Before we start, make sure you can type the following characters on your keyboard

```
~ | & @ / "
```

When in doubt, use the American English keyboard

Introduction to linux

Just enough to dip your toes

Contents

Installation of Ubuntu server

Connecting to our lab environment

Please connect to the following wifi network:

- SSID: `INSERT_HERE`
- Password: `DROP TABLE USERS;`

Then, open: `https://192.168.30.10:8006` (ignore the security warning)

The logins are:

- User: `linuxlab`
- Password: `SuperSecurePassword123`

Creating your vm

We will now create the virtual machine for the lab

1. Press the Create VM button on the top right

1. General: Give it a cool name (Find solution for VM ID collisions)
2. OS: Ubuntu 24.04 live server ISO Image
3. Disks: 20gb disk with default settings
4. CPU: 1 socket 1 core
5. Memory: 1500 MiB
6. Network: Leave as is
7. Confirm: Start after created

Installation process

1. Accept english language and keyboard
2. Install the full Ubuntu Server
3. Leave network configuration as is, and skip proxy
4. Let the mirror pass checks and press done
5. Keep “Use entire disk” but remove “Set up this disk as an LVM group”
6. Fill in the computer and user information.
   (Make sure you remember the username and password)
7. Skip setting up, ubuntu pro
8. Skip installing other software

Short break

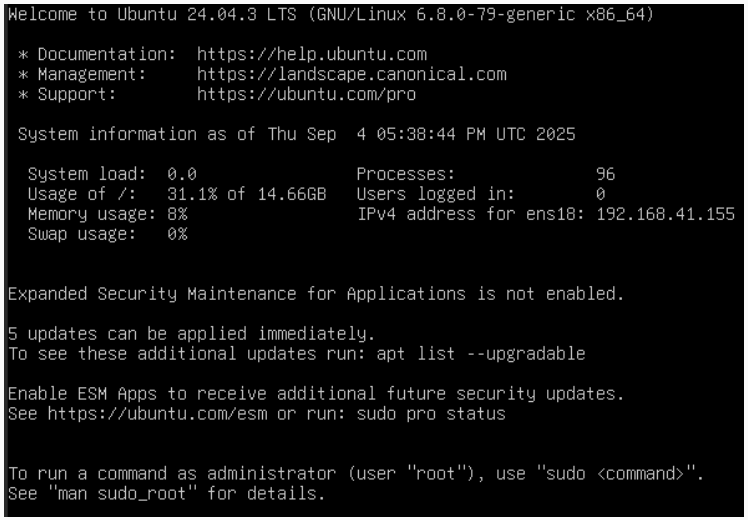
Grab some pizza and coffee while this completes

First steps

Starting to dip your toes

What is this place

When we log in, we see the MOTD (Message of the day)



This is something that ubuntu does for our “convenience”, but we can ignore it and clear screen with either `<Ctrl+l>` or `clear`

What is this place (continued)

What you're in is called a "shell".

In this particular case, the shell is BASH (**B**ourne **A**gain **SH**ell)

A "shell" is an environment that let's you interact with the computer through a command line.

You can run a command such as `ls`, which lists the content in a directory

But I don’t see anything ?

Ubuntu server *doesn’t* include the typical folders everyday operating systems include such as: `Desktop`, `Documents`, `Downloads`, etc.

There are however a few hidden files we can see if we run `ls -la`

What we added are flags, most commands will have options to tweak their behaviour, in this case we told it to present the output as a list, and to show all files, including hidden files

By default, all files starting with a dot are considered hidden files.

What's a .bashrc ?

You might recognize that our friend BASH, has a file here that shares the same name, and that's no coincidence.

The file is run whenever there is a new bash instance is started.

The `rc` suffix is taken from an old UNIX tool called `runcom`, the suffix can be found in a few places, notably the `initrc` project, or with configuration files like `vimrc`, `sshrc`, `bashrc`, `zshrc`

We're going to be editing this file, so let's create a backup copy of it so we can revert our modifications. `cp .bashrc .bashrc.old`

We can now open the file in my favourite command line text editor `nano .bashrc`

Going through the file

Most of what is set here are constants and configurations, so we will not touch them

If we head to the bottom of the file, we can add our own stuff at the end

We will be appending `echo "Welcome to my puter :)"` , which will simply print a string to the screen

We can save our modifications by hitting `<Ctrl-x>` followed by `y` to confirm, then pressing `enter` to tell it to overwrite the file

Let's start a new BASH session in our current session by simply typing `bash`, we will see that our changes worked

Let's exit our second BASH session by typing `exit` once.

Ok but where are we ?

We've seen how to look at what's aroud us, but seriously, where are we?

We can answer this question by typing the `pwd`, which will print the **P**ath to the **W**orking **D**irectory

We can see that we are in our home directory, which is where you store your personal files for your user.

Let's take a look at what's in the root of the filesystem by moving there `cd /` and listing the directories `ls -l`

We can see a there are a few directories, some are self explanatory like `/boot`, but let's dive into `/bin` by typing `cd /bin`

There are a lot of files in here, but if you have a keen eye, you might recognize some names such as `ls`.

This folder is one of the locations where your executables are stored. Instead of running the command, let's try running the file directly with `/bin/ls`.

You can see that you get the same output, which is pretty cool.

Where colours

You might've noticed that our output, unlike when running the command, has no colors. There is a cool reason for this

Let's look at what we're running when we type ls, by running `type ls`

What we see is that what we're running is actually an alias to ls, which adds some flags.

Let's take a look at the documentation for ls and see what it does `man ls`

You can search for color by typing `/color` and see that the flag is pretty self-explanatory

Let's exit the manual by pressing `q`, and adding this flag to our command `/bin/ls --color=auto`

We see that our outputs now match.

But how does bash find these files ?

`PATH` is an environment variable which is set in most operating systems (Linux, BSD, MacOS, Windows, etc.)

It contains a list of absolute paths where your shell will look for executables.

Let's take a look at our `PATH` value by running `echo $PATH`

This is a bit messy, let's pipe it into translate to replace all colons with line breaks `echo $PATH | tr ':' '
'`

We can see a nice list of all the paths loaded in `PATH`

Why are there so many paths ?

General guidelines for the different binary directories as indicated by this comment on UNIX StackExchange https://unix.stackexchange.com/a/6140

Path prefix	**/bin**	**/sbin**
/	Core binaries required for system functioning	Core binaries required for booting and low-level repair
/usr	Binaries used by logged in users	Binaries typically used for configuring binaries in /sbin
/usr/local	Same as /usr, but for user-compiled binaries not installed by your package manager	
/snap	Executables installed with Ubuntu's snap	

(These are just guidelines, placing executables in the wrong location likely won't break anything)

What other directories are there ?

Let's go back and take a look at the `/etc` directory by typing `cd ../etc`

If we take a look at the content by typing `ls`, we will see a lot of directories and plaintext files

This folder contains system configuration files. This is just like the windows registry, but actually useable

Configurations in `/etc` require higher privileges to edit, we can see that by looking at the info for the `cron` scheduler config with `ls -lah /etc/crontab`

File permissions

Here are the permissions we got for the configuration file

```
-rw-r--r-- 1 root root 1.2K Sep  7 18:37 /etc/crontab
```

The first dash indicates the type of file, so we can ignore it

Then we have three bunches of 3 characters indicating the permissions for User, Group, and Other

There are three permissions that can be set for each designation, **R**ead **W**rite and E**x**ecute

A file with max permisions would be `-rwxrwxrwx`

Then we have the filesize, the user and the group who own the file, which are both root

Root is the user with all permissions on linux. Similar to `NT AUTHORITY\SYSTEM` on Windows.

Cron

To be able to write our changes to the file, we need to elevate our privileges to root by prefixing our command with `sudo`

Let's add our own configuration to the `cron` scheduler with `sudo nano /etc/crontab`

Cron is a bit finicky in it's syntax, since this is the system crontab, let's write our entry like this

`11 * * * *   root    wall "I AM ANNOYING"`, which means on the 11th minute of every hour of every day of the month of every month regardless of the day of the week, run the command `wall "I AM ANNOYING"` as root

The `wall` command writes a message to all logged in users on the system

There is also a per-user crontab which you can access with `crontab -e`, which is doesn't require you to write the username and is likely the one you should be using

Devices

Let's go take a look at another important location, `cd /dev` and `ls`

In Linux, everything is a file, including your physical and virtual devices.

For example, we have `/dev/sda` which is your first drive and `/dev/sda1` which is your first partition on your first drive

Look at your current partitions with `lsblk`.

These locations are very useful when you're working with partitions with a tool like `fdisk`

These locations reference the underlying devices and not the data on them. These files are created by the kernel.

Virtual devices

There are a ton of virtual devices here too, such as `/dev/tty*`, which are used by your shell sessions to interact with the system

If you connect an Arduino or some other serial console, it'll likely appear as `/dev/ttyUSB0`

`/dev/loop*` are virtual devices for mapping files to block devices, you might use them when reading .ISO files

Utility devices

There are a few devices that are utility devices.

`/dev/null` is a tarpit where you can just pipe data into that you want to toss in your shell scripts.

`/dev/urandom` is an endless source of random data. Let's read the first 32 bytes from it with `dd bs=32 count=1 status=none if=/dev/urandom | base64`

`/dev/random` legacy worse version of `/dev/urandom` that is not guaranteed to be endless

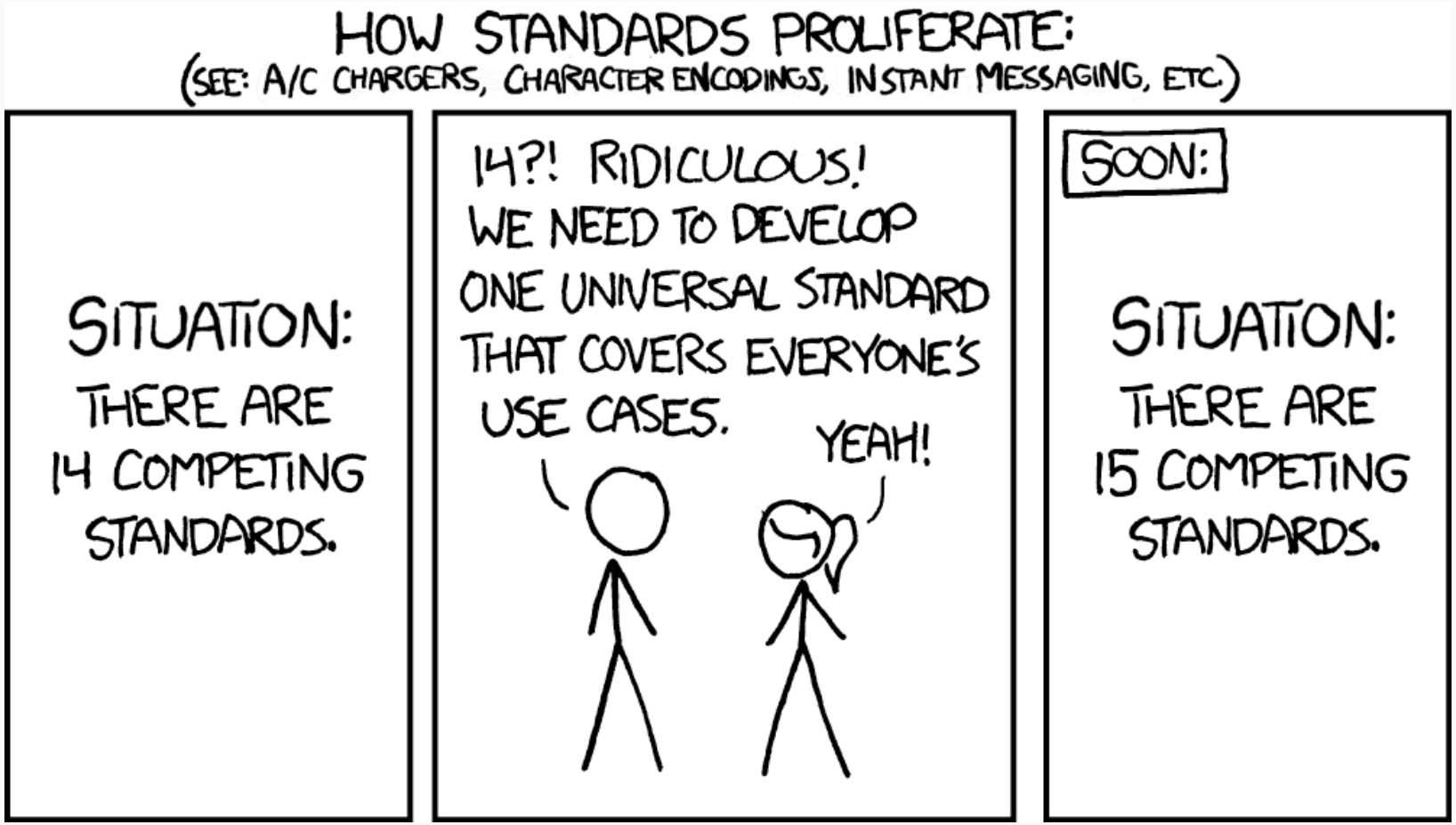
`/dev/zero` is an endless source of null characters

Some other relevant locations

Other relevant locations you must know about but we won't dive deep into

Location	Purpose
/mnt	Dsignated spot for system administrators to temporarily mount a device
/media	Contains mount points specifically for removable media (USB Sticks)
/opt	Used for installing programs not part of the OS that don't follow typical conventions
/proc	Contains files relating to processes, stdin, stdout, file descriptors, etc.
/var	Contains variable files, such as cache and your best friend: the logs
/tmp	Contains temporary files cleared on reboot. Also world writeable and excutable

Obligatory XKCD comic



Package management

Your typical distribution will come with one main package management system, and often you'll have a few other secondary ones.

In our case, since Ubuntu is Debian based, we will use `apt`

(If a tutorial makes you use "aptitude" or "apt-get", just use "apt", it's standard way to do things + it's shorter)

One other one you might consider is `flatpak` with it's own set of pros and cons

Ubuntu also includes `snap`, which the consensus on is that it's terrible.

There are also some language package managers which you might mess with (opam, mix, rebar3, composer cargo, npm, pip)

Using apt

The first step when using the `apt` package manager, is always to refresh the list of available packages with `sudo apt update`

This doesn't actually install anything new on your system, to actually install the new versions of your software, you need to do `sudo apt upgrade`. It will then provide you with a list of packages it'll install and total download and install sizes

Let's install the best package in the world, `ssh`, by typing `sudo apt install openssh-server`

Service management

SSH has two core components, the CLI tool which is included by default, and the server, which is what we just installed

We just installed the server, however we did not start it effectively does nothing. We can see it's status by running `systemctl status ssh`

Let's start by enabling the service, which means it'll startup with the service according to its configuration: `sudo systemctl enable ssh`

We can now start it so it actually runs with `sudo systemctl start ssh`

Let's check if it's actually running `systemctl status ssh`

Those familiar with `SSH` know that by default, it operates on TCP port 22, we can check if it opened that port with `sudo ss -lptn | grep 22`

We can look at the logs for the service with `journalctl -u ssh`

Connecting through ssh

`SSH` stands for **S**ecure **Sh**ell. It is a tool that allows you to connect open a shell on a remote device. It can do a ton of other cool and useful stuff with tunneling, but that's for another day

Right now we will use it for something simple: connecting to our vm from our local terminal, not having to deal with a browser tab

Let's start by getting the ip address of our VM by typing `ip a` and looking for the one that starts with `192.168`

Open your shell, on Windows it's `cmd`, on MacOS it's likely called `terminal`

Let's use the `ssh` command to connect to the vm, replacing the ip and the username appropriately `ssh user@192.168.xxx.xxx`

After typing the password, you should now have a remote session on the VM, open on your local computer.

Compiling our own software

Why would you do that

There are a few reasons why you might want/need to compile sofware:

- It is not included in your package manager
- You need to run a specific version that isn't available yet
- You need to patch it to fix a bug or add a feature
- You want to enable/disable a specific feature that the package manager version doesn't set
- You made the software

Whatever the reason, we are going to be compiling busybox

Compiling busybox

The first step is acquiring the sources. Often it'll be on Github, or SourceForge before that. Sometimes even it'll be on their own private git server.

Regardless, I have already acquired the sources for you so we won't have to deal with the slow download speeds

Let's make sure we're in our home directory with `cd ~` and download it from my local server with `wget ftp://192.168.30.200/busybox.tar.gz`

What is this file? A `tar.gz`? Let's take a look with `file busybox.tar.gz`.

It says it's a compressed file, and the `tar` extension means it's a **T**ape **Ar**chive

We can use the tar command to extract it with `tar -xzf busybox.tar.gz` (e**x**tract with g**z**ip the **f**ile)

Let's enter the newly created busybox folder with `cd busybox`

Some required rependencies

I have done some homework and have already found exactly what needs to be done to get this to compile

To start, we will need a few packages, which we will install with `sudo apt install build-essential libncurses-dev`

Many projects will include a configuration utility the menuconfig tool. In this project, we can access it with `make menuconfig`

This is a menu made using ncurses where you can toggle many features. There is one we will have to turn off so the project compiles

Let's go down to `Networking utilities` and then find `tc` and turn it off

We can then press right arrow, Exit, Exit again and save the configuration

We are now ready to compile the project with `make`

Another short break to grab some water

We're so back

Compiling software takes a few business days, depending on the language, project size, computer hardware, etc.

As it stands, we now have the code compiled, but just sitting everywhere with the source code, we need to install it somehow

By default, installations default to the root of our filesystem, however we want to contain it in a designated folder so it doesn't go everywhere

Let's install it in `/usr/local/busybox` with the following command `sudo make CONFIG_PREFIX=/usr/local/busybox install`

Let's go take a look at the files we just installed with `cd /usr/local/busybox` and `ls -lah`

Interesting, you might notice how some of these directory names seem familiar

Let's dig further and look at the binaries we generated `ls -lah bin`

Why are these all symlinks ??

It makes no sense at all, why are all the commands just shortcuts to the same file?

This is super interesting and one of the things that makes busybox so appealing for embedded systems.

It is a self-proclaimed “multi-call binary”, meaning that all the commands are in the same executable.

This means that they can make the executable really really small, which is great for when you have very little space

You can test it out by running one of these files such as `./bin/ls` and see that it behaves just as the normal command would

Let's set this as our default shell

Let's now change our shell to use busybox instead of BASH

The first step is to add our new shell to `/etc/shells` using our favourite editor `sudo ed /etc/shells`

`ed` is a fun text editor that was designed for when computers didn't have screens and operated by printing on paper with a teletype

Only type exactly what I say. Type in `a` then enter, this makes you enter append mode, adding to the current line which is the last line

Next we will type the path to our new shell executable `/usr/local/busybox/bin/sh` and press enter

We will exit insert mode by typing `.` then enter

We will now save and quit by typing `wq` then enter

Let’s set this as our default shell

We now need to set our users shell. We can see our current user info with `pinky -l username`

We see that the shell is `/bin/bash`, let’s change that by typing the `chsh` command

It will ask for our password, then we can type the path to the new shell, in this case `/usr/local/busybox/bin/sh`

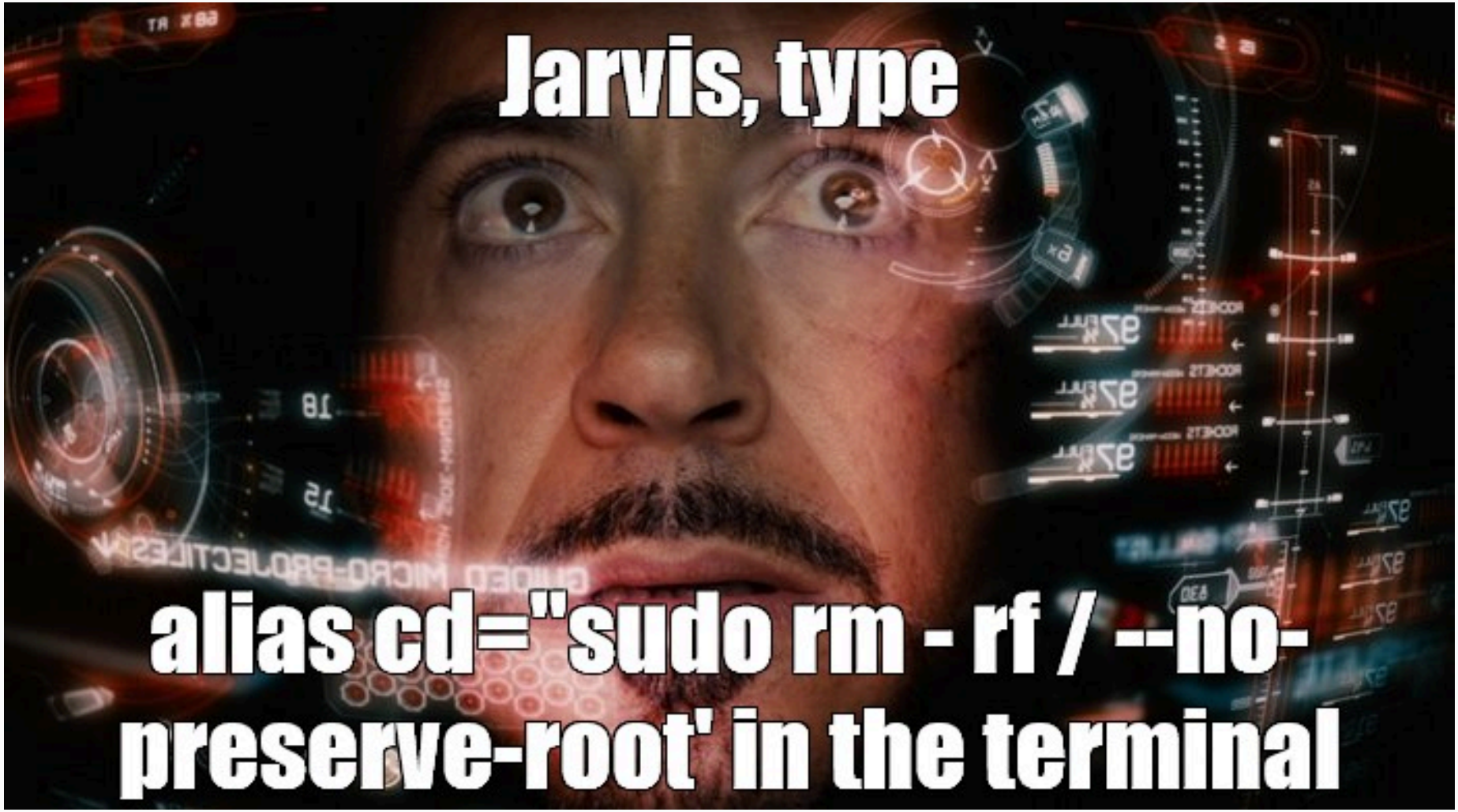
We can now exit our of our ssh connection with `exit`, and connect back to our VM

We might notice that it looks a little bit different, and if we `echo $SHELL`, we will see we’re using our busybox shell

We can also see the new shell on our user with `pinky -l username` again

Let’s break it all

Let’s wipe our install



Let’s wipe our install

Since the filesystem is structured as a tree, if we recursively delete from the root, we can just wipe all our data, including any mounted network drives, removable devices and drives.

Since this is an isolated VM with nothing connected, it is safe to run `sudo rm -rf / --no-preserve-root`

This will wipe your vm, if you want to keep messing around with it after the presentation, then feel free to ignore it, otherwise you can run it and destroy your installation for fun

Thank you for listening

Consider joining our club

Discord link

Official membership form