16 – Networking, OS, and Package Management

CS 2043: Unix Tools and Scripting, Spring 2019 [1]

Matthew Milano

March 1, 2019

Cornell University

Table of Contents

1. Firewalls
2. Operating systems, and what they do.
3. Containers, and how they work
4. Package Management
5. System Specific Package Managers
6. Other Managers

Firewalls

Firewalls

- In a perfect world, we wouldn't need a firewall.
- Lives in the network, or in the kernel
- inspects traffic *before* it reaches its destination
- Two primary uses: filter legitimate services, block unwanted ones

Firewalls: the good uses

- Legit: *Filters* certain ports to prevent regions of the internet from accessing them
    - Cornell firewall drops all traffic destined to on-campus servers originating from off-campus IPs
    - `wash` firewall does the same
    - mail relay firewall would only allow known senders to connect
- prevents server from being overloaded by random external griefers
- prevents aggressive server scans from the darkweb
    - which, by the way, exists. ask me later.

Firewalls: the lazy uses.

- Block insecure / old apps
- cover up for weird/bad OS/system design
    - Example: print server on a mac at port 631
    - Example: just a lot of windows
- Block **all** uninvited remote connections
    - if your laptop isn't a server, shouldn't have exposed ports
    - if it does have exposed ports, some application is doing a bad.
- Fundamentally lazy: right answer is to secure the applications, not hide them.
- lots of legacy apps (that we're stuck with) can't be fixed, so also fundamentally necessary

Operating systems, and what they do.

Processors

- The CPU; the chip at the center of your computer
- it actually runs your code
- wired via a *bus* to everything else in your computer
- Has multiple *cores* or *hyperthreads*
    - to allow code to execute simultaneously

Processors have protection modes

- Pieces of code get associated wth a protection mode
  - there's an instruction that literally says "when you run this code, drop these privileges"
- Protection modes let you drop lots of privileges
  - device access
  - physical memory access
  - ability to change protection modes
- Operating system always runs first and keeps all its privileges
- Operating system's job is to run processes for its users

What is a process, really?

- A sequence of processor instructions
- runs from start to finish
- only thing running on CPU core
- what can a process do?
    - access its own memory
    - run arbitrary computation CPU commands
    - fire *interrupts*

What is an interrupt?

- An “unexpected event”
- A request for something else to take over
- Like a signal (in C/unix), or Exception (in java/python/etc)
- Can register *interrupt handlers*, pieces of code that run interrupts
- The operating system registers itself as an *interrupt handler*
- A *syscall* is an interrupt handled by the OS
    - is how you read files, use network, etc.
    - OS registered the handler, so can have all privileges
    - most basic C functions / linux commands just fancy syscall wrappers!

A potential process flow

- start a process
  - drop privileges
  - jump to process code
- do some computation
- read a file
  - fire an interrupt
  - interrupt handler (in OS) gets file
  - file placed in process memory
  - jump back to process code
- use file contents
- do more computation
- exit with result
  - fire an interrupt
  - interrupt handler (in OS) gets result
  - OS clears process memory

Where VMs fit into this

- Using devices (from the OS) also interrupt-based!
- special instruction that sends message along system *bus*
- When host OS launches a VM
    - drops *some* privileges
    - registers *itself* (host OS) for device interrupts
    - launches guest OS
- when guest process wants to use a resource
    - interrupt back to guest OS
    - guest OS interrupts for device
        - Host OS gets interrupt
        - Host OS interrupts for device, or
        - Host OS takes over for a bit

Containers, and how they work

chrooting

change root directory

```
chroot <dir> <command>
```

- Must execute as root
- hides filesystem below `<dir>`
- `dir` looks like new `/`

- Why do this?
  - all **PATH**s relative to new root
  - system programs and libraries used from new root
  - can use programs that need incompatible libraries
  - can avoid upgrading system when using a program
- demo

chrooting

- What's still the same in the chroot?
  - kernel
  - process space
  - RAM
  - devices
- Halfway to a container; can have a `chroot` of debian on ChromeOS
- No isolation between `chroot`ed processes and "real" ones

containers

- Special OS feature called LXC containers
- hides processes from each other
- can limit device access within a single container
    - how? checks PID after interrupt, denies request from container process
- 90% of a docker container is chroot + LXC
- Other 10%? Secretly a VM.
    - but only when needed
    - this is why “fancy” Windows 10 is required
- Docker build scripts and bundles are also nice

Package Management

Package Management Overview

- If I had to give *only one reason* why Unix systems are superior to Windows: Package Management.
- Can install almost anything with ease of from your terminal.
- Update to the latest version with one command.
  - No more download the latest installer nonsense!
- Various tools can be installed by installing a *package*.
  - A package contains the files and other instructions to setup a piece of software.
  - Many packages depend on each other.
  - High-level package managers download packages, figure out the dependencies for you, and deal with groups of packages.
  - Low-level managers unpack individual packages, run scripts, and get the software installed correctly.
- In general, these are “pre-compiled binaries”: no compilation necessary. It’s already packaged nice and neat just for you!

Package Managers in the Wild

- GNU/Linux:
  - Low-level: two general families of *packages* exist: `deb`, and `rpm`.
  - High-level package managers you are likely to encounter:
    - Debian/Ubuntu: `apt-get`.
    - Some claim that `aptitude` is superior, but I will only cover `apt-get`. They are roughly interchangeable.
    - SUSE/OpenSUSE: `zypper`.
    - Fedora: `dnf` (Fedora 22+).
    - `zypper` and `dnf` use `SAT`-based dependency solvers, which many argue is fundamentally superior. The dependency resolution phase is usually not the slowest part though...installing the packages is. See [3] for more info.
    - RHEL/CentOS: `yum` (until they adopt `dnf`).
- Mac OSX:
  - Others exist, but the only one you should ever use is `brew`.
  - Don't user others (e.g. `port`), they are outdated / EOSL.

Using Package Managers

- Though the syntax for each package manager is different, the concepts are all the same.
  - This lecture will focus on `apt-get`, `dnf`, and `brew`.
  - The `dnf` commands are almost entirely interchangeable with `yum`, by design.
  - Note that `brew` is a “special snowflake”, more on this later.
- What does your package manager give you? The ability to
  - `install` new packages you do not have.
  - `remove` packages you have installed.
  - `update` installed packages.
  - update the lists to search for files / updates from.
  - view `dependencies` of a given package.
  - a whole lot more!!!

A Note on `update`

- The `update` command has importantly different meanings in different package managers.
- Some **do**, and some do **not** default to system (read linux kernel) updates.
  - Ubuntu: default is *no*.
  - Fedora: default is *yes*.
  - RHEL: default is *no*.
- It depends on your operating system, and package manager.
  - Know your operating system, and look up what the default behavior is.
- If your program needs a specific version of the linux kernel, you need to be very careful!

A Note on Names and their Meanings

- You may see packages of the form:
  - `<package>.i[3456]86` (e.g. `.i386` or `.i686`):
    - These are the `32-bit` packages.
  - `<package>.x86_64`: these are the `64-bit` packages.
  - `<package>.noarch`: these are independent of the architecture.
- Development tools can have as many as three packages:
  - The header files are usually called something like:
    - `deb`: usually `<package>-dev`
    - `rpm`: usually `<package>-devel`
  - The library you will need to link against:
    - If applicable, `lib<package>` or something similar.
  - The binaries (executables), often provided by just `<package>`.
  - Most relevant for `C` and `C++`, but also `Python` and others.
  - Use the `search` functionality of your package manager.

Example Development Tool Installation

- If I needed to compile and link against `Xrandr` (X.Org X11 libXrandr runtime library) pn Fedora, I would have to install
  - `libXrandr`: the library.
  - `libXrandr-devel`: the header files.
  - Not including `.x86_64` is OK / encouraged, your package manager knows which one to install.
  - Though in certain special cases you may need to get the `32-bit` library as well.
    - In this case, if I were compiling a program that links against `libXrandr`, but I want to release a pre-compiled *32bit* library, it must be installed in order for me to link against it.
- The `deb` versions should be similarly named, but just use the `search` functionality of find the right names.
- This concept has no meaning for `brew`, since it compiles everything.

System Specific Package Managers

Debian / Ubuntu Package Management (`apt-get`)

- Installing and uninstalling:
  - Install a package:
    `apt-get install <pkg1> <pkg2> ... <pkgN>`
  - Remove a package:
    `apt-get remove <pkg1> <pkg2> ... <pkgN>`
  - Only one `pkg` required, but can specify many.
  - "Group" packages are available, but still the same command.
- Updating components:
  - Update lists of packages available: `apt-get update`.
    - No arguments, it updates the whole list (even if you give args).
  - Updating currently installed packages: `apt-get upgrade`.
    - Specify a `package` name to only update / upgrade that package.
  - Update core (incl. kernel): `apt-get dist-upgrade`.
- Searching for packages:
  - Different command: `apt-cache search <pkg>`

RHEL / Fedora Package Managers (**yum** and **dnf**)

- Installing and uninstalling:
  - Install a package:
    `dnf install <pkg1> <pkg2> ... <pkgN>`
  - Remove a package:
    `dnf remove <pkg1> <pkg2> ... <pkgN>`
  - Only one **pkg** required, but can specify many.
  - "Group" packages are available, but different command:
    - `dnf groupinstall 'Package Group Name'`
- Updating components:
  - Update EVERYTHING: `dnf upgrade`.
  - **update** exists, but is essentially **upgrade**.
    - Specify a **package** name to only upgrade that package.
  - Updating repository lists: `dnf check-update`
- Searching for packages:
  - Same command: `dnf search <pkg>`
- **yum** and **dnf** (**Dandified Yum**) nearly interchangeable: [3].

`dnf`: Cautionary Tales

- **WARNING**: if you install package Y, which installs X as a dependency, and later `remove Y`
  - By default, X will be removed!
  - Refer to [2] for workarounds.
  - Generally, won't know you needed to `mark` until it is too late.
- Solution?
  - Basically, **pay attention to your package manager**.
  - It gets removed because nothing *explicitly* depends on it.
  - So one day you may realize "OH NO! I'm missing package X"...
  - ...so just `dnf install X`.
    - So while `mark` is available, personally I don't use it.
  - Sad face, I know. Just the way of the world.

OSX Package Management: Install `brew` on your own

- Sitting in class right now with a Mac?
- **DON'T DO THIS IN CLASS**. You will want to make sure you do not have to interrupt the process.
  - Make sure you have the "Command Line Tools" installed.
    - Instructions are on the First Things First Config Page
  - Visit `http://brew.sh/`
  - Copy-paste the given instructions in the terminal *as a regular user (not* `root`*:)*.
- **VERY IMPORTANT**: READ WHAT THE OUTPUT IS!!!! It will tell you to do things, and you *have* to do them. Specifically

```
You should run 'brew doctor' BEFORE you install anything.
```

OSX Package Management (`brew`)

- Installing and uninstalling:
  - Install a *formula*:
    `brew install <fmla1> <fmla2> ... <fmla2>`
  - Remove a formula:
    `brew uninstall <fmla1> <fmla2> ... <fmlaN>`
  - Only one `fmla` required, but can specify many.
  - "Group" packages have no meaning in `brew`.
- Updating components:
  - Update `brew`, all *taps*, and installed formulae listings. This does not update the actual software you have installed with `brew`, just the definitions: `brew update`.
  - Update just installed formulae: `brew upgrade`.
    - Specify a `formula` name to only upgrade that formula.
- Searching for packages:
  - Same command: `brew search <formula>`

OSX: One of These Kids is Not Like the Others (Part I)

- Safe: confines itself (by default) in `/usr/local/Cellar`:
  - No `sudo`, plays nicely with OSX (e.g. Applications, `python3`).
  - Non-linking by default. If a conflict is detected, it will tell you.
  - **Really important to read what `brew` tells you!!!**
- `brew` is modular. Additional repositories ("*taps*") available:
  - Essentially what a `.rpm` or `.deb` would give you in linux.
  - These are 3rd party repos, not officially sanctioned by `brew`.
- Common taps people use:
  - `brew tap homebrew/science`
    - Various "scientific computing" tools, e.g. `opencv`.
  - `brew tap caskroom/cask`
    - Install `.app` applications! Safe: installs in the "Cellar", symlinks to `~/Applications`, but *now these update with brew all on their own* when you `brew update`!
    - E.g. `brew cask install vlc`

OSX: One of These Kids is Not Like the Others (Part II)

- `brew` installs *formulas*.
  - A `ruby` script that provides rules for where to download something from / how to compile it.
- Sometimes the packager creates a "Bottle":
  - If a bottle for your version of OSX exists, you don't have to compile locally.
  - The bottle just gets *downloaded* and then "*poured*".
- Otherwise, `brew` downloads the source and compiles locally.
- Though more time consuming, can be quite convenient!
  - `brew options opencv`
  - `brew install --with-cuda --c++11 opencv`
  - It really really really is magical. No need to understand the `opencv` build flags, because the authors of the `brew` formula are kind and wonderful people.
  - `brew reinstall --with-missed-option formula`

OSX: One of These Kids is Not Like the Others (Part III)

- Reiteration: **pay attention to `brew` and what it says**. Seriously.
- Example: after installing **`opencv`**, it tells me:

```
==> Caveats
Python modules have been installed and Homebrews site-packages
is not in your Python sys.path, so you will not be able to
import the modules this formula installed. If you plan to
develop with these modules, please run:

mkdir -p /Users/sven/.local/lib/python2.7/site-packages
echo 'import site; site.addsitedir(                    \
    "/usr/local/lib/python2.7/site-packages")' >> \
    /Users/sven/.local/lib/python2.7/site-packages/homebrew.pth
```

- **`brew`** gives copy-paste format, above is just so you can read.
- I want to use **`opencv`** in **`Python`**, so I do what **`brew`** tells me.

Less Common Package Management Operations

- Sometimes when dependencies are installed behind the scenes, and you no longer need them, you will want to get rid of them.
  - `apt-get autoremove`
  - `dnf autoremove`
  - `brew doctor`
- View the list of repositories being checked:
  - `apt-cache policy` (well, sort of...`apt` doesn't have it)
  - `dnf repolist [enabled|disabled|all]`
    - Some repositories for `dnf` are *disabled* by default (with good reason). Usually you want to just `dnf --enablerepo=<name> install <thing>` e.g. if you have `rawhide` (development branch for fedora).
  - `brew tap`

Other Managers

Like What?

- There are so many package managers out there for different things, too many to list them all!
- Ruby: `gem`
- Anaconda Python: `conda`
- Python: `pip`
- Python: `easy_install` (but really, just use `pip`)
- Python3: `pip3`
- LaTeX: `tlmgr` (uses the CTAN database)
    - Must install TeX from source to get `tlmgr`
- Perl: `cpan`
- Sublime Text: `Package Control`
- Many many others...

Like How?

- Some notes and warnings about Python package management.
- Notes:
  - If you want `X` in Python 2 **and** 3:
    - `pip install X` *and* `pip3 install X`
  - OSX Specifically: advise only using `brew` or Anaconda Python. The system Python can get really damaged if you modify it, you are better off leaving it alone.
  - So even if you want to use `python2` on Mac, I strongly encourage you to install it with `brew`.
- Warnings:
  - Don't mix `easy_install` and `pip`. Choose one, stick with it.
    - But the internet told me if I want `pip` on Mac, I should `easy_install pip`
    - NO! Because this `pip` will modify your **system** python, **USE BREW**.
  - Don't mix `pip` with `conda`. If you have Anaconda python, just stick to using `conda`.

References

[1] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. "Previous Cornell CS 2043 Course Slides".

[2] Reddit.com. *DNF Remove Package, keep dependencies??* 2016. URL: https://www.reddit.com/r/Fedora/comments/3pqrv9/dnf_remove_package_keep_dependencies/.

[3] Jack Wallen. *What You Need to Know About Fedora's Switch From Yum to DNF*. 2015. URL: https://www.linux.com/learn/tutorials/838176-what-you-need-to-know-about-fedoras-switch-from-yum-to-dnf.