Nix

This is a talk about the Nix package manager. Nix does a LOT of different stuff, so this covers only the basics. I am by no means an expert on Nix; I mainly signed up to do this talk so to provide myself with motivation to learn the tool. I make no guarantees about the accuracy of these notes. I wrote them so I would have reference material during a talk. They might be wrong.

System package managers

- `apt`
- `brew`
- `dnf`
- `zypper`
- `pacman`

Why do we use them?

- Convenience
  - It's cool to be able to run a single command to update all my software at once.
  - Anyone who's ever tried to do Linux From Scratch has stories of "dependency hell." Sometimes, things need to be built in just the right order and with just the right set of tools. It's nice that the package manager handles that for me.
- Security
  - Presumably, my distribution's packagers aren't putting malware in the repositories. (Unfortunately, this isn't always true.)
  - The package manager checks hashes and only retrieves software from trusted repositories, so it's harder to phish me.

- Cool
  - Some of them look pretty slick.
  - {Picture of pacman in ILoveCandy mode}

When don't we use them?

- When we want software that's weird.
- When we want software that's old.
  - Old software may expect old libraries. If the library has changed (looking at you, `glibc`) then your old programs may not build or run.
    * I just ran into this recently with the QEMU submodule in AFL. QEMU includes some headers from `glibc` that once contained no name conflicts, but do in more recent versions of `glibc`. Arch packages a recent `glibc`, so AFL's build system can't figure out how to get QEMU to compile. As a hack workaround,

    I just edited my system's `glibc` header files. QEMU built, and then I changed them back. No one should ever have to do that.

- When we want software that's proprietary.
  - Proprietary programs that use dynamic linking may require libraries that conflict with preexisting libraries on your system.
    - When this happens, you have no option but to patch the binary to use the API that your system supports.
- When we want software written in a programming language with a dependency culture.
  - For programs written in these languages, it's normal to pull in a million dependencies even for things that would be easy to implement.
    - Remember `left-pad`?
  - These languages provide their own package managers, which usually are a little sloppier than the one that comes

    with your system, and often pull packages from community repositories that are full of typo-squatting malware.
        * `pip` and `npm` are the big offenders here.
  - `venv` is a super clunky solution to this problem.
- When we need to use software in a way that works on someone else's computer.
  - When push comes to shove, people choose Docker.
    * Docker is often misused. While you *can* use lightweight containers with Alpine, many Docker containers just use Ubuntu, so you end up running 10 instances of Ubuntu simultaneously. This is slow on my little ThinkPad T420, and uses a lot of memory.
    * You need to configure device and network passthrough, and ensure that containers stay up to date on security patches.
    * Effectively, Docker adds another package manager to your system.

Nix

What is Nix?

- The Nix package manager **(The focus of today's talk)**
  - A package manager that solves some of the problems listed above.
- NixOS
  - A Linux distribution that uses the Nix package manager as its system package manager.
  - Allows the system to be configured reproducibly and declaratively.
    - i.e. you install GRUB with a few lines in a config file instead of having to run `grub-mkconfig` and `grub-install`.
  - You can also manage your system config with tools like home-manager.

- The Nix language
  - A functional programming language in which Nix packages are specified.

Using the Nix package manager

Setup

- You may be able to install Nix with your system's package manager. Otherwise, the Nix website recommends that you `curl` a script directly into your shell. This is usually kind of a bad idea for security, but it's your computer!
- Add yourself to the `nix-users` group.
- Enable the Nix daemon. This is available as a systemd service, but they have services available for other init systems as well.
- Add a Nix channel
  - This is like a version of the package repositories.
  - Because Nix supports many channels concurrently, you

can install older versions of software alongside newer versions.
    - You can think of this like being able to enable the Ubuntu 12.04 repositories on Ubuntu 22.04.
  - I recommend enabling the `nixpkgs-unstable` channel.
    - `nix-channel --add https://nixos.org/channels/nixpkgs && nix-channel --update`
- Log out and log back in.
  - This will add the appropriate things to your path.

Regular package management

- You can run `nix-env` to use Nix like a normal package manager.
  - To install a package, use `nix-env --install $SOME_PACKAGE`.
    - May also want to consider `nix-env --install -A`

    `nixpkgs.$SOME_PACKAGE`
  - To uninstall a package, use `nix-env --uninstall $SOME_PACKAGE`.
  - To list all your installed packages, use `nix-env --query`.
  - To search for a package, use `nix-env -qaP $SOME_PACKAGE`

Generations

- Now we actually get to something interesting.
- Every time you install or uninstall a Nix package, Nix creates a new generation.
- A generation encompasses the state of the packages on your machine.
- Use `nix-env --list-generations` to list all your Nix generations.
- If you install some software, and then things start to behave

strangely, you can roll back to your previous generation with `nix-env --rollback`

- If you need to go further back (or forward), you can switch generations directly with `nix-env --switch-generation $GENERATION_NUMBER`

Using multiple channels at once

- `nix-channel --add https://nixos.org/channels/nixos-15.09 nixpkgs1509` (for example)
- Now you can `nix-env -iA nixpkgs1509.wget` to get an old version of `wget`.

`nix-shell`

- Do the demo with python2
- Sometimes, you just want to use a package for a little while,

and then forget about.

- Enter `nix-shell`:
  - `nix-shell --packages bash` sticks you in a shell with the version of bash from your default nixpkgs repository.
  - You can also specify a particular version of nixpkgs with the -I flags.
  - This means I can send you a script that uses `nix-shell`, and be certain that when you run the script on your computer, it has exactly the same packages and configuration as when I run it on my computer.

How does all of this work?

- It's really complicated.
- The basics work like this:
  - When you install a package, Nix stores it in the Nix store, which is usually in `/nix/store`.

  - When your current generation would mean that you should have access to a particular package, Nix will symlink `.nix-profile/bin/wget` (for example) to the home of the appropriate version of `wget` in the Nix store.
    * On my system, that's `/nix/store/xw310xnhscrgk551d9kazr10`
  - The reason that the names in the Nix store are filled with junk is that every package's name in the Nix store is prefixed with the hash of that package's dependency graph.
    * This way, even if package maintainers have a bizarre versioning scheme that doesn't change when the dependencies change, the Nix package manager can still distinguish different versions of the same package effectively.
- Sandboxing
  - By default, packages built for Nix are built in a sandboxed environment such that as little as possible of the host

    system works its way into the package.
  - This also means that builds are reproducible.
  - Since the packages are specified in a purely functional language, the build scripts can't do weird things that cause the same build to make different results on different systems
    * This has the unfortunate side effect of making maintaining packages difficult
      · But this hasn't stopped people! There are over 80,000 Nix packages, and that number is rapidly increasing.

Solving dependency culture

- Nix also packages lots and lots of Python packages, so you don't have to use pip!

Garbage collection

- As you might expect, Nix sticks a lot of stuff in the Nix store, just like your system package manager keeps a lot of stuff in its package cache.
- If your Nix store gets too full, use `nix-collect-garbage` to clear it out.

No sudo needed?

- Nope!

Other frontends

GNU Guix

/usr/lib/xdg-desktop-portal –verbose -r /usr/lib/xdg-desktop-portal-wlr -l DEBUG