Using SSH and Raspberry Pi for Self-Hosted Backups

2019-04-22 - By Robert Elder

Introduction

The purpose of this guide is to show you the steps required to build your own automated backup solution that you can use for backing up source code or small files using git and SSH. This backup technique can be used to back up your data to another computer in your house or office, but you can also use it to back up to multiple locations over the internet securely.

This guide is targeted at individuals who plan to build this solution in an environment where the client and server will both use Linux. It may be possible to adapt the steps shown here to work on a Windows machine, but that won't be covered in this guide.

The following Linux commands will be used. Some of them will be explained below, but you may want to Google them if you've never seen them before:

- ip
- netstat
- sudo
- apt-get
- ssh
- vim
- nano
- git

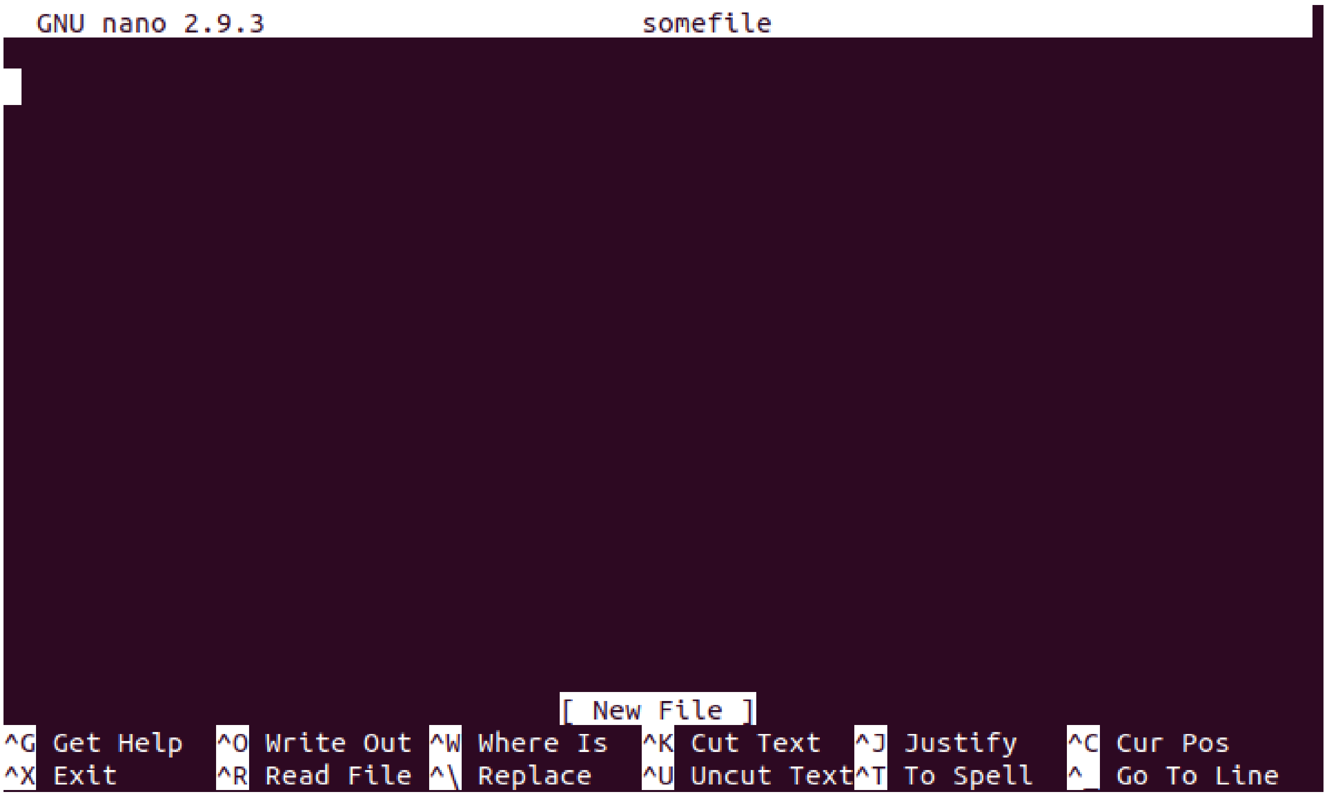
In the rest of this article I will present the final solution as a series of smaller bite sized 'goals' that build upon each other. It may not be obvious how each individual goal connects to the final outcome of backing up your files, but eventually this will all be tied together.

Goal 0: Editing Files and Installing Prerequisites

Later in this guide, you'll need to make a few edits to files on the command-line. If you're new to using the command-line, this might be difficult for you if you don't know what editor to use. Personally, I use an editor called 'vim', but if you've never heard of that before, I suggest using 'nano'. With nano, you can edit or create a file by running a command like this:

```
nano somefile.txt
```

Here's what nano looks like:



```
  GNU nano 2.9.3                    somefile

                                 [ New File ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

For the noobs out there, the instructions at the bottom of the screen that show the carrot symbol mean to press the control key and the letter key at the same time to perform the desired action. For example, to the '^O' means that you can press 'Ctrl + o' to 'write out' and save the file to disk. I won't say much else about nano since that's fairly off-topic and you can find guides online elsewhere.

Another thing to do before we get started is to install pre-requisites. I'll assume that you're using Ubuntu on your desktop/laptop. Here's the install command we need:

```
sudo apt-get install nmap git
```

You'll know that nmap is installed if you can run this command and get a version number back:

```
nmap -v
```

Once you've got nmap and git installed and you're confident that you can work with a command-line editor that can edit and create files, then you've completed goal 0!

Goal 1: Creating The Simplest git Remote

In this section, we'll make sure you have the confidence to set up your own git 'remote'. What is a 'remote' you ask? Well, it's the 'remote' place where your code and files end up when you do 'git push origin master' to push your code to GitHub (or BitBucket, or gitlab etc.). If you do a 'git clone ...', you are copying the files from the 'remote'. A 'remote' can be located on another computer, on GitHub, or even in another folder on the same computer. The simplest example of setting up a git 'remote' is actually just to create a directory on your computer and turn it into a 'remote'. First, let's make sure git is installed:

```
sudo apt-get update
sudo apt-get install git
```

Now, let's set up our git 'remote'. You can run these commands in any directory you like:

```
#  Set up and initialize a 'remote'
mkdir remote1
cd remote1
git init --bare
cd ..
#  Set up and initialize a local repo
mkdir my-repo
cd my-repo
git init
cd ..
```

The folder 'remote1' now contains a fully functional 'remote' that you can push code to, just like GitHub! The folder 'my-repo' contains an empty repo that you can start committing files to. Let's do that now:

```
cd my-repo
echo "This is my readme" > README.md
git add .
git commit -m "Create a readme file."
```

Now, what happens if you try to push to 'origin master'?

```
$ git push origin master
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

It didn't work because we didn't describe the relationship between our current repo (the my-repo folder) to the remote (the remote1 folder)! You can list all remotes by using the following command:

```
git remote --verbose
```

But we don't have any 'remotes' set up, so let's add one right now called 'origin':

```
git remote add origin ../remote1
```

Now, let's check to see what remotes there are:

```
$ git remote --verbose
origin  ../remote1 (fetch)
origin  ../remote1 (push)
```

Now let's try to push again:

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 235 bytes | 235.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ../remote1
 * [new branch]      master -> master
```

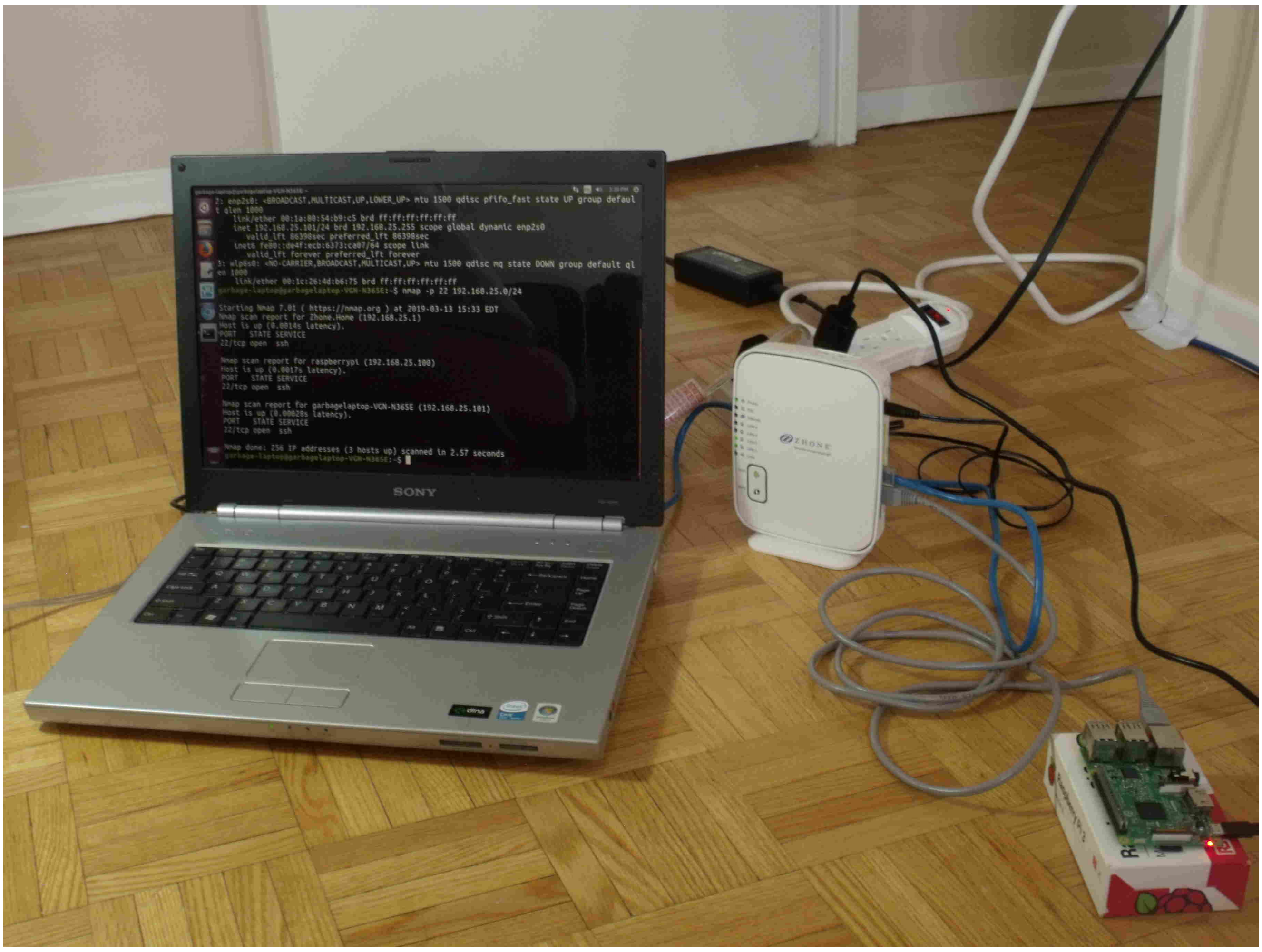
Awesome, it worked! We just created our own 'git remote' that we can push our repository to. This remote is still just on the same computer in another directory, but later we'll show how you can put it on another computer. You can even clone from the repo just like you would with any other git rep URL:

```
git clone /the/path/to/remote1/
```

Goal 2: Using SSH to Access Another Computer on the LAN

For this goal, we'll focus on making sure that you can use SSH to access the Raspberry Pi and run commands on it remotely. This goal doesn't have anything to do with git, but we'll use the two together in another goal. If you're not sure what 'SSH' is or what it does, you should do a quick skim of the article what is ssh before continuing with the goal.

For the following steps, I will assume that you're going to be working on a LAN setup that works something like this:



```
2: enp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group defaul
t qlen 1000
    link/ether 00:1a:80:54:b9:c5 brd ff:ff:ff:ff:ff:ff
    inet 192.168.25.101/24 brd 192.168.25.255 scope global dynamic enp2s0
       valid_lft 86398sec preferred_lft 86398sec
    inet6 fe80::de4f:ecb:6373:ca07/64 scope link
       valid_lft forever preferred_lft forever
3: wlp6s0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN group default ql
en 1000
    link/ether 00:1c:26:4d:b6:75 brd ff:ff:ff:ff:ff:ff
garbage-laptop@garbagelaptop-VGN-N365E:~$ nmap -p 22 192.168.25.0/24

Starting Nmap 7.01 ( https://nmap.org ) at 2019-03-13 15:33 EDT
Nmap scan report for Zhone.Home (192.168.25.1)
Host is up (0.0014s latency).
PORT   STATE SERVICE
22/tcp open  ssh

Nmap scan report for raspberrypi (192.168.25.100)
Host is up (0.0017s latency).
PORT   STATE SERVICE
22/tcp open  ssh

Nmap scan report for garbagelaptop-VGN-N365E (192.168.25.101)
Host is up (0.00028s latency).
PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 256 IP addresses (3 hosts up) scanned in 2.57 seconds
garbage-laptop@garbagelaptop-VGN-N365E:~$
```

To describe the setup above, this is one where you have your main laptop or desktop connected to the router (using either ethernet cable of WiFi), and your Raspberry Pi also connected to the same router using an ethernet cable.

With this setup, the first thing we should do is identify what local IP address the Raspberry Pi has on the LAN. In order to do that, you can run this command from the laptop to help us:

```
ip address
```

Here's the output that I get on my current laptop:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group
default qlen 1000
```

```
    link/ether d8:cb:8a:f3:2b:94 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.112/24 brd 192.168.0.255 scope global dynamic noprefixroute
enp3s0
       valid_lft 580673sec preferred_lft 580673sec
    inet6 2607:f2c0:e570:1c01:19ef:d1a8:ebea:214f/64 scope global temporary dynamic
       valid_lft 563416sec preferred_lft 61948sec
    inet6 2607:f2c0:e570:1c01:83eb:165e:445b:b377/64 scope global dynamic
mngtmpaddr noprefixroute
       valid_lft 563416sec preferred_lft 131416sec
    inet6 fe80::7f28:1a7d:9453:79b1/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: wlp2s0: <BROADCAST,MULTICAST> mtu 1500 qdisc mq state DOWN group default qlen
1000
    link/ether ac:2b:6e:14:e9:cf brd ff:ff:ff:ff:ff:ff
```

Note the highlighted part, '192.168.0.112/24' in this example, which indicates that my laptop has IPV4 address 192.168.0.112 on my LAN with the first 24 bits of that address being common to every computer on the network. Please be aware that the number 192.168.0.112/24 is just an example and your IP address will be different. Usually, your LAN ip address will start with something like '192.168.0.', but some routers also use addresses that start with '192.168.1.' or '192.168.25.' by default. In fact if you look at the picture above closely, you'll see that on the computer I used to pose for this photo, its IP address was actually 192.168.25.101/24. Often, you can also manually configure this LAN IP address prefix on the router settings.

Now, since the Raspberry Pi is also connected to the same network as the '192.168.0.112/24' address, we can run a port scan using this command:

```
nmap 192.168.0.112/24
```

to show all computers on this same network that answer back on common open ports. Since we're specifically interested in using SSH to access the Raspberry Pi, you can use this more specific command to only look for computers that answer back on port 22 (the default SSH port):

```
nmap -p 22 192.168.0.112/24
```

Note that sometimes, I've found that you will need to explicitly specify port 22 in order for the open port to actually be found. I've also experienced situations where I need to repeat the scan several times before it detects the open port. I assume that this is related to some kind of security filtering that certain routers do. Once you finish running nmap, you should see something like this:

```
Starting Nmap 7.60 ( https://nmap.org ) at 2019-00-00 23:19 EDT
Nmap scan report for router (192.168.0.1)
Host is up (0.00051s latency).

...
Nmap scan report for 192.168.0.177
Host is up (0.00054s latency).

PORT   STATE SERVICE
22/tcp open  ssh
...

Nmap done: 256 IP addresses (N hosts up) scanned in 2.55 seconds
```

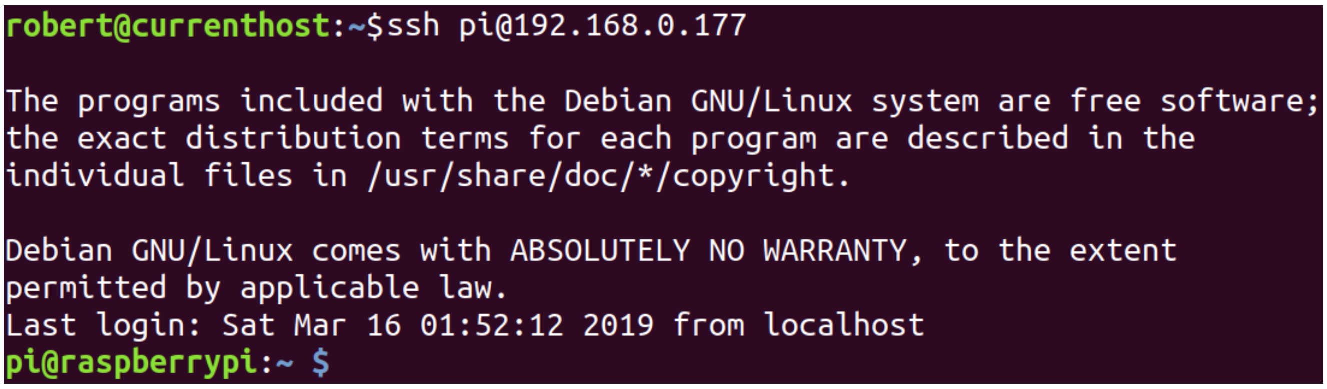
In this example, the IP address 192.168.0.177 is the address of the Raspberry Pi when it connects to my router, and doing a port scan with nmap was how we found it. When you run this command, you might get multiple results that have port 22 open, and if that's the case, it means you probably have multiple computers connected to the router that are running SSH.

 If you don't find any other computers that are running SSH, the Raspberry Pi might not have been set up to have its SSH server turned on yet! Newer versions of the Raspberry Pi operating systems usually have SSH disabled by default. Here is some Raspberry Pi documentation that describes how to enable SSH.

At this point, you should be able to run this command:

```
ssh pi@192.168.0.177
```

And, depending on how much you've set up your Raspberry Pi, it will likely ask you for a password. Do a Google search to find out the default password for your version of the Raspberry Pi OS. Once you successfully get access to the raspberry Pi them you've completed this goal. Here's a picture of what success looks like:



```
robert@currenthost:~$ssh pi@192.168.0.177

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Mar 16 01:52:12 2019 from localhost
pi@raspberrypi:~ $
```

You can use the 'exit' command to exit out of the Raspberry Pi SSH session:

```
exit
```

You have now successfully completed goal 2!

Goal 3: Using Public And Private Keys.

In this section, we will discuss the process of setting up SSH key pairs. This guide will focus on situations where you have physical access to both computers that you'll be using. The end result will allow you to use SSH to access one computer from the other without needing to type in a password. We will also assume that you already have the ability to access the machine you're granting access for through either password-based SSH authentication, or physical access which you can use to modify files directly.

For this discussion, we will assume that the computer you want to access is a Raspberry Pi (it could be any other Unix machine running an SSH server), and that its IP address on the LAN is '192.168.0.177'. We'll also assume that you want to log in under the user 'pi'. First, let's make sure that you have a '.ssh' directory on your laptop/desktop computer:

```
mkdir -p ~/.ssh
```

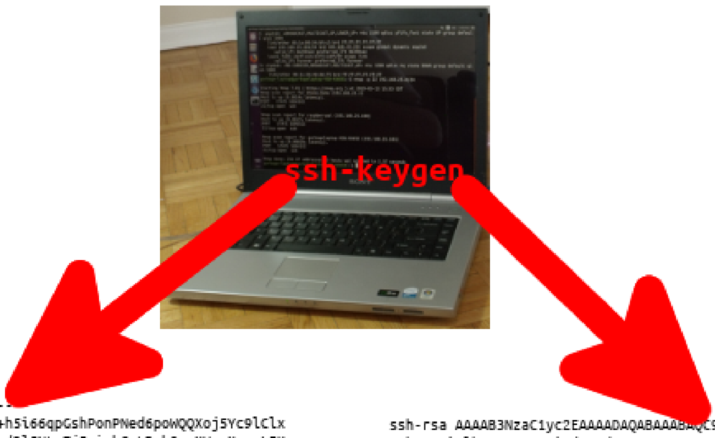
Now, in order to generate a public and private keypair (on your laptop/desktop), you can use these commands:

```
cd ~/.ssh
ssh-keygen
```

to create a keypair named 'my-first-keypair'. When you do this, adding a password is optional so we will leave it blank for this example. Here's what this looks like when I ran through the process:

```
robert@computer:~/.ssh$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/robert/.ssh/id_rsa): my-first-keypair
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in my-first-keypair.
Your public key has been saved in my-first-keypair.pub.
The key fingerprint is:
SHA256:2L1U7/JJ8Gak8RqUizZl+Y5b/gNPdybvMgtSBapp3GQ robert@computer
The key's randomart image is:
+---[RSA 2048]----+
|            .    |
|           . .   |
|        E +   .  |
|       B X *   . |
|  .     T @ +    |
|       . o %. o +|
|          * @+ =.|
|           # o= .|
|          +.=.o*.|
+----[SHA256]-----+
```

The result of running this command is to create two files: 'my-first-keypair' and 'my-first-keypair.pub'. The file 'my-first-keypair' contains the private key and the 'my-first-keypair.pub' contains the public key. Here's a visual illustration of what you just did:



```
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAvaQKJoVGU7huk+h5i66qpGshPonPNed6poWQQXoj5Yc9lClx
pmfYnv2iLXbkQRedsqthTu8YQJ438x/Rl5NLr7jIsinkCnLBwkQvaMHerMmmmLIK
GVlHIykiPKh5XyZmoqFYEpGGp3+ProOxNeEe6KBhja4MmiGAPtGLJ3QDVH90mVod
yzM4tJm+Bh/20cGZ7I8LUjgygTH8/dG+iB2oIukym5V3MDvM7B7uyUy9KM76BkWM
jm2kVfAQ8MjzAbQ1Ny6YqU9eaofXQrYuB7JQr72S7XvF4e8hXl39qGcFoCCuMfev
AaDoOGcq/kBcbbzhaLwo1gQ+oQ59RYA8fzoTGQIDAQABAoIBABUv8JphJXxRDLml
rX63owT/mxU7nFTvabqVDYrjNIN+WpjlEHeA/qi0uZvOvq2vD8VYdGS3PcRqaOHW
2jnPEROu121gVMP+Eju8FnfjwRcY6G4C5jHy0YnSHKseP4Er7yeYUxiv/ZVVIR5k
RDY60pM25+9GVM4blBJwotVjlsOn0Gyf6AjVOptVE2TbsSRHHxq2qVD9Ihye4Eg6
xWeG1d5SDPXVWzU4B9c9iBsB0bf6QCE9qjKgcMUErORt6pOswY4Hj+3idG5wsGQ3
Dxheok1XWP/IhLBoQqRB6g1f3V3dkOvM/Havt2ew0BoAYGuG6usGea5wZMvs+eDo
Y6EjWaECgYEA660G0futK7xeA0EZ7AF03r4VFgaOHx0OfdPfuA4RAwPHPh0K0k7Z
tzAKatm2BOxiGjf75OiMzSePMpnuWR1t72UVagjYdzamBKWR2gqrGzPOW/Hakdnu
U20jkmjdW9xwY1qMg2far3qF8yQfyMZ84DOn4zkkDru7ttJcaZ2yD6UCgYEAzf60
VKpZP13qzKxNOieg38I9IO3Dj3LtsR9Zh2kMhruzSXVwqj6fF+2KOWAsMlWZjakG
Mhd8cpOC2xWDt7HPooPzohgqLtiE7nuYtaAbC2OuqTPv7Hbfe575ufCp8XLAWQ8D
ZLkB73rQF+/tcil50XMf/qz0xaqpxTH1lW6tm2UCgYEAtz3VK7dPsc3IGyenDAUl
5XpN0VkY78Ab4GG4dNcbCwkJBZZ3L4X3aaOEtgeIqtNQg6o7xIO71leH04Qz+j4T
mCs0NzmhdS+cbch+WtYorAHf1UV6+CRGVb9qBRBb7Gy9RcLFtWDle8L8G0xfGjCa
L4h6ZQWsr3fwP/nXuJbQN7ECgYEAjjXU0M55pqFzw7D/77IEAfXatazvYUz6g1kd
ObpwB2P1tjT5fs6UjrR5xKF8YZNu2rlhqZtFMuRbZlxH/r0mlw95VuJP3N9lhi6V
km3hvc8neXB9pkW6q2FpDdDzwdhyqgkPUq3WhDnJB31/81xm2Q4dfHhy0zOq2JRQ
8sTh3XECgYEAyv/TGajUY1xafQnXlEZ1ZckzqRavGmr3ap/awipDCES0n6XM/83S
3uG4dEu181WAzw5jMvg6x0jGKccnzVAhhqWov6Ph5eENDMRiqzX0fyv5t7sAc53E
7rPkw6VFrHKWbZ8EldEMEh1gXK6Ts9B/AtykDf2UUHBSCJOj+UevJok=
-----END RSA PRIVATE KEY-----
```

'The private key'

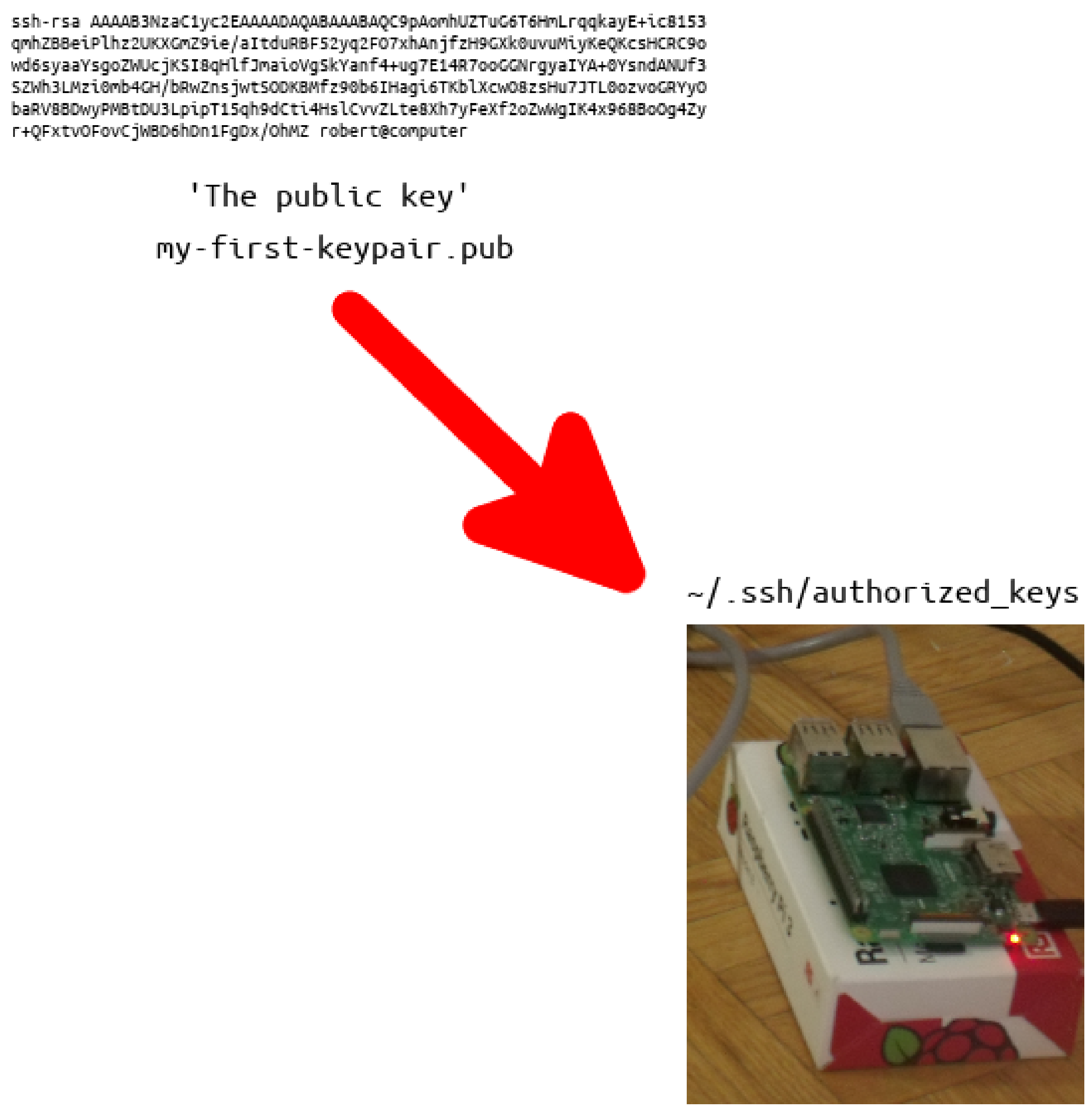
my-first-keypair

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9pAomhUZTuG6T6HmLrqqkayE+ic8153
qmhZBBeiPlhz2UKXGmZ9ie/aItduRBF52yq2FO7xhAnjfzH9GXk0uvuMiyKeQKcsHCRC9o
wd6syaaYsgoZWUcjKSI8qHlfJmaioVgSkYanf4+ug7E14R7ooGGNrgyaIYA+0YsndANUf3
SZWh3LMzi0mb4GH/bRwZnsjwtSODKBMfz90b6IHagi6TKblXcwO8zsHu7JTL0ozvoGRYyO
baRV8BDwyPMBtDU3LpipT15qh9dCti4HslCvvZLte8Xh7yFeXf2oZwWgIK4x968BoOg4Zy
r+QFxtvOFovCjWBD6hDn1FgDx/OhMZ robert@computer
```

'The public key'

my-first-keypair.pub

The 'private' key is called private key because you're supposed to keep it secret and it never leaves the computer where it was created. The 'public' key is called public because you can freely share it with others publicly. The private key is something we will use to log into the Raspberry Pi using SSH after we add our public key onto the Raspberry Pi. In order to 'distribute' the public key, you must add it inside a file located at '~/.ssh/authorized_keys' on the machine you need to log into:



```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC9pAomhUZTuG6T6HmLrqqkayE+ic8153
qmhZB8eiPlhz2UKXGmZ9ie/aItduRBF52yq2FO7xhAnjfzH9GXk0uvuMiyKeQKcsHCRC9o
wd6syaaYsgoZWUcjKSI8qHlfJmaioVg5kYanf4+ug7E14R7ooGGNrgyaIYA+0YsndANUf3
SZWh3LMzi0mb4GH/bRwZnsjwtSODKBMfz90b6IHagi6TKblXcwO8zsHu7JTL0ozvoGRYyO
baRV8BDwyPMBtDU3LpipT15qh9dCti4HslCvvZLte8Xh7yFeXf2oZwWgIK4x968BoOg4Zy
r+QFxtvOFovCjWBD6hDn1FgDx/OhMZ robert@computer
```

In order to use SSH keys to log into your Raspberry Pi, you must first copy your public key to the Raspberry Pi inside a file located at '~/.ssh/authorized_keys'. The 'authorized_keys' file can store multiple keys (one on each line) if you have multiple people or keypairs that are allowed to log in. If you're able to use SSH with a password to access the Pi, you should first make sure that the '~/.ssh' directory exists on the Raspberry Pi:

```
ssh pi@192.168.0.177 "mkdir -p ~/.ssh"
```

Now copy the public key over and add it to 'authorized_keys' by running this this command from the laptop:

```
cat ~/.ssh/my-first-keypair.pub | ssh pi@192.168.0.177 "cat - >>
~/.ssh/authorized_keys"
```

This command looks complicated, but it just reads a copy of the public key on your current computer, then sends it through the SSH connection and adds it to the end of the 'authorized_keys' file. If you want to check to make sure the public key got copied over, run this command from the laptop:

```
ssh pi@192.168.0.177 "cat ~/.ssh/authorized_keys"
```

If you see something that looks like your public key, then it has been copied successfully. Now, try using your private key to see if you can use ssh to log into the Raspberry Pi without a password:

```
ssh -i ~/.ssh/my-first-keypair pi@192.168.0.177
```

If you get a command prompt on the Raspberry Pi, then you're in! Congratulations! You have now completed the process of distributing your SSH key!

Goal 4: Setting Up An SSH config File

For this goal, our objective is to make it easier to use SSH to access your Raspberry Pi. For example, we'll make it so that instead of typing this:

```
ssh -i ~/.ssh/my-first-keypair pi@192.168.0.177
```

you can type this instead:

```
ssh pi-backup
```

which is much shorter and easier to remember!

The way to accomplish this is by editing a file located at '~/.ssh/config'. It's likely that this file won't already exist, and you'll have to create it. Run this command to edit the ssh config file:

```
nano ~/.ssh/config
```

And to set up the alias for 'pi-backup', add this to the file:

```
Host pi-backup
    HostName 192.168.0.177
    Port 22
    User pi
    IdentityFile ~/.ssh/my-first-keypair
```

Then save and exit. Also, keep in mind that the IP address '192.168.0.177' written above is just a specific example. You'll need to replace it with the IP address of your Raspberry Pi. Once you do, you should now be able to run this command:

```
ssh pi-backup
```

Once you're able to SSH into the Pi using this easier method, you've successfully completed goal 4!

Goal 5: Pushing to a repository on the Raspberry Pi Through SSH

Now we're ready to do something that looks a bit closer to actually backing up files onto your Raspberry Pi! Remember all the steps you did for goal 1? You're going to repeat them, but with a couple differences. First, let's use SSH to log into the Pi:

```
ssh pi-backup
```

Now, set up a git 'remote' in your home directory on the Raspberry Pi:

```
#  Set up and initialize a 'remote'
mkdir my-first-backup.git
cd my-first-backup.git
git init --bare
```

Then exit back to your laptop/desktop computer:

```
exit
```

Now create a local git repo on your laptop/desktop and add some data into it:

```
#  Set up and initialize a local repo
mkdir my-repo
cd my-repo
git init
echo "Hello World" > README
git add .
git commit -m "Create a readme file."
```

The last step is to tell the local git repo that the 'remote' we want to use can be accessed through an SSH tunnel using git. For this we add a remote that uses the ssh config file alias as the prefix followed by the directory on the Raspberry Pi where we set initialized the remote. We'll also run a one-time-step the set 'master' as the default branch on the remote.

```
git remote add pi-backup pi-backup:~/my-first-backup.git
git push --set-upstream pi-backup master
```

You should now be fully set up to push and pull to the repo located on your Raspberry Pi! Let's do another test just to make sure everything is working:

```
echo "Awesome" >> README
git add .
git commit -m "Edited readme file."
```

And now when you run this command:

```
git push pi-backup
```

And you should see something like this:

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 226 bytes | 226.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To pi-backup:~/my-first-backup.git
 * [new branch]      master -> master
```

If you do, you've completed goal 5!

Goal 6: Securing the Raspberry Pi

Securing your Raspberry Pi is a very large topic can cover many aspects of computer security, and as a great starting point I would recommend that you first read the Raspberry Pi foundation's own guide on Securing your Raspberry Pi. In addition, I will provide some commentary on this guide in order to emphasize what I think are the most important points, but also to add a few details that aren't reflected in the guide. I will make reference to the Raspberry Pi foundation's security guide a number of times in the paragraphs below.

The Most Important Thing

The absolute most import part of securing your Raspberry Pi is to make sure that you haven't left on password based SSH access with the default password. There are many bots scanning the internet constantly trying to log into things with a username of 'pi' and password 'raspberry' since this is a common default login for SSH on Raspberry Pi. If you fail to disable password-based SSH authentication and leave the password as default, you are effectively leaving your Raspberry Pi openly accessible to anyone on the local area network it is connected to. If your Raspberry Pi is behind a router on a home-based network, this will make it harder for hackers to attack you, but not impossible. If you want to be super safe, you should try to secure your Raspberry Pi as if it were directly connected to the internet, and it that case, you DON'T want to have password-based SSH connections enabled with the default password!

To fix this potential huge security problem you need to either disable password based SSH authentication, or change your password to be a strong password. I recommend doing both. To change the password to a more secure one, use this command:

```
passwd
```

you'll be prompted to update the password for whoever the current user is.

To disable password based SSH authentication, you need to edit the file located at '/etc/ssh/sshd_config' and make sure it contains the following line somewhere:

```
PasswordAuthentication no
```

If instead you find the line 'PasswordAuthentication no' OR if you find a line that starts with a '#' character ('#' means commented out), then you'll need to update or add this line as written above. Also, before you disable password based authentication, make sure that you first set up SSH key based authentication with public and private keys so that you can still access the Pi remotely. Otherwise, you won't have *any* way to log into the Pi using SSH. Once you add this new configuration, you need to restart the SSH server for it to take effect:

```
sudo service ssh restart
```

Another big win you can do to improve security is to require require a password when running the 'sudo' command. This is explained in the guide linked above, and the solution involves editing the sudoers file. Editing the sudoers file (or any file it includes) can be a potentially dangerous operation: If you manage to put a syntax error in the sudoers file and save it, then you won't be able to use sudo to become root and edit the file again! If you get into this situation, there may or may not be certain workarounds fix the problem, so take care to avoid it. There is a command called 'visudo' that can be used to edit sudoers file safely:

```
sudo visudo -f /etc/sudoers
```

The '-f' option of visudo is used to edit other files than the default one at '/etc/sudoers '. You'll need to search for a line that looks like this:

```
pi ALL=(ALL) NOPASSWD: ALL
```

and replace it with one that looks like this:

```
pi ALL=(ALL) PASSWD: ALL
```

You may need to snoop around in some of the included files. In my case, I found it in '/etc/sudoers.d/010_pi-nopasswd'.

Another important part of staying secure is making sure you have the latest security fixes. You can get yourself up to date by running the following two commands:

```
sudo apt-get update
sudo apt-get upgrade
```

The Raspberry Pi security guide suggests installing a cron job to keep SSH up to date, but you can also look into installing Unattended Upgrades to do this without a cron job. You can also get more flexibility in how and when you upgrade by reading more on the unattended upgrades documentation.

Another common security-minded practice is to change your SSH server configuration so that your SSH client runs on a non-standard port other than the default of 22. This doesn't prevent anyone from doing anything that they couldn't do if you used port 22, but it does make your SSH server less likely to be detected by dumb scans of every host on the internet for port 22. However, some would argue that if you change your SSH port to a port over 1024 that this could actually be a security risk since non-privileged user processes are allowed to bind to ports over 1024, but you must be root in order to bind to ports less than 1024.

Yet another thing you can consider is removing the default Raspberry Pi user of 'pi' and replacing that with another hard to guess username. This makes it even harder for attackers to guess what login information they would need to use to gain access to your system. It would also make various other kinds of blind attacks more difficult. It should be noted though, that some versions of the Raspberry Pi *require* the 'pi' user to exist in order to function properly, so be sure to research this before deleting the 'pi' user.

One final thing you can do to make your setup more secure is password-protect your SSH private keys. When you use 'ssh-keygen' to generate your keys the private key is itself as good as password. Therefore, anyone who gains access to your computer with the private keys, even for a short time, can steal the private key and user it over and over like a password. If you password protect the private key, you'll have to type a password every time you log into something and use the private key since it will be encrypted.

Setup Notes For Old Laptop

There isn't a lot to say about this topic, but is worth mentioning that you could also make this backup solution work on an old spare laptop. I would suggest installing Ubuntu on the old laptop since most of the install instructions listed here that work for the Pi will also work on Ubuntu. You can download a copy of Ubuntu here.

Setup Notes For Raspberry Pi

If you bought a brand new Raspberry Pi, you may or may not need to flash the SD card with a new OS image. Some Raspberry Pi kits come with an OS like Raspbian pre-installed. If it is pre-installed, you should take note of what version of the OS is pre-installed.

If an OS other than Raspbian is installed, you will need to consult the documentation for that OS to learn how to make sure SSH will be set up and ready to use.

If you decide to install a custom version of Raspbian OS yourself or if your SD card came blank, consult the guide at Raspbian install instructions which provides and overview of the process for Windows, Linux and Mac.

Before you install Raspbian OS, you will notice that there are at least two different versions of the OS image. One is called the 'desktop' version, and one is called the 'lite' version. The 'desktop' version includes all the software needed to present a nice user interface that reminds you of Windows with lots of clickable buttons and icons etc. The 'lite' version doesn't have any of this and expects you to know Linux commands because it only presents you with a terminal where you can type in commands. If you're a n00b, you should probably go with the 'desktop' version. If you're more experienced with Linux commands, you may prefer the 'lite' version because it will run faster, use less RAM and it won't require a larger size SD card.

Once you've got your Raspbian OS installed, the next thing to do is make sure that you have an SSH server running on it so you can access it through the command-line. Consult this article on setting up SSH on Raspbian OS for details. If you're using the UI, there is an easy UI feature to enable SSH. &nbps;If you're on the command-line, you can do:

```
sudo touch /boot/ssh
```

Then, reboot the Raspberry Pi, and run the following command to make sure that the SSH server is running:

```
ps -ef | grep sshd
```

You should see at least one entry that contains a reference to the sshd executable '/usr/bin/sshd' like this:

```
root      1234     1  0 07:46 ?        00:00:00 /usr/sbin/sshd -D
```

If you don't, then the SSH server is probably not running and you'll have to debug why.

Static Versus Dynamic IPs

The backup solution described by this article has assumed that you have your Raspberry Pi hosted on the LAN on a given local IP address (192.168.0.177 in our example). However, we haven't considered the fact that next time you reset all your devices, this IP address is not guaranteed to be the same. This will mean that any SSH rules you've set up won't work anymore. But how do we solve this problem to make sure our backup solution is truly always 'automatic'? The answer to this question involves the DHCP protocol, which you may want to read up on.

There is more than one way to guarantee that our Raspberry Pi always has the same IP address on our local network, and two different general approaches are:

- **1) Change the settings in your router to always assign the Raspberry Pi the same IP address.** - Using this solution you will change some setting on your router only and leave all of the settings alone on your Raspberry Pi. Your Raspberry Pi will continue to use the DHCP protocol to obtain a 'dynamic' IP address, but the router will remember that your Raspberry Pi (specifically its MAC address) should always be assigned the same address. If you don't know how to log into your router's admin page, check the back of the router as it will usually have some default username/password and IP address printed on it. You can log into many common household routers by using a web browser to access '192.168.0.1'. You can also use the nmap command discussed elsewhere in this guide to scan for anything on the LAN that talks on port 80.
- **2) Change the Raspberry Pi's configuration to give it a static IP address.** - Using this method, you change some of the network setting on your Raspberry Pi so that it always uses the same IP address every time it boots up. In this case, it does not rely on the DHCP server hosted on your router to decide what IP address it has. It simply chooses to use an IP like '192.168.0.177' regardless of what everything else on the network is doing. In this situation, you need to be careful to make sure nothing else gets assigned the same IP address on the network, otherwise both computers would experience problems.

Option 1 is probably the easiest, although it assumes that your router includes such a configuration feature. Usually, what you can do is connect the Raspberry Pi to the Router, and then log into the router admin panel where it will show you what devices are connected, and then present you with the option to pin an IP address somewhere.

If you decide to use a static IP address for the Raspberry Pi, you should be careful not to use a static IP address that is not within the DHCP lease range that the router can assign. Otherwise, there could be a case where the router accidentally assigns the same IP address that your router is using to another computer on the network To determine the DHCP lease range, you can likely find it somewhere inside the router's admin panel. Also, make sure that the static IP that you use has the same subnet mask.

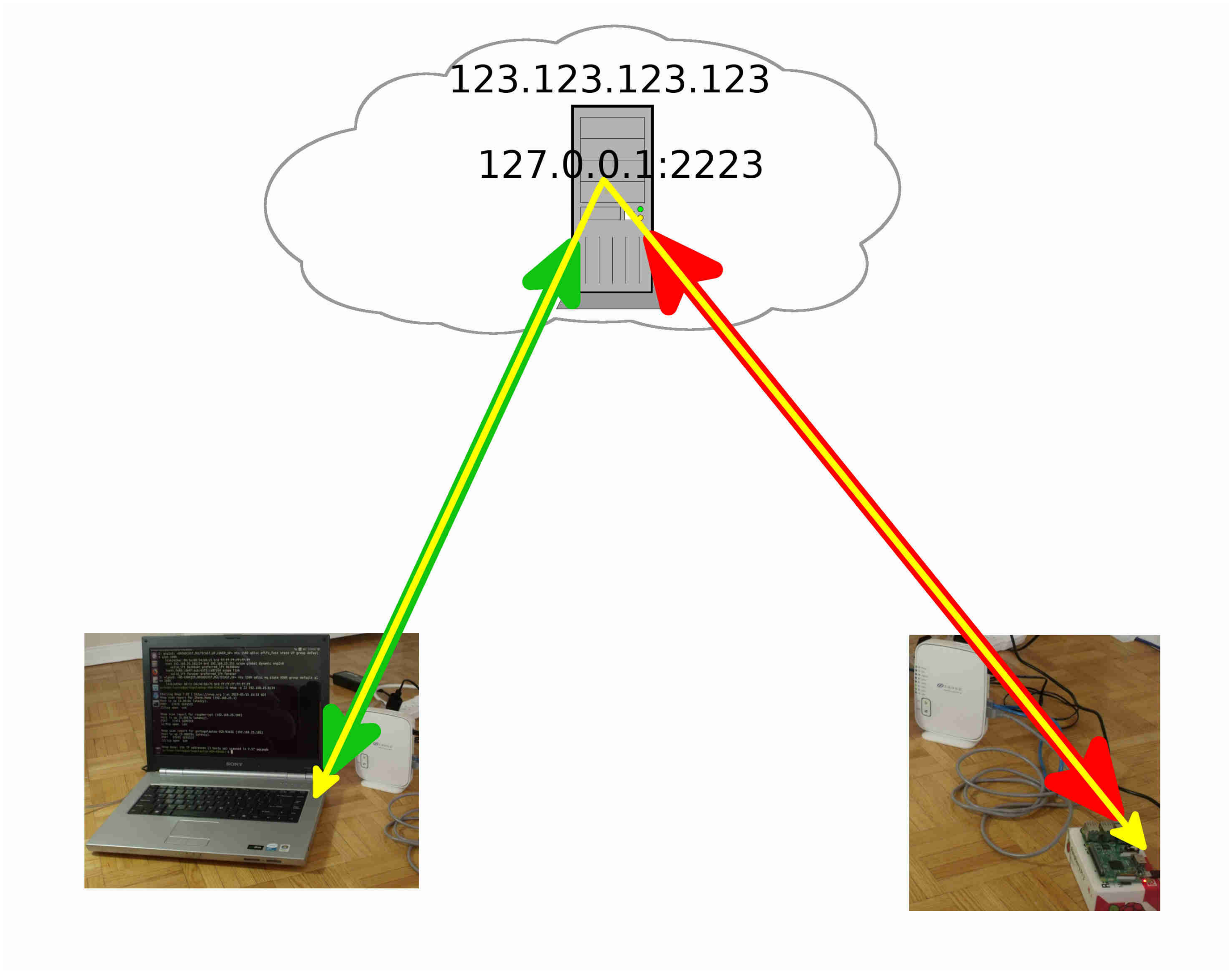
For more reading on this topic, consult Q/A on Assigning a fixed IP address to a machine in a DHCP network.

Making it Work Over The Internet

In this section, we will discuss one method of using SSH to remotely log into another computer from anywhere that you can access the internet. There are actually many different ways to accomplish this task in general, but this article will focus on showing you a method that involves setting up a proxy server with your favourite cloud provider, and then tunneling the connection to the Raspberry Pi through the proxy server, and down to your Raspberry Pi using SSH and port forwarding.

However, before you rush off to set this up, I recommend that you make sure you know what you're doing when setting up this kind of configuration. If you do something incorrectly, you could be exposing you and your data to every hacker out there on the internet! This is because the proxy (the one that you setup and control) will need to be accessible via a public IP on the open internet.

In order to get started with this technique, you will have to set up your own proxy server using one of the many popular cloud providers. This is something you will have to pay for each month, but a service like Amazon Web Services is fairly cheap if you go with a small sized instance. It will likely cost you less than $5.00 per month to set one up. It doesn't have to be powerful, it just needs to be a simple machine with network access and Linux installed. To give you an idea of what the setup will look like once you have it up and running, here is an illustration:



The setup described in the picture above is one where the laptop and the Raspberry Pi are in different countries and have their own local LAN connections. Both of them connect to the internet through their own router on their own LAN (not the *same* router as might be suggested by the way the pictures were cropped). Both the Laptop and the Raspberry Pi will set up their own SSH connection to the proxy. In this example, we will assume that the proxy server has public IP address '123.123.123.123'.

In order to achieve the final solution depicted above, you must create 3 SSH key pairs. The first is created on your Laptop with the public key added to '~/.ssh/authorized_keys' on the proxy server ('my-proxy-keypair-laptop.pub'). The second is created on the Raspberry Pi with the public key added to '~/.ssh/authorized_keys' on the proxy server ('my-proxy-keypair-pi.pub'). The third is created on the Laptop with the public key added to '~/.ssh/authorized_keys' on the Raspberry Pi ('my-first-keypair.pub'). If you need to refresh your memory about how to set up SSH keypairs, see the article How To Set Up and Distribute SSH Key Pairs.

Let's review the three SSH commands that you'd need to run to set up a connection from your laptop to the Raspberry Pi through the proxy if you did it manually every time you wanted to set up the tunnel:

```
#  Represented by the Green arrow in the picture. (Run this command from Laptop)
#  Forward the Laptop's incomming traffic to port 2222 through the
#  tunnel and send it to 127.0.0.1:2223 once inside the proxy server.
ssh -L 2222:127.0.0.1:2223 -i ~/.ssh/my-proxy-keypair-laptop admin@123.123.123.123

#  Represented by the Red arrow in the picture. (Run this command from Raspberry
Pi)
#  SSH into to the proxy server, then forward any incoming traffic destined
#  for port 2223 (on the proxy server), down the tunnel, and send it to
#  port 22 on the Raspbery Pi (127.0.0.1:22)
ssh -R 2223:127.0.0.1:22 -i ~/.ssh/my-proxy-keypair-pi admin@123.123.123.123

#  Represented by the Yellow arrow in the picture. (Run this command from Laptop)
#  Open an SSH connection, but try to connect to 127.0.0.1:2222 with the
#  user 'pi'.  Since the laptop's port 2222 is forwarded onto the proxy
#  servers's port 2223, and port 2223 on the proxy is forwarded back down
#  to the Raspberry Pi, this forwards the SSH connection request down to
#  the Raspberry pi.
ssh -p 2222 -i ~/.ssh/my-first-keypair pi@127.0.0.1
```

Pay special attention to the red, green and yellow arrows in the picture. Each of them corresponds to a different SSH command that is shown above. These three commands show how you can connect to the Raspberry Pi over the internet by typing these commands manually on the command-line, but what we want is a more automatic solution. We also need to set up ssh configs that let us SSH directly onto the pi without having to manually set up the dependent tunnel and port forwarding rules. Furthermore, we need a way to make sure that the Raspberry Pi keeps trying to connect to the proxy server even if the power goes out, the local internet goes down, or some other event occurs that disrupts connectivity. Here is an equivalent ssh config file you can use on your laptop:

```
Host my-proxy
    HostName 123.123.123.123
    Port 22
    User admin
    IdentityFile ~/.ssh/my-proxy-keypair-laptop
    LocalForward 2222 127.0.0.1:2223
Host pi-backup
    HostName 127.0.0.1
    Port 2222
    User robert
    IdentityFile ~/.ssh/my-first-keypair
    ProxyJump my-proxy
```

Putting this in your ~/.ssh/config file on your laptop will cause the SSH tunnel to the proxy to be automatically set up whenever you try to ssh to 'pi-backup'. These two host entries will take care of the first half of the green arrow and half of the yellow one. However, we still have to do something on the Raspberry Pi to allow incoming connections that come through the proxy server. For this, you can create a ~/.ssh/config rule on the Raspberry Pi:

```
Host my-proxy
    HostName 123.123.123.123
    User admin
    Port 22
    IdentityFile ~/.ssh/my-proxy-keypair-pi
    ServerAliveInterval 60
```

The above SSH host rule will set up a connection to the proxy server, but we haven't included the directive to make it actually listen for incoming connections from our laptop yet. For that part, we'll include it as part of a script that can run regularly to make sure that the tunnel is active and working:

```
#!/bin/bash

LOCAL_SSH_PORT=22
REMOTE_SSH_PORT=2223

LOCAL_SSH_ADDRESS=127.0.0.1
REMOTE_SSH_ADDRESS=127.0.0.1
PROXY_HOST=my-proxy

SSH_COMMAND="ssh -q -N -R
${REMOTE_SSH_ADDRESS}:${REMOTE_SSH_PORT}:${LOCAL_SSH_ADDRESS}:${LOCAL_SSH_PORT}
```

```
${PROXY_HOST}"
LOG_FILE=~/.ssh/tunnel-log.log

#  Do a request to a site we own regularly so we can remotely check if the
Raspberry Pi is alive and what it's public IP is.
curl https://www.example.com/?rpi_checkin=true > /dev/null 2>&1

#  Check if there was a tunnel launched with the same command, otherwise, start one
if [ $(pgrep -f -x "$SSH_COMMAND" | wc -l) -eq 0 ]; then
   $SSH_COMMAND &
   PREV_PID=$!
   echo "$(date): No tunnel was active: starting tunnel with pid: ${PREV_PID}." >>
"${LOG_FILE}"
fi

MATCHING_LISTENS=$(ssh ${PROXY_HOST} "netstat -an" | egrep
"tcp.*${REMOTE_SSH_ADDRESS}:${REMOTE_SSH_PORT}.*LISTEN")
LISTEN_CHECK_RTN=$?
#  Count listens using awk, wc -l won't work if there is no trailing newline.
NUM_MATCHING_LISTENS=$(echo -en "${MATCHING_LISTENS}" | awk 'END{print NR}')

#  Re-start the tunnel if we can't ssh into the remote and verify that the tunnel
is working
if [ ${LISTEN_CHECK_RTN} -ne 0 ] ; then
   pkill -f -x "$SSH_COMMAND"
   $SSH_COMMAND &
   PREV_PID=$!
   echo "$(date): Failed to check for active tunnel, restarted tunnel.  New pid:
${PREV_PID}." >> "${LOG_FILE}"
elif [ ${NUM_MATCHING_LISTENS} -lt 1 ] ; then
   pkill -f -x "$SSH_COMMAND"
   $SSH_COMMAND &
   PREV_PID=$!
   echo "$(date): No matching listens found in proxy server, restarted tunnel.  New
pid: ${PREV_PID}." >> "${LOG_FILE}"
else
   echo "$(date): Tunnel appears to be active on remote, do nothing." >>
"${LOG_FILE}"
fi
```

You can put this script on the Raspberry Pi, and then run it from a cron job, or manually on the command-line. Every time the script runs, it will check to make sure there is an active SSH tunnel that is also forwarding traffic from 127.0.0.1:2223 on the proxy back down to port 22 on the Raspberry Pi. This script also does a couple other things like log what is happening (which is useful for debugging purposes), and it also regularly pings some other host that we control for reasons that will be explained later.

One security related note about using the ssh -R flag is that this command can pose a potential security risk: If you were to SSH into a remote host and then use -R to bind to a port that was listening on a public network interface, you could end up forwarding traffic from the open internet down to your Raspberry Pi. Because of this, the default SSH config usually doesn't let you use the -R flag to bind to interfaces other than localhost. This can be changed with the 'GatewayPorts' directive in /etc/ssh/sshd_config, but this is not recommended for what we're doing here. If you want to be ultra safe, you should ensure that you aren't listening on a public interface on the proxy server for the incomming Raspberry Pi connections. However, because the proxy server is going to be somewhere in the cloud, it *will have to* be listening on a public interface for SSH connections to the proxy server iteself! To mitigate the risk of hacking attempts on the proxy server, I would strongly recommend setting up firewall rules to explicitly whitelist both your own IP address, and the IP address of where the Raspberry Pi is. What happens if your IP changes though? Well, you'll just have to update the firewall rule, or losen the security by only allowing IP addresses from a larger address range that is guaranteed to be on your ISP's network, but not open to the entire internet. When the IP address of the Raspberry Pi changes, you won't be present to check the IP address, so that's why the script listed above includes an 'rpi_checkin' curl request. If the Raspberry Pi address changes, you can check your logs on a simple web server that you set up somewhere (for example on the proxy server itself), and then see what IP it's talking to you from and white-list that.

How to Set Up A Proxy Server?

I haven't yet explained how to set up your proxy server if you decide you need to connect to your Raspberry Pi over the internet. The exact solution will depend on the cloud provider you use, but if you'd like an explanation of how to launch a simple server on Amazon Web Services, you can check out the guide on Amazon Cloud Servers For Beginners: Console VS Command-Line. This guide explains how to launch servers from the web interface and the command-line. You'll only need to launch the server once, so you can follow the instructions on how to launch a server through the web interface. You can ignore pretty much everything about using the 'AWS Command Line Interface', and the section 'Launching An Amazon Server Via The Command Line'. You should, however, follow the steps in the section 'Connecting To Your Server With SSH' to verify that you can actually connect to the server. This guide also shows you a bit of information about security groups in AWS, although you should consider using a rule that restricts SSH access to 'My IP address' instead of '0.0.0.0/0' as shown in the guide which would allow full public access to SSH login attempts.

Automation Using Cron

Cron jobs are fast and easy way to automate various tasks on a Linux/Unix system. In order to set up a cron job, you can open up the crontab editor and edit up your user's cron file where you use a special syntax to describe what Linux command you want to run, and when you want it to run. The first time you try to edit your crontab file, it will usually ask you what editor you want to use. I would suggest using nano if you're not experienced with command-line editors yet.

```
crontab -e
```

Here is an example of a cron file that has a single entry that will run the script 'do-backup.sh' once per day at 6:01pm:

```
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command
1 18 * * * /home/robert/do-backup.sh
```

The lines that start with '#' are just comments.

For the actual backup script, you can use something like this:

```
#!/bin/bash

cd ~/my-repo
git push pi-backup master
```

Just replace the directory name, the remote and the branch name with whatever git repo you can to push.

The syntax for cron jobs is easy to forget, and it can also get more complicated. A really good site for remembering cron syntax can be found on https://crontab.guru/.

You can also use a cron job to regularly run the script provided in the section 'Making it Work Over The Internet' to make sure that the Raspberry Pi will regularly make sure that the remote connection tunnel is always listening for new SSH connections even if the tunnel dies, the power reset, or something else breaks the connection. Assuming that you put the script inside a file called '~/connection-script.sh' you could do:

```
5 * * * * ~/connection-script.sh
```

Which will run the connection keep-alive script every hour, 5 minutes after the hour.

Another use for cron is to create a rule to periodically install updates, although there is also an automatic updates feature that may be better suited to this purpose.

Flash Storage Issues

A very common problem experienced by people who use digital cameras, the Raspberry Pi, or USB flash drives is flash memory corruption. There are several contributing factors that can lead to flash memory corruption, but one of the most important is that most consumer flash memory is just poorly designed and manufactured. As of writing this article, you can pick up 16GiB of flash memory at an office supply store for about $10. These cheap consumer SD cards are often not well designed or manufactured and it is actually quite common for them to fail. Industrial SD cards, such as those you can purchase from Digikey are more reliable, but they can be *extremely* expensive especially for the larger capacity ones.

Before diving into the details of what can cause 'flash corruption', it's important to distinguish between two different types of 'corruption' that can occur. The following two terms have been coined mainly just for use in this article so they aren't universal, but let's divide up the types of flash 'corruption' you can experience into **soft corruption** and **hard corruption**.

Soft Corruption is meant to describe 'software' based corruption of the information stored on the device without any physical issue with the device itself. In this type of corruption, your data may be lost of damaged, but you can likely re-format the device and it will work as good as new again. For example, you can sometimes end up with 'soft corruption' if you copy something to a flash drive, and then quickly pull out the flash device before it has finished writing, or without doing the 'safely remove' feature. In this case, when you try to read the drive again, it might say you need to format it, or the files may not open. In this case you can often format the device and use it again to store files just fine. With soft corruption, you may be able to use various software tools to read all valid and invalid data from the media and then perform manual steps to reconstruct the filesystem structure. This job is often performed by data recovery specialists, and it can be very time consuming or expensive.

Hard Corruption is meant to describe 'hardware' based corruption (permanent physical effects) that prevent the flash device from reliably storing information. With hard corruption, you can expect your data to be lost or damaged, and any kind of formatting or software-based repair tool won't have any hope of making the device reliable again. Hard corruption of flash devices will occur naturally over time with repeated write cycles as the electrostatic mechanisms used to actually 'store' and represent data in the device become more and more noisy. Other forms of hard corruption could be things like electrical shorts that damage the silicon chip inside the SD card, or physical changes to the memory cells in the chip that result from intense static electricity discharge.

Cheap flash memory has been known to sometimes experience soft corruption even with very specific write sequences that you might encounter from heavy Raspberry Pi use. This can be because of poor programming and testing on the part of the SD card manufacturer who may only test their firmware for usage cases that involve only the most common and predictable write patterns. They may assume that most people using the memory will be doing things like taking photos or video with a digital camera. These trivial test cases are enough to cover the most common uses of consumer memory and are therefore 'good enough' to be able to sell the product.

Common Causes of Flash Memory Corruption

In addition to cheap memory being to blame, there are number of other contributing factors and many of them have to do with not having a stable power supply. To summarize, a list of many problems that can lead to corrupted flash memory are:

- Removing the flash media from the Raspberry Pi/Camera/Computer while it is running without safely unmounting it, or using the 'remove safely' feature.
- Buying the cheapest SD card you can find on eBay.
- Power outages, brownouts or voltage surges.
- Using a poor quality USB wall wart adapter. See YouTube search for 'usb adapter teardown'.
- Using a USB cable with wires that are too thin or don't make proper contact when plugged in (causing high wire resistance and voltage drop).
- Using a good quality USB cable and adapter that has an inadequate current rating for your use case (ex. using a 1A adapter when 2A is required to run the Pi, + camera + HD during 100% CPU usage).
- Using extremely high-density flash which is often more prone to failure than low-density flash.
- Using flash memory which, internally, does not have mechanisms (such as write-ahead logging or transactions) to recover from power loss or power sag.
- Electrostatic discharge (static electricity shocks).
- Writing/rewriting to the flash memory too many times.

Many of the power related failure cases described above are likely to cause soft corruption where the data will be lost or damaged, but the flash media can often be used again after a simple re-formatting. The reason for this is that the 'corruption' can manifest itself in a few bits of information that describe the layout of files on the flash device, rather than the actual data you intend to store. When a device is 'formatted' with a filesystem, various entries are written to the device which describe things like: how many files there are, the start and end of each file, the length of filenames, directory structure, etc. If you power off a poor-quality flash device that doesn't expect to be interrupted in the middle of changing the filesystem structure, it could end up with an incomplete listing of files that doesn't make any sense. Then, whatever device tries to read it will notice that the filesystem says things that don't make sense, so your OS will say: "I don't know what to do with this directory that claims to have 7573259375438759843758437597435843543 files in it (that's impossible), so I'm just going to suggest that you format the flash device instead and not even bother to try and figure out what is wrong.".

Another effect that can contribute to soft corruption is that your Raspberry Pi, Camera, or other device, usually caches writes by holding them in some other part of memory before writing large sections to the flash media all at once. The programming that actually writes data from the Pi, camera, etc. might do 'writes' one byte at a time. If the OS actually did writes to the flash media one byte at a time, this would be very slow and wear out the flash much faster. Instead, your OS will usually cache the writes in memory until there is enough of them to bother writing one big chunk of data at once. The trouble is, if the OS is only half-finished writing changes from the cache to the flash storage and you then power off the device, the other half of un-written cached data that was in memory is now lost. Even worse, the half that was written probably doesn't make sense and can contribute to corrupted filesystem structures.

It's also worth pointing out that you can encounter 'soft' corruption that is technically still just bits flipped in software states, but still have what most consumers would consider 'hard' corruption. This is because along with your data, the manufacturer needs to program the flash storage with small computer programs and data structures that only they have the expertise to work with (flash memory actually contains a small CPU with its own software!). If you can find out what tools and processes the manufacturer uses internally, you might have a shot, but if a bit gets flipped inside the flash memory's internal firmware, some internal data structure, or an error correction algorithm code, then you're probably going to have a tough time fixing this unless you work for the company who makes them. It's worth mentioning that one of the suggestions above about not buying cheap flash memory from eBay is relevant here: Some of the super cheap '1TB' memory cards you buy are actually much smaller (512MB for example) cards where some of the internal data structures have been updated to simply report some incredibly huge partition size instead of the real size. When you plug them into a computer, it will say the actual size is '1TB' because it gets this size information by asking for it from the SD card. When the card has been purposefully re-programmed to lie about how large it is, you can make it say any size you want! If you're interested in this topic, I suggest reading On Hacking MicroSD Cards.

Solutions To Flash Memory Corruption

In order to mitigate these problems, you should consider doing the following:

- Make sure you always unmount ('safely remove') your flash media before physically removing it.
- If you have a choice between buying high-density flash (high GB/$) and low-density flash (low GB/$) for the same amount of money, pick the lower density one. You'll get less 'storage' per dollar, but the integrity of the flash per bit is likely to be higher.
- Try not to buy cheap consumer grade flash memory.
- Become aware of approximately how many amps your device will consume with specific consideration to current spikes that can happen when it needs to do a lot of work quickly. For a Raspberry Pi, you should consider how many peripherals are attached to it. Make sure you get a wall wart adapter that is rated for the max amperage you plan to use.
- If you suspect that you're close the current limit that your wall adapter supports, avoid doing workloads that max out the CPU at 100%, or activate multiple peripherals at once.
- Minimize the number of writes to your flash to maximize lifetime. For a Raspberry Pi, you should consider doing as much work in memory, without touching disk, as possible.
- If you have cheap consumer flash, try not to create high random read/write workloads, or unusual read/write patterns.
- Avoid static electricity discharges on the pins of the flash memory, especially on the pins that are used to transfer data.

Many of the internal details of the flash memory features and functionality (such as density, power outage recovery, programming quality) are not things you will be able to easily discover for most pieces of cheap consumer flash memory since they are often 'hidden' and proprietary. It is reasonable to suggest that they may even change between different batches of the same product under the same model number from a given manufacturer.

If you have a serious project that requires using flash memory, you should be sure to read up on the difference between SLC, MLC and TLC flash. Keep in mind that the difference between SLC, MLC and TLC between different vendors is unlikely to be an apples to apples comparison. Evaluating flash memory at this level of detail is more about the physics used in the individual manufacturing and programming processes of the chip itself than any kind of well standardized interpretation of what it means for something to be considered 'SLC'.

Another important detail worth considering in how you make use of flash from a programming perspective (and also how the flash may be organized internally) is the concept of write amplification. Write amplification involves the consideration of how the number of writes required to commit information can 'amplify' because of the need to update different data structures, rearrange data, and also satisfy block-level operation constrains.

I can personally vouch for one of the Raspberry Pi flash SD cards that came with the 'CanaKit' Raspberry Pi kit I purchased several years ago. This Raspberry Pi that has been operating without any issues for several years now. Having said that, I don't know if it has really experienced any power failure or brownout conditions and these would really be the important stress test to consider. I have also used a slightly more expensive SD card recently (AF8GUD3) for some of my newer Raspberry Pi project which I purchased from DigiKey, and I haven't had any problems with this one.

If you're interested in learning more about the details of flash memory corruption, an advanced overview requires a detailed understanding of the physics involved in the individual manufacturing techniques used for the particular model of memory you are using. Here is an excellent talk on the subject: Tutorial: Why NAND Flash Breaks Down, from the The Linux Foundation's YouTube channel.

Using An External USB Disk

One way to avoid using flash completely is to use an external hard drive to host the data that you're backing up. When you plug in most USB hard drives, you can usually find out where they have mounted by using the 'df' command. You'll see output like this:

```
Filesystem      1K-blocks       Used  Available Use% Mounted on
udev             10200812          0   10200812   0% /dev
tmpfs             2046288       1204    2045084   1% /run
/dev/sda1       921923300  589451908  285570556  68% /
/dev/sdb1      3844607992      90140 3649152324   1% /mnt
```

Where, in this case, the 'sdb1' entry is the external USB hard disk. In your case, the device name will be different, but it will sometimes auto-mount to '/mnt'. If you don't see your external USB in the output of 'df', then it might not be mounted. In order to mount it, you'll need to use a tool like 'fdisk' which can list off all storage devices, even ones that are not mounted. Explaining fdisk is beyond the scope of this article, but if you do end up using it just make sure you read the documentation. Fdisk is able to modify partition tables of your storage devices, and **if you accidentally edit a partition table of one of your storage devices, you could lose all your data!**. After you find out which storage device is your USB disk, you can use the 'mount' command to manually mount it.

However, there is a problem with using the 'mount' command to manually mount the USB disk: You may need to manually re-mount it every time you reboot the Raspberry Pi, otherwise, when your script tries to push data to a git repo stored on the disk that isn't mounted, it will fail.

You can fix this problem by editing the '/etc/fstab' file and instructing it to auto-mount the USB disk every time the Raspberry Pi starts. One draw-back of editing the fstab file is that, by default, it will interrupt the boot process if the disk is not present when it tries to mount. This makes sense when for a server with an internal hard disk, but for a removable USB drive that you may take out every once in a while, it can be annoying. Therefore, you can use a special 'nofail' option in the fstab entry to prevent it from hanging up the boot process:

```
UUID=1234XXXX-AAAA-BBBB-CCCC-DDDDEEEEFFFF /mnt-my-USB ext4 defaults,nofail 0 0
```

Also, be very careful when editing your fstab file, and make sure you know what you're doing. If you accidentally switch where your disks are mounted or break your boot process, it may cause mistakes that lead to data loss.

Finally, in order to add these entries in a way that is consistent under different race conditions from which device is detected first, use the UUID based method of identifying devices. You can find the UUIDs of devices with the 'blkid' command:

```
blkid /dev/sda1
```

Various Troubleshooting

If you encounter trouble getting your SSH connections to work, especially when using tunneling through the proxy server, a very useful command to run is:

```
netstat -an
```

You may want to pipe the result of this command into less so you can look at the result easier (press 'q' to exit):

```
netstat -an | less
```

The results of running this command will look something like this:

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0     88 123.123.123.123:22      124.124.124.124:52032   ESTABLISHED
tcp6       0      0 :::22                   :::*                    LISTEN
Active UNIX domain sockets (servers and established)
...
```

The example output above is what it looks like when I am SSH'ed into one of my servers. Pay special attention to the port numbers and states of each of the connections. In the above output, we can see that the first entry is 'tcp' (aka IPV4 TCP) and is listening for new connections from packets that have a destination port of 22, and are destined for any interface (0.0.0.0:22). Furthermore, we are listening for connections from any IP with any source port.

In the second entry, we see that there is an ESTABLISHED from my laptop which has IP 124.124.124.124 originating from port 52032 on my laptop (124.124.124.124:52032). This connection is sending packets to the server at 123.123.123.123 to port 22 (no surprise because that's the port for SSH connections).

In the third entry we see another listen socket for 'tcp6' which just means it is also listening for IPV6 connections too. In the above output, there are not remote forwarded tunnels set up, and you'll see more entries when there are. It may take you a while to get used to reading this output, but eventually you'll be able to glance at it and tell what is connected, what's waiting for connections, and what is unrelated.

Another thing you should do if you're having troubling setting up your SSH connection is use verbose mode when invoking SSH itself. You can enable full verbose mode with the '-vvv-' flag:

```
ssh -vvv pi-backup
```

Here is an example of the kind of output you might see:

```
robert@computer:~$ ssh -vvv pi-backup
openSSH-10.3 Ubuntu-ubuntu0.4, OpenSSL 1.1.3e  21 Nov 2009
debug1: Reading configuration data /home/robert/.ssh/config
debug1: /home/robert/.ssh/config line 3: Applying options for pi-backup
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 11: Applying options for *
...
```

Depending on what your problem is, you may be able to glean some useful information from the output that can help solve your problem.

Conclusion

In this article we've discussed many topics related to hosting a backup solution using your Raspberry Pi or spare laptop. This includes simple situations that only require communication with a Raspberry Pi hosted on the same LAN, but also more complex situations that require the connection to go over the internet. Concerns like flash memory corruption were discussed with the conclusion that you should avoid buying the absolute rock bottom cheapest flash memory, and also make sure you use a good power supply. A method of automating the backup 'push' operation was discussed that involves using cron jobs.