MSR Hackathon: Networking Lab

1 Introduction

In this lab, we will be doing some basic networking operations in Linux. Networking can be quite complex, and certainly deserves a course all of its own. In this lab I will try to walk you through several simple examples and tools that could be useful as you work through the rest of the year.

2 Checking for Required Packages

Before you begin you need to check that you have all required packages. Generally, your Linux distribution will already include the package that allows you to `ssh` into another machine, but many distributions do not include the package that allows someone else to `ssh` into your machine. To check, on a Debian-based machine like Ubuntu, use `dpkg -l | grep openssh`. This will print all packages that are installed or have been installed with "openssh" in their name. If you don't see `openssh-server` in the list, then you need to install that using `sudo apt-get install openssh-server`.

3 Establishing a Local Network

The first thing that we need to do is establish a network between you and your partner's computer. If you were already on the same local (via a wired or wireless router), you would be able to skip this step. If you were both on the internet, but not on the same local router, you'd still be able to complete the rest of this lab, but there may be many more steps required to ensure that your computers would be able to find each other. Complete the following steps:

1. **Disable your network manager** The `network-manager` daemon is Ubuntu's default tool for connecting to networks. When you click the little WiFi symbol in the upper right of the screen, you are using `network-manager`. We want to turn `network-manager` off to ensure that there are no conflicts between what it's trying to do with your hardware, and what we are trying to do. There is definitely a way to get this done without disabling `network-manager`, but it could be complicated. Disable network manager with

   ```
   sudo service network-manager stop
   ```

   and after this exercise, you can re-enable with

   ```
   sudo service network-manager start
   ```

2. **Plug an Ethernet cable into the port on each partner's computer** A few years back, there were special Ethernet cables, called *crossover cables*, for directly connecting two computers directly. These days, network adapters are capable of auto-detecting the need for a crossover communication channel, and automatically reconfiguring internally.

3. **Establishing a local network** We will use Avahi for assigning IP addresses to the computers on your local network and hostname resolution (multicast DNS). Avahi is a so-called *zeroconf* tool for automatically creating and managing local networks. This step could be done with old-school Linux tools like `ifup` and `ifdown`, but this way is much easier. First we need to get the name of your Ethernet *network interface controller* or *NIC*. This is the name that your computer uses to refer to the physical hardware that provides your Ethernet connection. Running the command `ifconfig -a` will report information on all network interfaces that are currently available, even if they aren't currently connected. On older systems the name of your Ethernet NIC is likely something like *eth0* or *eth1*.

On newer systems `systemd` has switched to using "predictable network interface device names" (read more here). In that case, the name you are looking for should begin with an `en` – it will likely be something like `enp3s0f1`. If you can't figure out the correct device using `ifconfig -a` you could try running `lshw -class network` (or even better, add `sudo` to the front of that command if you have permission). Then look for the `logical name` provided for each interface that has info printed out.

Once you have the names for each computer, open a terminal and type (where the last term should be the correct name)

```
sudo avahi-autoipd enp3s0f1
```

Keep this terminal running throughout the duration of the lab. Note that Avahi can easily be set to run automatically. When your command is finished running you should see a line that says *Successfully claimed IP address 169.254.X.XXX*.

4. **Examining your network** Both partners should be able to see their IP address now using `ifconfig` or `ip addr show`. Each partner should also be able to `ping` their partner's computer using their IP address. So if your partner's IP address was `169.254.44.204` you should be able to type type `ping 169.254.44.204` and see a response from their machine e.g.:

```
jarvis@vedauwoo:~/work/hackathon_2018 [2018*] ping 169.254.44.204
PING 169.254.44.204 (169.254.44.204) 56(84) bytes of data.
64 bytes from 169.254.44.204: icmp_seq=1 ttl=64 time=0.749 ms
64 bytes from 169.254.44.204: icmp_seq=2 ttl=64 time=0.525 ms
64 bytes from 169.254.44.204: icmp_seq=3 ttl=64 time=0.550 ms
^C
--- 169.254.44.204 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.525/0.608/0.749/0.100 ms
```

If name resolution is working properly you would also be able to ping your partner's computer using their hostname:

```
jarvis@vedauwoo:~/work/hackathon_2018 [2018*] ping test2018.local
PING test2018.local (169.254.44.204) 56(84) bytes of data.
64 bytes from test2018.local (169.254.44.204): icmp_seq=1 ttl=64 time=0.369 ms
64 bytes from test2018.local (169.254.44.204): icmp_seq=2 ttl=64 time=0.466 ms
^C
--- test2018.local ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.369/0.417/0.466/0.052 ms
```

You can figure out your own hostname using the `hostname` command. Note that we often need to add the `.local` to the end of the hostname when working with hostnames on a local network.

If pinging via hostname doesn't work, both partners can try restarting the `avahi` service using `sudo service avahi-daemon restart`. If hostname pinging still doesn't work, for now, just move on. One could edit the `/etc/hosts` file to provide a link between the partner's IP address and hostname, and then hostname pinging would almost certainly work. Quoting from the `hosts` manpage:

```
This manual page describes the format of the /etc/hosts file. This file is a
simple text file that associates IP addresses with hostnames, one line per IP
address.
```

4 Basic SSH usage

SSH stands for *Secure Shell*, and it is a common tool in Linux/Unix for providing secure, encrypted communication between two networked computers. It is important in both Git and ROS. SSH can be used for securely logging into remote systems, transferring files, running remote processes, tunneling, window forwarding, and more. It works well in both Linux and Mac. On Windows, it is trivial to SSH into a computer, but Windows does not support running an SSH client (to allow others to remotely log into the Windows computer). One of my most common SSH use cases is to log onto a remote server and run code that may run for a very long time. SSH combined with a terminal multiplexer (such as `screen` or `tmux`) allow me to log into the server and start some long-running process. I can then detach my multiplexer and log out while leaving the process running on the server.

In this section you and your partner will simply log into each other's machines.

1. **Note Details of Partner's Computer** You will need your partner's username, hostname, and IP address. Note that the default terminal prompt in Ubuntu is something like `username@hostname:pwd$`
2. **Using Partner A's Computer Log Into Partner B's Machine as Partner B via IP** Your command should be something like

   ```
   ssh partnerb@169.254.X.XXX
   ```

   You will then need Partner B's password as you are logging onto their machine as their user. If you had an account on their computer, you could instead SSH into their machine as yourself, and use your own password. **This is how remote logging in via SSH is actually done, you should never give someone your password!**
3. **Using Partner A's Computer Log Into Partner B's Machine as Partner B via Hostname** This command should be something like

   ```
   ssh partnerb@hostname.local
   ```

   the *.local* at the end of the command is not always needed. It generally depends on local router configurations; in our case with Avahi's default options, we need it.
4. Repeat the previous two steps but switch roles.

5 Copying Files Over a Network

There are two common tools for copying files over a network `scp` and `rsync`. SCP stands for secure copy, and it uses SSH for data transfer. It is very simple, and basically works exactly like the `cp` Linux command. Rsync is significantly more powerful at the cost of greater usage complexity. Rsync supports things like automatic-backups, network compression, intelligent ignoring of files, and special include/exclude tools. It is powerful enough that many experienced Linux users completely replace even the local `cp` command with Rsync. In this section we will create a dummy file, and copy it to your partner's machine using both programs.

1. **Create Dummy File** Run the command

   ```
   cd && echo "I just created this file!" > transfer_demo.txt
   ```

   to put a text file in your home directory.
2. **Partner A Copy File to Partner B with SCP** On partner A's machine, run the command (note you are connecting to partner B's machine as partner B)

   ```
   scp ~/transfer_demo.txt partnerb@hostname.local:~/scp_transfer_demo.txt
   ```

3. **Partner A Copy File to Partner B with Rsync** On partner A's machine, run the command (note you are connecting to partner B's machine as partner B)

```
rsync -avz ~/transfer_demo.txt partnerb@hostname.local:~/rsync_transfer_demo.txt
```

   The `-avz` means use archive mode, tell me what files you are transferring, and use compression when sending the data. Note that the biggest trick to using rsync is that whether or not there is a trailing slash in a directory actually changes rsync's behavior in a way that is not replicated by `cp` or `scp`.

4. Repeat the previous two steps but switch roles.

6 Advanced SSH Usage

6.1 Public/Private Key Authentication

SSH is equipped to handle several different standards for asymmetric encryption. Without getting into too much detail, asymmetric encryption is a general class of encryption techniques that require two different *keys*; one key, called the *public key*, is essentially just a huge number used for encrypting things. The other key, called the *private key*, is a different huge number used for decryption. The critical concept is that the private key *cannot*[1] be calculated from the public key. So Alice can create a public/private key-pair, then share the public key with Bob[2]. Bob can then encrypt a message for Alice using Alice's public key. Bob then transmits the encrypted message to Alice, and Alice can decrypt using her private key. As long as Alice never shares her private key, she can widely distribute her public key, and anyone with a copy will be able to transmit messages to Alice that only she can see.

There are many different asymmetric encryption algorithms (RSA being the most popular), and many different protocols/programs that implement/rely on these algorithms (GPG, SSL/TLS, SSH, Bitcoin). SSH can use several different algorithms for authentication. If Alice uses SSH to generate a public/private key-pair, and Bob adds Alice's public key to his list of "authorized keys"; then, Alice would be able to use SSH to Bob's computer without ever having to sign in or type a password. Her usage includes tools that use SSH as a transport layer (Rsync, SCP, Git, etc.). Note that it is trivial to register your public key with GitHub so that you can push to your repositories without using a password.

Let's use public/private keys over SSH:

1. **Both partners should check for a pre-existing public/private key-pair** Navigate to `~/.ssh/` and look for files called `id_rsa` and `id_rsa.pub`. If those files exist, then `id_rsa` contains your private key and `id_rsa.pub` contains your public key. These were generated by the `ssh-keygen` command line tool, and stored in SSH's default location. Note that you can have more than one key-pair, and they can be in other locations, but you then need to specifically tell SSH to use a key that is not in the default location. Also note that when you generate a key, `ssh-keygen` gives you the option to assign a password to the private key. If you enter one, you won't be able to authenticate using your private key unless you know that password. **So be sure to remember your password!** In Ubuntu, `gnome-keyring-daemon` works with `ssh-agent` to cache this password so you don't have to type it every time. You generally have to type the password once, and then `gnome-keyring-daemon` will remember the password until you logout or manually remove it (using `ssh-add -D`, for example).

2. **If you need a key-pair, generate one** Use the command (where you probably want to use a real email address)

```
ssh-keygen -t rsa -C "your_email@example.com"
```

   Then select the default location, and decide if you want your private key to have a password (I always use one). Read more here: https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/

[1] Technically, you could "brute-force" this calculation, but it would require an infeasible amount of computation

[2] For some reason cryptography people like the names Alice and Bob for their stand-ins.

3. **Both partners share their public keys** Each person should use SCP or Rsync to transmit a copy of their public key (~/.ssh/id_rsa.pub) to their partner. **Be sure not to accidentally overwrite either of the public keys!** In other words, don't copy from ~/.ssh/id_rsa.pub on your machine to ~/.ssh/id_rsa.pub on your partner's machine.

4. **Add your partner's public key to the list of authorized keys** The file ~/.ssh/authorized_keys contains the list of public keys that you are letting SSH into your machine via key-pair authentication. You can use a command like the following to add your partner's public key to the list

   ```
   cat <LOCATION_OF_PARTNERS_PUBLIC_KEY>/id_rsa.pub >> ~/.ssh/authorized_keys
   ```

   You could also do this by editing the authorized_keys file with a text editor and pasting your partner's public key onto its own line. Note that this step and the previous step can be done automatically using ssh-copy-id.

5. **SSH without a password** You should now be able to SSH into your partner's machine (as their user) without using your their password! Although, you may need your SSH key's password. Also note that even without logging in TAB autocomplete in Bash will now work over the network. For example, let's say you are trying to scp a file from your partner's computer to your computer. Then typing scp partnerb@partnerb.local:/home/ and then pressing tab several times should autocomplete the directories located in /home on partner B's computer without ever needing a password.

One final important point is that the files in ~/.ssh/ are per-user files. So if you were on a computer with multiple users each user gets to control their own authorized keys, and public/private keypairs. Also note that you've now given your partner access to login to your machine as your user without ever requiring a password (other than possibly the password to their private key). **This is obviously not very safe!** When you are done with this lab, be sure to remove your partner's key from your ~/.ssh/authorized_keys file (you could just delete the file).

6.2 Window Forwarding

SSH supports window forwarding with the -X option (capital letter). This allows you to open GUI programs on your partner's computer and have them show up on your computer. When you do this, you will typically want to enable some compression algorithms to reduce the amount of data that must be transmitted over the network.

1. **SSH with window forwarding and compression** Use the following command to SSH into your partner's machine with window forwarding and compression enabled:

   ```
   ssh -X partnerb@hostname.local
   ```

   The -X is all that's needed for window forwarding. Note that you can also control which *ciphers* SSH will use for compression when window forwarding. Although, when you start adding these options your SSH commands might be very long, e.g. ssh -c aes128-gcm@openssh.com -XC partnerb@hostname.local Many people use aliases or SSH config files to store these types of complicated commands so that they don't have to remember them.

2. **Open a graphical program** Just type gedit once logged in, and a gedit window should pop on your computer, but it is running on your partner's computer.

Note that there are many other window forwarding services that are more efficient that SSH. If you find yourself doing this regularly, it is definitely worth setting up something more efficient. In my opinion, the benefit of SSH window forwarding is that it is already on your system and requires basically no configuration. Common other options would be VNC, xpra, and nx.

6.3 SSH config files

SSH allows you to store settings for logging into various other computers in the file `~/.ssh/config`. Some great examples are presented here: http://nerderati.com/2011/03/17/simplify-your-life-with-an-ssh-config-file/ Try and write a config file for logging into your partner's computer.

6.4 Terminal Multiplexers

As mentioned earlier, the use of a terminal multiplexer with SSH can greatly increase your functionality. A multiplexer allows you to have multiple terminals open within a single SSH login – this allows you to only login over SSH once, and then simultaneously use multiple terminals. This is a very different functionality than the "tabs" provided by `gnome-terminal`. Additionally, multiplexers are persistent – they allow you to start processes, open multiple tabs, etc. and then detach from the multiplexer. Once detached, you are no longer using the multiplexer, but all of your processes and tabs still exist. You can re-attach, maybe even using a different computer, and your processes would still be there.

Likely, the most popular multiplexers are GNU Screen and tmux. `tmux` is generally considered to be more modern and easier to configure, but `screen` is great too. I personally use byobu basically all of the time, even when not using SSH. `byobu` is basically just a nice configuration for either `screen` or `tmux` – you can use `byobu` with either backend, and `byobu` will provide a consistent look and user interface. The keyboard shortcuts are easy to use, and `byobu` is easy to install on most Linux systems. You can see all of `byobu's` keybindings by looking at `/usr/share/doc/byobu/help.tmux.txt` or by visiting this post.

Try SSH'ing into your partner's machine, opening a `byobu` session, creating several tabs and starting several programs, and then detach from the session and logout. Log back in and re-attach to see all of your tabs and programs are still there.

6.5 GitHub SSH authentication

If you've used GitHub much with HTTP-protocol remotes (e.g. `https://github.com/username/project.git`) you may have noticed that you need to constantly type your username and password for GitHub when you are pushing to the remote. This is annoying! Nominally, you should be able to use the Git credential helper functionality to help you cache your GitHub passwords, but I've never had much luck with that. Instead, most of the time[3], I prefer to work with GitHub over SSH.

Since you already have an SSH key generated, you should be able to simply follow GitHub's documentation on Adding a New SSH Key to Your Account. Then you'd want to use the SSH URL for cloning, pushing, pulling, etc. The default, non-SSH URL would be something like `https://github.com/jarvisschultz/system_configurations.git` while the SSH-based URL would be `git@github.com:jarvisschultz/system_configurations.git`. Try setting up an SSH key with GitHub and then pushing to a remote over the SSH authentication protocol.

Here are a few helpful resources:

- GitHub SSH help topics: `https://help.github.com/articles/connecting-to-github-with-ssh/`
- Change a Git remote's URL: `https://help.github.com/articles/changing-a-remote-s-url/`
- Pro Git on *Working with Remotes*: `https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes`
- Pro Git on *The Protocols*: `https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols`

[3] One notable exception is when I've worked at places that had a secure internet connection that blocked SSH connections to the outside.