docker` starts containers with restricted access to many of the host OS system calls. It does so in order to keep the container isolated from its host environment. But this badly broke the `emacs` build which had fits because the way the OS was allocating memory. The fix is to use the above argument to give the container full access to all the system calls. You do *not* want to do this in normal container operations. This is strictly for building things. More information on this here:

  https://pastebin.tundraware.com/view/e309f836

Let's Do This Already

- The steps below should be done *in the order given.*
- Whenever it says, "create a new `docker` container", do so with the `--security-opt seccomp=unconfined` option.
- This example assumes CentOS/RedHat7 docker images, so the native package management commands are based on `yum`.
- This example assumes that we want to build our tools tree under `/opt/TundraWare/tools`. Feel free to use your own directory location, but do so consistently throughout the process.
- Make sure the user doing all this has write access to your target tools directory.
- This example assumes that `/shared` is common to both the host and docker instances. It's where we'll preseve our tarballs and other project artifacts across container rebuilds.

Procedure

> **Note**
>
> If you're doing this behind a proxy, you may need to configure things to get around man-in-the middle madness introduced by many enterprise proxies. Things like `curl` need to verify certificates to initiate TLS transport. System `curl` knows how to find the OS trust store, but the `curl` we build with this procedure has its own trust store that needs to be made aware of your CA chain.
> This procedure creates new `openssl` instances (1.0 and 1.1) they need to be made aware of your system's CA chain right after they are installed. The existing `.../tools/etc/openssl/cert.pem` and `.../tools/etc/openssl@1.1/etc/cert.pem` should be moved to a backup name or location. Then these names should be symlinked to the CA chain found in your system trust store. On a `CentOS` system this is typically found under:
>
> `/etc/pki/....`
>
> You'll have to see how your system is configured to find the CA cert chain you want.
> If you absolutely cannot get this working you can turn off certificate validation *but this is highly discouraged.* By doing so, you can introduce code from illegitimate sources this way:
>
> `echo insecure » /.curlrc git config --global http.sslVerify false`

First, we're going to create the bootstrap instance:

1) Create a new docker image. Log in as or become root on it. Then:

   ```
   yum -y groupinstall "Development Tools"
   ```

2) Now login or revert back to being an unprivileged user. Then:

   ```
   git clone https://github.com/Linuxbrew/brew.git /opt/TundraWare/tools
   ```

3) Do *not* include the tools directories we're about to build in `$PATH`. We only want to use system tools in this phase of the build.

4) Now, bootstrap the environment using the native OS compiler tools:

   ```
   brew install gcc make nload
   ```

   `nload`? Really? Yes. Some packages seem to insist that they only way they'll build is via the system tools. Stuff that depends on `autoconft`, `automake`, and `perl` can be particularly snarky about this. So ... we just add such packages to the bootstrap phase.

5) Tidy up:

```
brew config       # Check the environment
brew prune        # Tidy up from the bootstrap build
brew cleanup      # Get rid of old build artifacts
brew doctor       # Check to make sure things look OK
```

6) Save the result:

```
tar -czvf /shared/bootstrap-linuxbrew.tar.gz /opt/TundraWare/tools
```

Now we can build a freestanding instance of the tools tree without (almost) any OS tools. The example below should be tuned for the packages you want:

1) Create a new docker image. Log in as or become root on it. Then install the minimal set of tools required to accommodate the previously described package build silliness:

```
yum -y install perl autoconf automake
```

Make sure the native development tools are NOT otherwise installed in this instance.

2) Install our bootstrap environment:

```
tar -xzvf /shared/bootstrap-linuxbrew.tar.gz -C /
```

3) Update the path to search our new tools tree for things first. It's a good idea to also put this into `.bashrc` and then copy it to `/shared` for future use:

```
export PATH="/opt/TundraWare/tools/bin:/opt/TundraWare/tools/sbin:$PATH"
```

4) Now we can start installing our desired packages. Note that we are now using *the* `linuxbrew` *compiler chain*, NOT the system tools. Notice also that we want to install our languages before anything else, so that subsequent tools installations will use these rather than the ones present in the OS:

```
brew install perl python
ln -s /opt/TundraWare/tools/bin/python2 /opt/TundraWare/tools/bin/python
ln -s /opt/TundraWare/tools/bin/pip2 /opt/TundraWare/tools/bin/pip
pip install ansible pew pew[pythonz] -U --ignore-installed
brew install emacs file-formula git htop joe nmap screen the_silver_searcher tree vim
... and so on.
```

5) Save the results:

```
tar -czvf /shared/full-linuxbrew.tar.gz /opt/TundraWare/tools
```

Deploying The Tools

You should now be able to install and use this tool tree on a new docker, VM, or physical Linux instance by doing this:

```
tar -xzvf /shared/full-linuxbrew.tar.gz -C /
export PATH="/opt/TundraWare/tools/bin:/opt/TundraWare/tools/sbin:$PATH"
```

If all you want to do is execute the binaries you've just installed, that should be it.

If you also want to be able to install additional packages in the new tools instance, you may need to install system `perl`, `automake`, etc. as mentioned above.

This is not really a great idea, though. The better way is to keep a master tools configuration on a build server, and add- or delete content there as we've just seen. Then build tarballs for distribution via `tsshbatch` or `ansible`. If each tools instance does its own upgrades, when you finally do release a new master update, it's likely going to wipe or or conflict with the local bespoke version.

Gotchas

Here are a few things to keep in mind:

- Some packages are just broken and require surgery to get working. For example, `socat` requires `openssl@1.1`. However, as of this writing, that package will not install, so `socat` won't either. The fix turns out to require a manual installation that disables testing:

  ```
  brew install openssl@1.1 --without-test
  ```

- When you bootstrap the system, you are building it with the OS' own compilers and header files. You're actually producing a compiler that will be used to compile the rest of your packages. If you later copy your work to a machine with a wildly different older- or newer kernel, you may run into compatibility issues. The fix is to redo the above on a host with the kernel version of interest.

Resources

You'll find support for automating most of this work (via a makefile) at:

https://gitbucket.tundraware.com/tundra/tools-builder

The main `linuxbrew` page is:

http://linuxbrew.sh

The related GitHub projects are here:

https://github.com/Linuxbrew

If you run into a problem building a package, run this command:

```
brew gist-logs package-name
```

This produces a Github gist URL you can submit to the devs for help.

Author & Copyright

Tim Daneliuk, tundra@tundraware.com

Document Information

Revision: `a452c97`

Produced On: `Thu Jan 11 15:02:56 CST 2018`

You can find the latest version of this document at:

http://www.tundraware.com/TechnicalNotes/Divorce-Your-Linux-Admin

A PDF version of the document may also be downloaded from:

http://www.tundraware.com/TechnicalNotes/Divorce-Your-Linux-Admin/Divorce-Your-Linux-Admin.pdf

This document was produced using `reStucturedText` and `TeXLive`.