4
Encrypting and SSH Hardening

You may work for a super-secret government agency, or you may be just a regular Joe or Jane citizen. Either way, you will still have sensitive data that you need to protect from prying eyes. Business secrets, government secrets, personal secrets—it doesn't matter; it all needs protection. Locking down user's home directories with restrictive permissions settings, as we saw in `Chapter 2`, *Securing User Accounts*, is only part of the puzzle; we also need encryption.

The two general types of data encryption that we'll look at in this chapter are meant to protect *data at rest* and *data in transit*. We'll begin with using file, partition, and directory encryption to protect data at rest. We'll then cover **Secure Shell** (**SSH**) to protect data in transit.

In this chapter, we'll cover:

- **GNU Privacy Guard** (**GPG**)
- Encrypting partitions with **Linux Unified Key Setup** (**LUKS**)
- Encrypting directories with eCryptfs
- Using VeraCrypt for the cross-platform sharing of encrypted containers
- Ensuring that SSH protocol 1 is disabled
- Creating and managing keys for password-less logins
- Disabling root user login
- Disabling username/password logins
- Setting up a chroot environment for SFTP users

GNU Privacy Guard

We'll begin with **GNU Privacy Guard** (**GPG**). This is a free open source implementation of Phil Zimmermann's *Pretty Good Privacy*, which he created back in 1991. You can use either one of them to either encrypt or cryptographically sign files or messages. In this section, we'll focus strictly on GPG.

There are some advantages of using GPG:

- It uses strong, hard-to-crack encryption algorithms.
- It uses the private/public key scheme, which eliminates the need to transfer a password to a message or file recipient in a secure manner. Instead, just send along your public key, which is useless to anyone other than the intended recipient.
- You can use GPG to just encrypt your own files for your own use, the same as you'd use any other encryption utility.
- It can be used to encrypt email messages, allowing you to have true end-to-end encryption for sensitive emails.
- There are a few GUI-type frontends available to make it somewhat easier to use.

But, as you might know, there are also some disadvantages:

- Using public keys instead of passwords is great when you work directly only with people who you implicitly trust. But, for anything beyond that, such as distributing a public key to the general population so that everyone can verify your signed messages, you're dependent upon a web-of-trust model that can be very hard to set up.
- For the end-to-end encryption of email, the recipients of your email must also have GPG set up on their systems, and know how to use it. That might work in a corporate environment, but lots of luck getting your friends to set that up. (I've never once succeeded in getting someone else to set up email encryption.)
- If you use a standalone email client, such as Mozilla Thunderbird, you can install a plugin that will encrypt and decrypt messages automatically. But, every time a new Thunderbird update is released, the plugin breaks, and it always takes a while before a new working version gets released.

Even with its numerous weaknesses, GPG is still one of the best ways to share encrypted files and emails. GPG comes preinstalled on both Ubuntu Server and CentOS. So, you can use either of your virtual machines for these demos.

Creating your GPG keys

Getting started with GPG requires you to first generate your GPG keys. You'll do that with:

```
gpg --gen-key
```

Note that, since you're setting this up for yourself, you don't need sudo privileges.

The output of this command is too long to show all at once, so I'll show relevant sections of it, and break down what it means.

The first thing that this command does is to create a populated `.gnupg` directory in your home directory:

```
gpg: directory `/home/donnie/.gnupg' created
gpg: new configuration file `/home/donnie/.gnupg/gpg.conf' created
gpg: WARNING: options in `/home/donnie/.gnupg/gpg.conf' are not yet active
during this run
gpg: keyring `/home/donnie/.gnupg/secring.gpg' created
gpg: keyring `/home/donnie/.gnupg/pubring.gpg' created
```

You'll then be asked to select which kinds of keys you want. We'll just go with the default RSA and RSA. (RSA keys are stronger and harder to crack than the older DSA keys. Elgamal keys are good, but they may not be supported by older versions of GPG.):

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
```

For decent encryption, you'll want to go with a key of at least 2048 bits, because anything smaller is now considered vulnerable. Since 2048 just happens to be the default, we'll go with it:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
```

Next, select how long you want for the keys to remain valid before they automatically expire. For our purposes, we'll go with the default `key does not expire`.

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

Provide your personal information:

```
GnuPG needs to construct a user ID to identify your key.

Real name: Donald A. Tevault
Email address: donniet@something.net
Comment: No comment
You selected this USER-ID:
    "Donald A. Tevault (No comment) <donniet@something.net>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
```

Create a passphrase for your private key:

```
You need a Passphrase to protect your secret key.

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
```

This could take a while, even when you're doing all of the recommended things to create entropy. Be patient; it will eventually finish. By running a `sudo yum upgrade` in another window, I created enough entropy so that the process didn't take too long:

```
gpg: /home/donnie/.gnupg/trustdb.gpg: trustdb created
gpg: key 19CAEC5B marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   2048R/19CAEC5B 2017-10-26
      Key fingerprint = 8DE5 8894 2E37 08C4 5B26  9164 C77C 6944 19CA EC5B
uid                  Donald A. Tevault (No comment) <donniet@something.net>
sub   2048R/37582F29 2017-10-26
```

Verify that the keys did get created:

```
[donnie@localhost ~]$ gpg --list-keys
/home/donnie/.gnupg/pubring.gpg
-------------------------------
pub   2048R/19CAEC5B 2017-10-26
uid                  Donald A. Tevault (No comment) <donniet@something.net>
sub   2048R/37582F29 2017-10-26

[donnie@localhost ~]$
```

And, while you're at it, take a look at the files that you created:

```
[donnie@localhost ~]$ ls -l .gnupg
total 28
-rw-------. 1 donnie donnie 7680 Oct 26 13:22 gpg.conf
drwx------. 2 donnie donnie    6 Oct 26 13:40 private-keys-v1.d
-rw-------. 1 donnie donnie 1208 Oct 26 13:45 pubring.gpg
-rw-------. 1 donnie donnie 1208 Oct 26 13:45 pubring.gpg~
-rw-------. 1 donnie donnie  600 Oct 26 13:45 random_seed
-rw-------. 1 donnie donnie 2586 Oct 26 13:45 secring.gpg
srwxrwxr-x. 1 donnie donnie    0 Oct 26 13:40 S.gpg-agent
-rw-------. 1 donnie donnie 1280 Oct 26 13:45 trustdb.gpg
[donnie@localhost ~]$
```

These files are your public and private keyrings, your own `gpg.conf` file, a random seed file, and a trusted users database.

Symmetrically encrypting your own files

You may find GPG useful for encrypting your own files, even when you never plan to share them with anyone else. For this, you'll use symmetric encryption, which involves using your own private key for encryption. Before you try this, you'll need to generate your keys, as I outlined in the previous section.

Symmetric key encryption is, well, just that, symmetric. It's symmetric in the sense that the same key that you would use to encrypt a file is the same key that you would use to decrypt the file. That's great for if you're just encrypting files for your own use. But, if you need to share an encrypted file with someone else, you'll need to figure out a secure way to give that person the password. I mean, it's not like you'd want to just send the password in a plain-text email.

Let's encrypt a super-secret file that we just can't allow to fall into the wrong hands:

```
[donnie@localhost ~]$ gpg -c secret_squirrel_stuff.txt
[donnie@localhost ~]$
```

Note that the `-c` option indicates that I chose to use symmetric encryption with a passphrase for the file. The passphrase that you enter will be for the file, not for your private key.

One slight flaw with this is that GPG makes an encrypted copy of the file, but it also leaves the original, unencrypted file intact:

```
[donnie@localhost ~]$ ls -l
total 1748
-rw-rw-r--. 1 donnie donnie      37 Oct 26 14:22 secret_squirrel_stuff.txt
-rw-rw-r--. 1 donnie donnie      94 Oct 26 14:22
secret_squirrel_stuff.txt.gpg
[donnie@localhost ~]$
```

Let's get rid of that unencrypted file with `shred`. We'll use the `-u` option to delete the file, and the `-z` option to overwrite the deleted file with zeros:

```
[donnie@localhost ~]$ shred -u -z secret_squirrel_stuff.txt
[donnie@localhost ~]$
```

It doesn't look like anything happened, because `shred` doesn't give you any output. But, an `ls -l` will prove that the file is gone. Now, if I were to look at the encrypted file with `less secret_squirrel_stuff.txt.gpg`, I would be able to see its contents, after being asked to enter my private key passphrase:

```
Shhh!!!!  This file is super-secret.
secret_squirrel_stuff.txt.gpg (END)
```

As long as my private key remains loaded into my keyring, I'll be able to view my encrypted file again without having to reenter the passphrase. Now, just to prove to you that the file really is encrypted, I'll create a shared directory, and move the file there for others to access:

```
sudo mkdir /shared
sudo chown donnie: /shared
sudo chmod 755 /shared
mv secret_squirrel_stuff.txt.gpg /shared
```

When I go into that directory to view the file with `less`, I can still see its contents, without having to reenter my passphrase. But now, let's see what happens when Maggie tries to view the file:

```
[maggie@localhost shared]$ less secret_squirrel_stuff.txt.gpg
"secret_squirrel_stuff.txt.gpg" may be a binary file.  See it anyway?
```

And when she hits the *Y* key to see it anyway:

```
<8C>^M^D^C^C^B<BD>2=<D3>u<93><CE><C9>MOOy<B6>^O<A2><AD>}Rg9<94><EB><C4>^W^E
<A6><8D><B9><B8><D3>(<98><C4>æF^_8Q2b<B8>C<B5><DB>^]<F1><CD>#<90>H<EB><90><
C5>^S%X  [<E9><EF><C7>
^@y+<FC><F2><BA><U+058C>H'+<D4>v<84>Y<98>G<D7>~
secret_squirrel_stuff.txt.gpg (END)
```

Poor Maggie really wants to see my file, but all she can see is encrypted gibberish.

What I've just demonstrated is another advantage of GPG. After entering your private key passphrase once, you can view any of your encrypted files without having to manually decrypt them, and without having to reenter your passphrase. With other symmetric file encryption tools, such as Bcrypt, you wouldn't be able to view your files without manually decrypting them first.

But, let's now say that you no longer need to have this file encrypted, and you want to decrypt it in order to let other people see it. Just use `gpg` with the `-d` option:

```
[donnie@localhost shared]$ gpg -d secret_squirrel_stuff.txt.gpg
gpg: CAST5 encrypted data
gpg: encrypted with 1 passphrase
Shhh!!!!  This file is super-secret.
gpg: WARNING: message was not integrity protected
[donnie@localhost shared]$
```

The `WARNING` message about the message not being integrity protected means that I had encrypted the file, but I never signed the file. Without a digital signature, someone could alter the file without me knowing about it, and I wouldn't be able to prove that I am the originator of the file. (Have no fear, we'll talk about signing files in just a bit.)

Hands-on lab – combining gpg and tar for encrypted backups

For this lab, you'll combine `tar` and `gpg` to create an encrypted backup on a simulated backup device. You can perform this lab on either one of your virtual machines:

1. Start off by creating your GPG keys. You will do that with the following command:

```
gpg --gen-key
```

2. Create some dummy files in your home directory, so that you'll have something to back up:

```
touch {file1.txt,file2.txt,file3.txt,file4.txt}
```

3. Create a backup directory at the root level of the filesystem. (In real life, you would have the backup directory on a separate device, but for now, this works.) Change ownership of the directory to your own account, and set the permissions so that only you can access it:

```
sudo mkdir /backup
sudo chown your_username: /backup
sudo chmod 700 /backup
```

4. Create an encrypted backup file of your own home directory. Compression is optional, but we'll go ahead and use `xz` for the best compression. (Note that you'll need to use `sudo` for this, because the `.viminfo` directory in your home directory is owned by the root user.):

```
cd /home
sudo tar cJvf - your_username/ | gpg -c >
/backup/your_username_backup.tar.xz.gpg
```

5. Now, let's say that either your home directory got deleted, or that you accidentally deleted some important files from your own home directory. Extract and decrypt the original home directory within the `/backup` directory:

```
cd /backup
sudo gpg -d your_username.tar.xz.gpg | tar xvJ
ls -la your_username/
```

Note that, by combining `tar` with `gpg`, the `-C` option of `tar` to automatically place your home directory back within the `/home` directory won't work. So, you'll either need to manually copy the extracted directory back to `/home`, or move the encrypted backup file to `/home` before you extract it. Also, be aware that when you extract an encrypted archive with `gpg`, the ownership of the files will change to that of whoever extracted the archive. So, this probably wouldn't be a good choice for backing up an entire `/home` directory with home directories for multiple users. Finally, since this creates one huge archive file, any type of corruption in the archive file could cause you to lose the entire backup.

6. End of Lab.

Using private and public keys for asymmetric encryption and signing

Symmetric encryption is great if you're just using GPG locally for your own stuff, but what if you want to share an encrypted file with someone, while ensuring that they can decrypt it? With symmetric encryption, you'd need to find a secure way to transmit the passphrase for the file to the file's recipient. In doing so, there will always be the risk that some third party could intercept the passphrase, and could then get into your stuff. Here's where asymmetric encryption comes to the rescue. To demonstrate, I'm going to create a file, encrypt it, and send it to my buddy Frank to decrypt.

Asymmetric encryption, is, well, asymmetric. Being asymmetric means that you would use one key to encrypt a file, and another key to decrypt it. You would keep your private key to yourself and guard it with your life, but you would share the public key with the whole world. The beauty of this is that you can share encrypted files with another person, and only that person would be able to decrypt them. This is all done without having to share a password with the recipient.

To begin, both Frank and I have to create a key set, as we've already shown you. Next, each of us needs to extract our public keys, and send them to each other. We'll extract the key into an ASCII text file:

```
cd .gnupg
gpg --export -a -o donnie_public-key.txt

donnie@ubuntu:~/.gnupg$ ls -l
total 36
-rw-rw-r-- 1 donnie donnie 1706 Oct 27 18:14 donnie_public-key.txt
```

```
. . .

frank@ubuntu:~/.gnupg$ ls -l
total 36
-rw-rw-r-- 1 frank frank 1714 Oct 27 18:18 frank_public-key.txt
```

Normally, the participants in this would either send their keys to each other through an email attachment, or by placing the keys in a shared directory. In this case, Frank and I will receive each other's public key files, and place them into our respective `.gnupg` directories. Once that's done, we're ready to import each other's keys:

```
donnie@ubuntu:~/.gnupg$ gpg --import frank_public-key.txt
gpg: key 4CFC6990: public key "Frank Siamese (I am a cat.) <frank@any.net>"
imported
gpg: Total number processed: 1
gpg: imported: 1 (RSA: 1)
donnie@ubuntu:~/.gnupg$

frank@ubuntu:~/.gnupg$ gpg --import donnie_public-key.txt
gpg: key 9FD7014B: public key "Donald A. Tevault <donniet@something.net>"
imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
frank@ubuntu:~/.gnupg$
```

Now for the good stuff. I've created a super-secret message for Frank, and will asymmetrically encrypt it (`-e`) and sign it (`-s`). (Signing the message is the verification that the message really is from me, rather than from an impostor.):

```
donnie@ubuntu:~$ gpg -s -e secret_stuff_for_frank.txt

You need a passphrase to unlock the secret key for
user: "Donald A. Tevault <donniet@something.net>"
2048-bit RSA key, ID 9FD7014B, created 2017-10-27

gpg: gpg-agent is not available in this session
You did not specify a user ID. (you may use "-r")

Current recipients:

Enter the user ID.  End with an empty line: frank
gpg: CD8104F7: There is no assurance this key belongs to the named user

pub  2048R/CD8104F7 2017-10-27 Frank Siamese (I am a cat.) <frank@any.net>
 Primary key fingerprint: 4806 7483 5442 D62B B9BD  95C1 9564 92D4 4CFC
6990
      Subkey fingerprint: 9DAB 7C3C 871D 6711 4632  A5E0 6DDD E3E5 CD81
```

```
04F7

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y

Current recipients:
2048R/CD8104F7 2017-10-27 "Frank Siamese (I am a cat.) <frank@any.net>"

Enter the user ID.  End with an empty line:
donnie@ubuntu:~$
```

So, the first thing I had to do was to enter the passphrase for my private key. Where it says to enter the user ID, I entered `frank`, since he's the intended recipient of my message. But, look at the line after that, where it says, `There is no assurance this key belongs to the named user`. That's because I still haven't *trusted* Frank's public key. We'll get to that in a bit. The last line of the output again says to enter a user ID, so that we can designate multiple recipients. But, Frank is the only one I care about right now, so I just hit the *Enter* key to break out of the routine. This results in a `.gpg` version of my message to Frank:

```
donnie@ubuntu:~$ ls -l
total 8
. . .
-rw-rw-r-- 1 donnie donnie 143 Oct 27 18:37 secret_stuff_for_frank.txt
-rw-rw-r-- 1 donnie donnie 790 Oct 27 18:39 secret_stuff_for_frank.txt.gpg
donnie@ubuntu:~$
```

My final step is to send Frank his encrypted message file, by whatever means available.

When Frank receives his message, he'll use the `-d` option to view it:

```
frank@ubuntu:~$ gpg -d secret_stuff_for_frank.txt.gpg

You need a passphrase to unlock the secret key for
user: "Frank Siamese (I am a cat.) <frank@any.net>"
2048-bit RSA key, ID CD8104F7, created 2017-10-27 (main key ID 4CFC6990)

gpg: gpg-agent is not available in this session
gpg: encrypted with 2048-bit RSA key, ID CD8104F7, created 2017-10-27
      "Frank Siamese (I am a cat.) <frank@any.net>"
This is TOP SECRET stuff that only Frank can see!!!!!
If anyone else see it, it's the end of the world as we know it.
(With apologies to REM.)
```

```
gpg: Signature made Fri 27 Oct 2017 06:39:15 PM EDT using RSA key ID
9FD7014B
gpg: Good signature from "Donald A. Tevault <donniet@something.net>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
owner.
Primary key fingerprint: DB0B 31B8 876D 9B2C 7F12  9FC3 886F 3357 9FD7 014B
frank@ubuntu:~$
```

Frank enters the passphrase for his private key, and he sees the message. At the bottom, he sees the warning about how my public key isn't trusted, and that there's `no indication that the signature belongs to the owner`. Well, since Frank knows me personally, and he knows for a fact that the public key really is mine, he can add my public key to the *trusted* list:

```
frank@ubuntu:~$ cd .gnupg
frank@ubuntu:~/.gnupg$ gpg --edit-key donnie
gpg (GnuPG) 1.4.20; Copyright (C) 2015 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.


gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 2u
pub  2048R/9FD7014B  created: 2017-10-27  expires: never       usage: SC
                     trust: ultimate      validity: ultimate
sub  2048R/9625E7E9  created: 2017-10-27  expires: never       usage: E
[ultimate] (1). Donald A. Tevault <donniet@something.net>

gpg>
```

The last line of this output is the command prompt for the `gpg` shell. Frank is concerned with trust, so he'll enter the command, `trust`:

```
gpg> trust
pub  2048R/9FD7014B  created: 2017-10-27  expires: never       usage: SC
                     trust: unknown       validity: unknown
sub  2048R/9625E7E9  created: 2017-10-27  expires: never       usage: E
[ unknown] (1). Donald A. Tevault <donniet@something.net>

Please decide how far you trust this user to correctly verify other users'
keys
(by looking at passports, checking fingerprints from different sources,
etc.)

  1 = I don't know or won't say
```

```
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y
```

Frank has known me for quite a while, and he knows for a fact that I'm the one who sent the key. So, he chooses option `5` for ultimate trust. Once Frank logs out and logs back in, that trust will take effect:

```
frank@ubuntu:~$ gpg -d secret_stuff_for_frank.txt.gpg

You need a passphrase to unlock the secret key for
user: "Frank Siamese (I am a cat.) <frank@any.net>"
2048-bit RSA key, ID CD8104F7, created 2017-10-27 (main key ID 4CFC6990)

gpg: gpg-agent is not available in this session
gpg: encrypted with 2048-bit RSA key, ID CD8104F7, created 2017-10-27
      "Frank Siamese (I am a cat.) <frank@any.net>"
This is TOP SECRET stuff that only Frank can see!!!!!
If anyone else see it, it's the end of the world as we know it.
(With apologies to REM.)
gpg: Signature made Fri 27 Oct 2017 06:39:15 PM EDT using RSA key ID
9FD7014B
gpg: Good signature from "Donald A. Tevault <donniet@something.net>"
frank@ubuntu:~$
```

With no more warning messages, this looks much better. At my end, I'll do the same thing with Frank's public key.

What's so very cool about this is that even though the whole world may have my public key, it's useless to anyone who isn't a designated recipient of my message.

On an Ubuntu machine, to get rid of the `gpg-agent is not available in this session` messages, and to be able to cache your passphrase in the keyring, install the `gnupg-agent` package:

```
sudo apt install gnupg-agent
```

Signing a file without encryption

If a file isn't secret, but you still need to ensure authenticity and integrity, you can just sign it without encrypting it:

```
donnie@ubuntu:~$ gpg -s not_secret_for_frank.txt

You need a passphrase to unlock the secret key for
user: "Donald A. Tevault <donniet@something.net>"
2048-bit RSA key, ID 9FD7014B, created 2017-10-27

gpg: gpg-agent is not available in this session
donnie@ubuntu:~$ ls -l
. . .
-rw-rw-r-- 1 donnie donnie  40 Oct 27 19:30 not_secret_for_frank.txt
-rw-rw-r-- 1 donnie donnie 381 Oct 27 19:31 not_secret_for_frank.txt.gpg
```

Just as before, I create a `.gpg` version of the file. When Frank receives the file, he may try to open it with `less`:

```
frank@ubuntu:~$ less not_secret_for_frank.txt.gpg
"not_secret_for_frank.txt.gpg" may be a binary file.  See it anyway?

<A3>^A^Av^A<89><FE><90>^M^C^@^B^A<88>o3W<9F><D7>^AK^A<AC>Fb^Xnot_secret_for
_frank.txtY<F3><C1><C0>This isn't secret, so I just signed it.
<89>^A^\^D^@^A^B^@^F^E^BY<F3><C1><C0>^@
^P<88>o3W<9F><D7>^AK6<AF>^G<FF>Bs<9A>^Lc^@<E9><ED><C2>-2<AE><A7><DF>ąB
<EC>/[:<D1>{<B2><FD>o8<C6><C9>x<FE>*4^D<CD>^G^O^F<F3>@v<87>_1<D0>^Bp<FE>q^N
3<B0><BE><85><D2>9]{<D6><EF><9A><D8>  `<C2><E4>^NC<9B>
"Ƣ^?M<89>s<9F>z^B"<DF>
s}`<A4>:<B4>&<F7><F4>\EjŌ!^Q
<9C>6^E|H<E2>ESC<D9>9<DC>p_ӟESCB<DE>^P<FF>i<CA>)^O
<A0>
<CB><C4>+<81><F5><A7>`5<90><BF>Y<DE><FF><<A0>z<BC><BD>5<C5><E8><FE>>
<B7><A2>^L^_^D<DD>Kk<E0><9A>8<C6>S^E<D0>fjz<B2>&G<A4><A8>^Lg$8Q>{<FF><FA>^M
_A
<A1><93><C3>4<DC><C4>x<86><D9>^]- <8A>
F0<87><8A><94>%A<96><DF><CD>C><80><C3>l
<D3>K<E5>^G<8E><90>d<8C><DA>Aŋb<86><89><DA>S<B6><91><D8><D2><E0><B3>K<FC><9
E>
<ED>^@*<EF>x<E7>jø<FD><D3><FA><9A>^]
not_secret_for_frank.txt.gpg (END)
```

There's a lot of gibberish there because of the signature, but if you look carefully, you'll see the plain, unencrypted message. Frank will use `gpg` with the `--verify` option to verify that the signature really does belong to me:

```
frank@ubuntu:~$ gpg --verify not_secret_for_frank.txt.gpg
gpg: Signature made Fri 27 Oct 2017 07:31:12 PM EDT using RSA key ID
9FD7014B
gpg: Good signature from "Donald A. Tevault <donniet@something.net>"
frank@ubuntu:~$
```

Encrypting partitions with Linux Unified Key Setup – LUKS

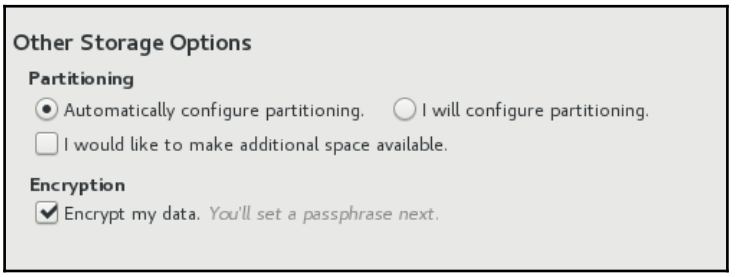
Being able to encrypt individual files can be handy, but it can be quite unwieldy for a large number of files. For that, we need something better, and we have three different methods:

- **Block encryption**: We can use this for either whole-disk encryption, or to encrypt individual partitions
- **File-level encryption**: We'd use this to encrypt individual directories, without having to encrypt the underlying partitions
- **Containerized Encryption**: Using third-party software that doesn't come with any Linux distribution, we can create encrypted, cross-platform containers that can be opened on either Linux, Mac, or Windows machines

The **Linux Unified Key Setup** (**LUKS**), falls into the first category. It's built into pretty much every Linux distribution, and directions for use are the same for each. For our demos, I'll use the CentOS virtual machine, since LUKS is now the default encryption mechanism for Red Hat Enterprise Linux 7 and CentOS 7.

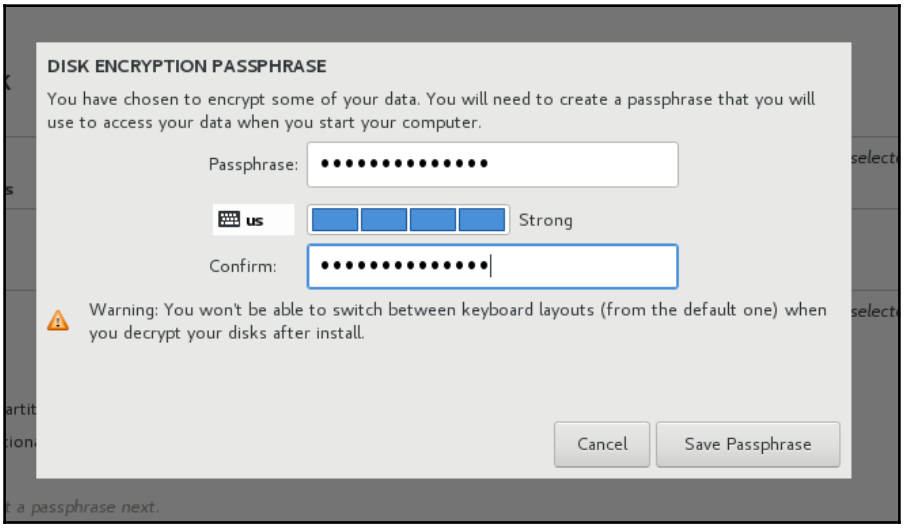
Disk encryption during operating system installation

When you install Red Hat Enterprise Linux 7 or one of its offspring, you have the option of encrypting the drive. All you have to do is to click on a checkbox:



Other than that, I just let the installer create the default partitioning scheme, which means that the `/` filesystem and the `swap` partition will both be logical volumes. (I'll cover that in a moment.)

Before the installation can continue, I have to create a passphrase to mount the encrypted disk:



Now, whenever I reboot the system, I need to enter this passphrase:

```
Please enter passphrase for disk VBOX_HARDDISK (luks-2d7f02c7-864f-42ce-b362-50d
d830d9772)!:_
```

Once the machine is up and running, I can look at the list of logical volumes. I see both the `/` logical volume and the `swap` logical volume:

```
[donnie@localhost etc]$ sudo lvdisplay
  --- Logical volume ---
  LV Path /dev/centos/swap
  LV Name swap
  VG Name centos
  LV UUID tsme2v-uy87-uech-vpNp-W4E7-fHLf-3bf817
  LV Write Access read/write
  LV Creation host, time localhost, 2017-10-28 13:00:11 -0400
  LV Status available
  # open 2
  LV Size 2.00 GiB
  Current LE 512
  Segments 1
  Allocation inherit
  Read ahead sectors auto
  - currently set to 8192
  Block device 253:2

  --- Logical volume ---
  LV Path /dev/centos/root
  LV Name root
  VG Name centos
  LV UUID MKXVO9-X8fo-w2FC-LnGO-GLnq-k2Xs-xI1gn0
  LV Write Access read/write
  LV Creation host, time localhost, 2017-10-28 13:00:12 -0400
  LV Status available
  # open 1
  LV Size 17.06 GiB
  Current LE 4368
  Segments 1
  Allocation inherit
  Read ahead sectors auto
  - currently set to 8192
  Block device 253:1

[donnie@localhost etc]$
```

And I can look at the list of physical volumes. (Actually, there's only one physical volume in the list, and it's listed as a `luks` physical volume.):

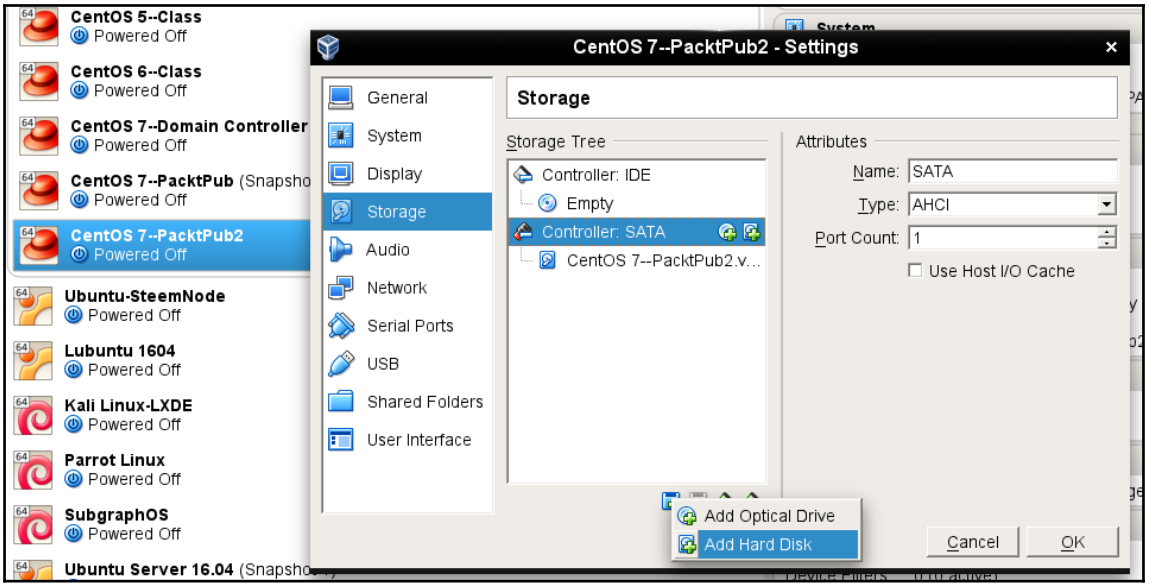
```
[donnie@localhost etc]$ sudo pvdisplay
  --- Physical volume ---
  PV Name               /dev/mapper/luks-2d7f02c7-864f-42ce-
b362-50dd830d9772
  VG Name               centos
  PV Size               <19.07 GiB / not usable 0
  Allocatable           yes
  PE Size               4.00 MiB
  Total PE              4881
  Free PE               1
  Allocated PE          4880
  PV UUID               V50E4d-jOCU-kVRn-67w9-5zwR-nbwg-4P725S

[donnie@localhost etc]$
```

This shows that the underlying physical volume is encrypted, which means that both the `/` and the `swap` logical volumes are also encrypted. That's a good thing, because leaving the swap space unencrypted—a common mistake when setting up disk encryption up manually—can lead to data leakage.

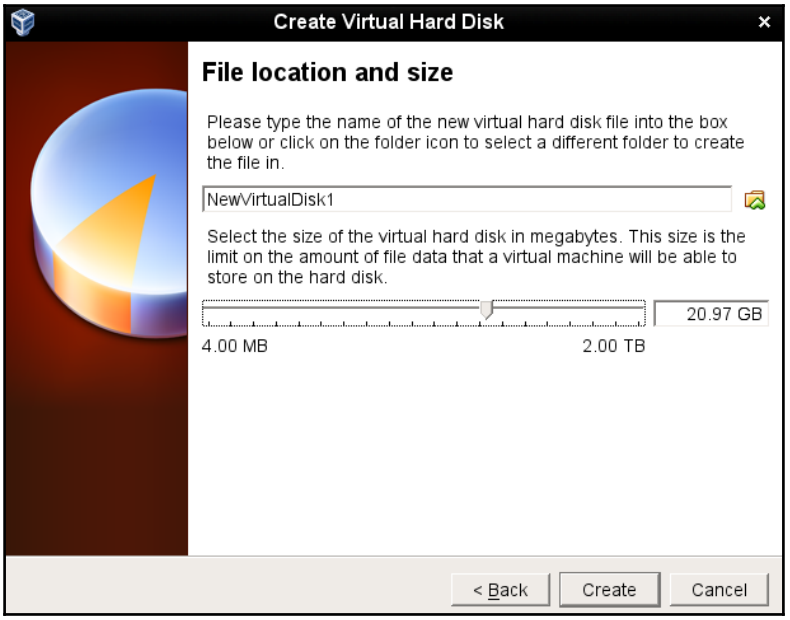
Adding an encrypted partition with LUKS

There may be times when you'll need to either add another encrypted drive to an existing machine, or encrypt a portable device, such as a USB memory stick. This procedure works for both scenarios.

To demonstrate, I'll shut down my CentOS VM and add another virtual drive:



I'll bump the drive capacity up to 20 GB, which will give me plenty of room to play with:



After rebooting the machine, I now have a `/dev/sdb` drive to play with. My next step is to create a partition. It doesn't matter whether I create a new-fangled GPT partition, or an old-fashioned MBR partition. I'll create a GPT partition, and my preferred utility for doing that is `gdisk`, simply because it's so similar to the old `fdisk` that I know and love so well. The only catch is that `gdisk` isn't installed on CentOS by default:

```
sudo yum install gdisk
sudo gdisk /dev/sdb
```

I'll use the entire drive for my partition, and leave the partition type set at the default `8300`. I now have the `/dev/sdb1` partition:

```
[donnie@localhost ~]$ sudo gdisk -l /dev/sdb
[sudo] password for donnie:
GPT fdisk (gdisk) version 0.8.6

Partition table scan:
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present

Found valid GPT with protective MBR; using GPT.
Disk /dev/sdb: 43978112 sectors, 21.0 GiB
Logical sector size: 512 bytes
Disk identifier (GUID): DC057EC6-3BA8-4269-ABE9-2A28B4FDC84F
Partition table holds up to 128 entries
First usable sector is 34, last usable sector is 43978078
Partitions will be aligned on 2048-sector boundaries
Total free space is 2014 sectors (1007.0 KiB)

Number Start (sector) End (sector) Size Code Name
   1 2048 43978078 21.0 GiB 8300 Linux filesystem
[donnie@localhost ~]$
```

I'll next use `cryptsetup` to convert the partition to LUKS format. In this command, the `-v` signifies verbose mode, and the `-y` signifies that I'll have to enter my passphrase twice in order to properly verify it. Note that when it says to type `yes` all in uppercase, it really does mean to type it in uppercase:

```
[donnie@localhost ~]$ sudo cryptsetup -v -y luksFormat /dev/sdb1

WARNING!
========
This will overwrite data on /dev/sdb1 irrevocably.
```

```
Are you sure? (Type uppercase yes): YES
Enter passphrase:
Verify passphrase:
Command successful.
[donnie@localhost ~]$
```

Although I don't have to, I'd like to look at the information about my new encrypted partition:

```
[donnie@localhost ~]$ sudo cryptsetup luksDump /dev/sdb1
LUKS header information for /dev/sdb1

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
. . .
. . .
```

There's a lot more to the output than what I can show here, but you get the idea.

Next, I'll map the partition to a device name. You can name the device pretty much whatever you want, and I'll just name mine `secrets`. (I know, it's a corny name. You probably won't want to make it so obvious where you're storing your secrets.):

```
[donnie@localhost ~]$ sudo cryptsetup luksOpen /dev/sdb1 secrets
Enter passphrase for /dev/sdb1:
[donnie@localhost ~]$
```

When I look in the `/dev/mapper` directory, I see my new `secrets` device, listed as a symbolic link to the `dm-3` device:

```
[donnie@localhost mapper]$ pwd
/dev/mapper
[donnie@localhost mapper]$ ls -l se*
lrwxrwxrwx. 1 root root 7 Oct 28 17:39 secrets -> ../dm-3
[donnie@localhost mapper]$
```

I'll use `dmsetup` to look at the information about my new device:

```
[donnie@localhost mapper]$ sudo dmsetup info secrets
[sudo] password for donnie:
Name:              secrets
State:             ACTIVE
Read Ahead:        8192
Tables present:    LIVE
Open count:        0
```

```
Event number:      0
Major, minor:      253, 3
Number of targets: 1
UUID: CRYPT-LUKS1-6cbdce1748d441a18f8e793c0fa7c389-secrets

[donnie@localhost mapper]$
```

The next step is to format the partition in the usual manner. I could use any filesystem that's supported by Red Hat and CentOS. But, since everything else on my system is already formatted with XFS, that's what I'll go with here, as well:

```
[donnie@localhost ~]$ sudo mkfs.xfs /dev/mapper/secrets
meta-data=/dev/mapper/secrets    isize=512    agcount=4, agsize=1374123
blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=1        finobt=0, sparse=0
data     =                       bsize=4096   blocks=5496491, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=1
log      =internal log           bsize=4096   blocks=2683, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
[donnie@localhost ~]$
```

My final step is to create a mount point and to mount the encrypted partition:

```
[donnie@localhost ~]$ sudo mkdir /secrets
[sudo] password for donnie:
[donnie@localhost ~]$ sudo mount /dev/mapper/secrets /secrets
[donnie@localhost ~]$
```

The `mount` command will verify that the partition is mounted properly:

```
[donnie@localhost ~]$ mount | grep 'secrets'
/dev/mapper/secrets on /secrets type xfs
(rw,relatime,seclabel,attr2,inode64,noquota)
[donnie@localhost ~]$
```

Configuring the LUKS partition to mount automatically

The only missing piece of the puzzle is to configure the system to automatically mount the LUKS partition upon boot-up. To do that, I'll configure two different files:

- `/etc/crypttab`
- `/etc/fstab`

Had I not chosen to encrypt the disk when I installed the operating system, I wouldn't have a `crypttab` file, and I would have to create it myself. But, since I did choose to encrypt the drive, I already have one with information about that drive:

```
luks-2d7f02c7-864f-42ce-b362-50dd830d9772 UUID=2d7f02c7-864f-42ce-
b362-50dd830d9772 none
```

The first two fields describe the name and location of the encrypted partition. The third field is for the encryption passphrase. If it's set to `none`, as it is here, then the passphrase will have to be manually entered upon boot-up.

In the `fstab` file, we have the entry that actually mounts the partition:

```
/dev/mapper/centos-root /                   xfs     defaults,x-
systemd.device-timeout=0 0 0
UUID=9f9fbf9c-d046-44fc-a73e-ca854d0ca718 /boot                   xfs
defaults        0 0
/dev/mapper/centos-swap swap                swap    defaults,x-
systemd.device-timeout=0 0 0
```

Well, there are actually two entries in this case, because I have two logical volumes, `/` and `swap`, on top of my encrypted physical volume. The `UUID` line is the `/boot` partition, which is the only part of the drive that isn't encrypted. Now, let's add our new encrypted partition so that it will mount automatically, as well.

This is where it would be extremely helpful to remotely log into your virtual machine from your desktop host machine. By using a GUI-type Terminal, whether it be the Terminal from a Linux or MacOS machine, or Cygwin from a Windows machine, you'll have the ability to perform copy-and-paste operations, which you won't have if you work directly from the virtual machine terminal. (Trust me, you don't want to be typing in those long UUIDs.)

The first step is to obtain the UUID of the encrypted partition:

```
[donnie@localhost etc]$ sudo cryptsetup luksUUID /dev/sdb1
[sudo] password for donnie:
6cbdce17-48d4-41a1-8f8e-793c0fa7c389
[donnie@localhost etc]$
```

I'll copy that UUID, and paste it into the `/etc/crypttab` file. (Note that you'll paste it in twice. The first time, you'll prepend it with `luks-`, and the second time you'll append it with `UUID=`.):

```
luks-2d7f02c7-864f-42ce-b362-50dd830d9772 UUID=2d7f02c7-864f-42ce-
b362-50dd830d9772 none
luks-6cbdce17-48d4-41a1-8f8e-793c0fa7c389
UUID=6cbdce17-48d4-41a1-8f8e-793c0fa7c389 none
```

Finally, I'll edit the `/etc/fstab` file, adding the last line in the file for my new encrypted partition. (Note that I again used `luks-`, followed by the UUID number.):

```
/dev/mapper/centos-root / xfs defaults,x-systemd.device-timeout=0 0 0
UUID=9f9fbf9c-d046-44fc-a73e-ca854d0ca718 /boot xfs defaults 0 0
/dev/mapper/centos-swap swap swap defaults,x-systemd.device-timeout=0 0 0
/dev/mapper/luks-6cbdce17-48d4-41a1-8f8e-793c0fa7c389 /secrets xfs defaults
0 0
```

When editing the `fstab` file for adding normal, unencrypted partitions, I always like to do a `sudo mount -a` to check the `fstab` file for typos. That won't work with LUKS partitions though, because `mount` won't recognize the partition until the system reads in the `crypttab` file, and that won't happen until I reboot the machine. So, just be extra careful with editing `fstab` when adding LUKS partitions.

Now for the moment of truth. I'll reboot the machine to see if everything works.

Okay, the machine has rebooted, and `mount` shows that my endeavors have been successful:

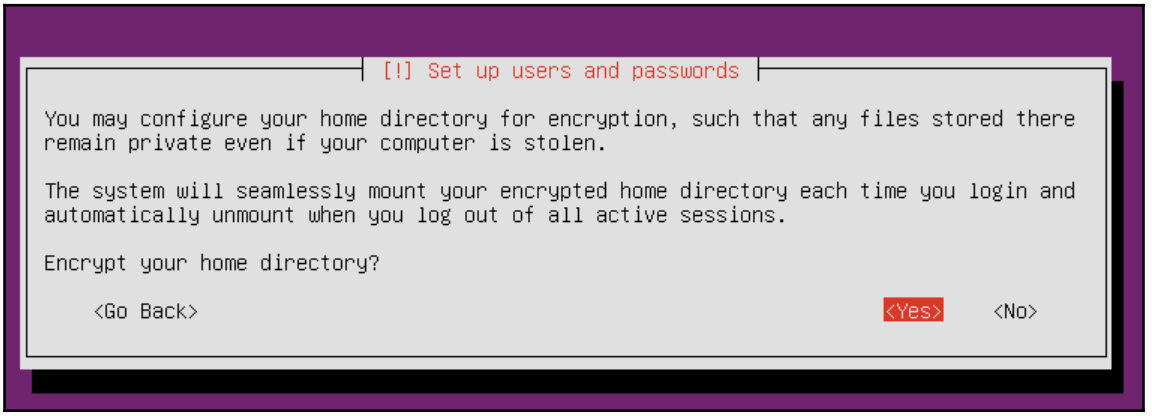
```
[donnie@localhost ~]$ mount | grep 'secrets'
/dev/mapper/luks-6cbdce17-48d4-41a1-8f8e-793c0fa7c389 on /secrets type xfs
(rw,relatime,seclabel,attr2,inode64,noquota)
[donnie@localhost ~]$
```

Encrypting directories with eCryptfs

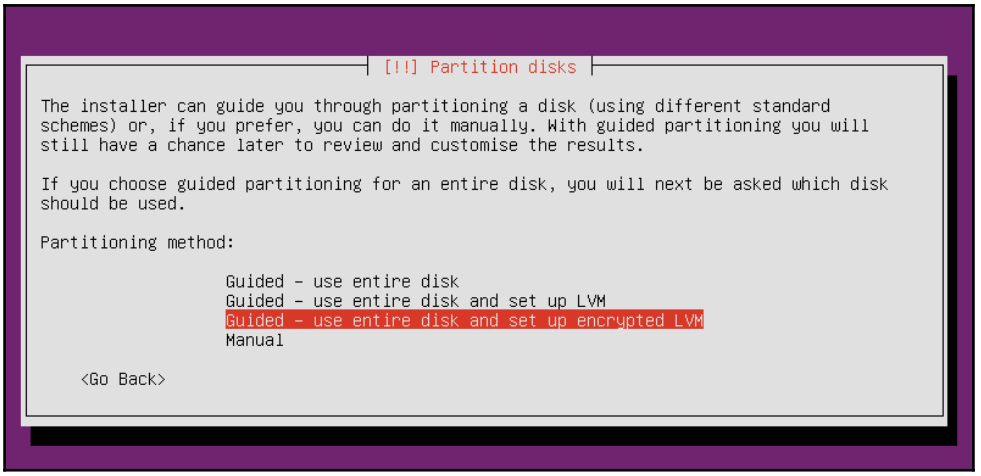
Encrypting entire partitions is cool, but you might, at times, just need to encrypt an individual directory. For that, we can use eCryptfs. We'll need to use our Ubuntu machines for this, because Red Hat and CentOS no longer include eCryptfs in version 7 of their products. (It was in Red Hat 6 and CentOS 6, but it's no longer even available for installation in version 7.)

Home directory and disk encryption during Ubuntu installation

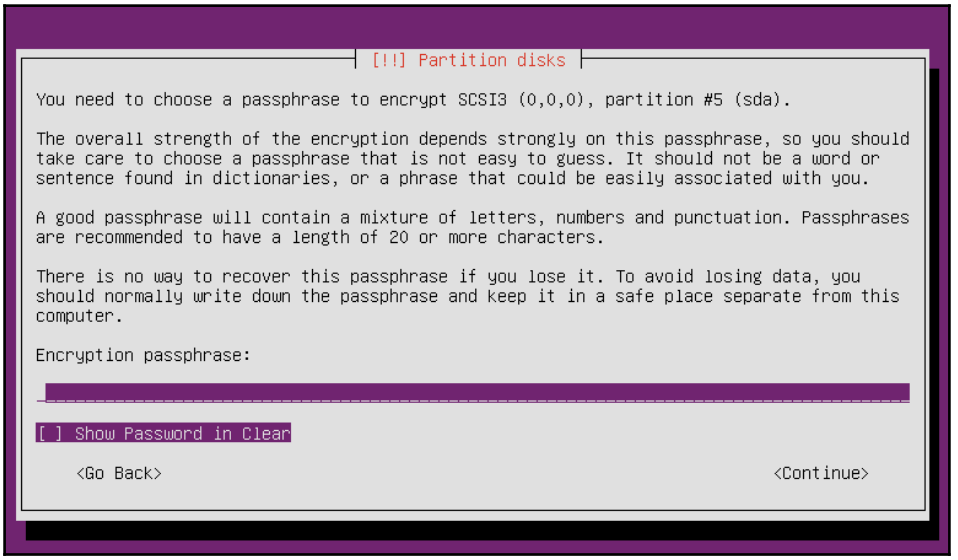
When you install Ubuntu Server, you have two chances to implement encryption. You'll first be given the chance to encrypt your home directory:



Later, on the **Partition disks** screen, you'll be given the chance to set up encrypted logical volumes for whole disk encryption:



After choosing this option, you will then be asked to enter a passphrase:



The disk encryption uses LUKS, just the same as we saw on the CentOS machine. To prove this, all we have to do is look for a populated `crypttab` file in the `/etc` directory:

```
donnie@ubuntu3:~$ cd /etc
donnie@ubuntu3:/etc$ cat crypttab
sda5_crypt UUID=56190c2b-e46b-40a9-af3c-4cb26c4fe998 none luks,discard
cryptswap1 UUID=60661042-0dbd-4c2a-9cf9-7f02a73864ae /dev/urandom
swap,offset=1024,cipher=aes-xts-plain64
donnie@ubuntu3:/etc$
```

Unlike Red Hat and CentOS, an Ubuntu machine will always have the `/etc/crypttab` file, even if there are no LUKS partitions. Without LUKS partitions, the file will be empty.

The home directory encryption uses eCryptfs, as evidenced by the `.ecryptfs` directory in the `/home` directory:

```
donnie@ubuntu3:/home$ ls -la
total 16
drwxr-xr-x 4 root root 4096 Oct 29 15:06 .
drwxr-xr-x 23 root root 4096 Oct 29 15:23 ..
drwx------ 3 donnie donnie 4096 Oct 29 15:29 donnie
drwxr-xr-x 3 root root 4096 Oct 29 15:06 .ecryptfs
donnie@ubuntu3:/home$
```

So, what we have here is encryption on top of encryption, for double protection. Is that really necessary? Probably not, but choosing to encrypt my home directory ensured that the access permissions for it got set to the more restrictive `700` setting, rather than the default `755` setting. Be aware though, that any user accounts you create now will have wide open permissions settings on their home directories. Unless, that is, we create user accounts with the encryption option.

Encrypting a home directory for a new user account

In `Chapter 2`, *Securing User Accounts*, I showed you how Ubuntu allows you to encrypt a user's home directory as you create his or her user account. To review, let's see the command for creating Goldie's account:

```
sudo adduser --encrypt-home goldie
```

When Goldie logs in, the first thing she'll want to do is to `unwrap` her mount passphrase, write it down, and store it in a secure place. (She'll need this if she ever needs to recover a corrupted directory.):

```
ecryptfs-unwrap-passphrase .ecryptfs/wrapped-passphrase
```

When you use `adduser --encrypt-home`, home directories for new users will automatically be set to a restrictive permissions value that will keep everyone out except for the owner of the directory. This happens even when you leave the `adduser.conf` file set with its default settings.

Creating a private directory within an existing home directory

Let's say that you have users who, for whatever strange reason, don't want to encrypt their entire home directories, and want to keep the `755` permissions settings on their home directories so that other people can access their files. But, they also want a private directory that nobody but them can access.

Instead of encrypting an entire home directory, any user can create an encrypted private directory within his or her own home directory. The first step, if it hasn't already been done, is for someone with admin privileges to install the `ecryptfs-utils` package:

```
sudo apt install ecryptfs-utils
```

To create this private directory, we'll use the interactive `ecryptfs-setup-private` utility. If you have admin privileges, you can do this for other users. Users without admin privileges can do it for themselves. For our demo, let's say that Charlie, my big Siamese/Gray tabby guy, needs his own encrypted private space. (Who knew that cats had secrets, right?):

```
charlie@ubuntu2:~$ ecryptfs-setup-private
Enter your login passphrase [charlie]:
Enter your mount passphrase [leave blank to generate one]:
Enter your mount passphrase (again):

**********************************************************************
YOU SHOULD RECORD YOUR MOUNT PASSPHRASE AND STORE IT IN A SAFE LOCATION.
  ecryptfs-unwrap-passphrase ~/.ecryptfs/wrapped-passphrase
THIS WILL BE REQUIRED IF YOU NEED TO RECOVER YOUR DATA AT A LATER TIME.
**********************************************************************
```

```
Done configuring.

Testing mount/write/umount/read...
Inserted auth tok with sig [e339e1ebf3d58c36] into the user session keyring
Inserted auth tok with sig [7a40a176ac647bf0] into the user session keyring
Inserted auth tok with sig [e339e1ebf3d58c36] into the user session keyring
Inserted auth tok with sig [7a40a176ac647bf0] into the user session keyring
Testing succeeded.

Logout, and log back in to begin using your encrypted directory.

charlie@ubuntu2:~$
```

For the `login` passphrase, Charlie enters his normal password or passphrase for logging into his user account. He could have let the system generate its own `mount` passphrase, but he decided to enter his own. Since he did enter his own mount passphrase, he didn't need to do the `ecryptfs-unwrap-passphrase` command to find out what the passphrase is. But, just to show how that command works, let's say that Charlie entered `TurkeyLips` as his mount passphrase:

```
charlie@ubuntu2:~$ ecryptfs-unwrap-passphrase .ecryptfs/wrapped-passphrase
Passphrase:
TurkeyLips
charlie@ubuntu2:~$
```

Yes, it's a horribly weak passphrase, but for our demo purposes, it works.

After Charlie logs out and logs back in, he can start using his new private directory. Also, you can see that he has three new hidden directories within his home directory. All three of these new directories are only accessible by Charlie, even though his top-level home directory is still wide open to everybody:

```
charlie@ubuntu2:~$ ls -la
total 40
drwxr-xr-x 6 charlie charlie 4096 Oct 30 17:00 .
drwxr-xr-x 4 root root 4096 Oct 30 16:38 ..
-rw------- 1 charlie charlie 270 Oct 30 17:00 .bash_history
-rw-r--r-- 1 charlie charlie 220 Aug 31 2015 .bash_logout
-rw-r--r-- 1 charlie charlie 3771 Aug 31 2015 .bashrc
drwx------ 2 charlie charlie 4096 Oct 30 16:39 .cache
drwx------ 2 charlie charlie 4096 Oct 30 16:57 .ecryptfs
drwx------ 2 charlie charlie 4096 Oct 30 16:57 Private
drwx------ 2 charlie charlie 4096 Oct 30 16:57 .Private
-rw-r--r-- 1 charlie charlie 655 May 16 08:49 .profile
charlie@ubuntu2:~$
```

If you do a `grep 'ecryptfs' *` command in the `/etc/pam.d` directory, you'll see that PAM is configured to automatically mount users' encrypted directories whenever they log into the system:

```
donnie@ubuntu2:/etc/pam.d$ grep 'ecryptfs' *
common-auth:auth    optional    pam_ecryptfs.so unwrap
common-password:password    optional    pam_ecryptfs.so
common-session:session    optional    pam_ecryptfs.so unwrap
common-session-noninteractive:session    optional    pam_ecryptfs.so unwrap
donnie@ubuntu2:/etc/pam.d$
```

Encrypting other directories with eCryptfs

Encrypting other directories is a simple matter of mounting them with the `ecryptfs` filesystem. For our example, let's create a `secrets` directory in the top level of our filesystem, and encrypt it. Note how you list the directory name twice, because you also need to specify a mount point. (Essentially, you're using the directory that you're mounting as its own mount point.)

```
sudo mkdir /secrets
sudo mount -t ecryptfs /secrets /secrets
```

The output from this command is a bit lengthy, so let's break it down.

First, you'll enter your desired passphrase, and choose the encryption algorithm and the key length:

```
donnie@ubuntu2:~$ sudo mount -t ecryptfs /secrets /secrets
[sudo] password for donnie:
Passphrase:
Select cipher:
 1) aes: blocksize = 16; min keysize = 16; max keysize = 32
 2) blowfish: blocksize = 8; min keysize = 16; max keysize = 56
 3) des3_ede: blocksize = 8; min keysize = 24; max keysize = 24
 4) twofish: blocksize = 16; min keysize = 16; max keysize = 32
 5) cast6: blocksize = 16; min keysize = 16; max keysize = 32
 6) cast5: blocksize = 8; min keysize = 5; max keysize = 16
Selection [aes]:
Select key bytes:
 1) 16
 2) 32
 3) 24
Selection [16]:
```

We'll go with the default of `aes`, and 16 bytes for the key.

I'm going to go with the default of no for `plaintext passthrough`, and with yes for filename encryption:

```
Enable plaintext passthrough (y/n) [n]:
Enable filename encryption (y/n) [n]: y
```

I'll go with the default `Filename Encryption Key`, and verify the mounting options:

```
Filename Encryption Key (FNEK) Signature [e339e1ebf3d58c36]:
Attempting to mount with the following options:
  ecryptfs_unlink_sigs
  ecryptfs_fnek_sig=e339e1ebf3d58c36
  ecryptfs_key_bytes=16
  ecryptfs_cipher=aes
  ecryptfs_sig=e339e1ebf3d58c36
```

This warning only comes up when you mount the directory for the first time. For the final two questions, I'll type `yes` in order to prevent that warning from coming up again:

```
WARNING: Based on the contents of [/root/.ecryptfs/sig-cache.txt],
it looks like you have never mounted with this key
before. This could mean that you have typed your
passphrase wrong.

Would you like to proceed with the mount (yes/no)? : yes
Would you like to append sig [e339e1ebf3d58c36] to
[/root/.ecryptfs/sig-cache.txt]
in order to avoid this warning in the future (yes/no)? : yes
Successfully appended new sig to user sig cache file
Mounted eCryptfs
donnie@ubuntu2:~$
```

Just for fun, I'll create a file within my new encrypted `secrets` directory, and then unmount the directory:

```
cd /secrets
sudo vim secret_stuff.txt
cd
sudo umount /secrets
ls -l /secrets

donnie@ubuntu2:/secrets$ ls -l
total 12
-rw-r--r-- 1 root root 12288 Oct 31 18:24
ECRYPTFS_FNEK_ENCRYPTED.FXbXCS5fwxKABUQtEPlumGPaN-RGvqd13yybkpTr1eCVWVHdr-
```

```
lrmi1X9Vu-mLM-A-VeqIdN6KNZGcs-
donnie@ubuntu2:/secrets$
```

By choosing to encrypt filenames, nobody can even tell what files you have when the directory is unmounted. When I'm ready to access my encrypted files again, I'll just remount the directory the same as I did before.

Encrypting the swap partition with eCryptfs

If you're just encrypting individual directories with eCryptfs instead of using LUKS whole-disk encryption, you'll need to encrypt your swap partition in order to prevent accidental data leakage. Fixing that problem requires just one simple command:

```
donnie@ubuntu:~$ sudo ecryptfs-setup-swap
[sudo] password for donnie:

WARNING:
An encrypted swap is required to help ensure that encrypted files are not
leaked to disk in an unencrypted format.

HOWEVER, THE SWAP ENCRYPTION CONFIGURATION PRODUCED BY THIS PROGRAM WILL
BREAK HIBERNATE/RESUME ON THIS SYSTEM!

NOTE: Your suspend/resume capabilities will not be affected.

Do you want to proceed with encrypting your swap? [y/N]: y

INFO: Setting up swap: [/dev/sda5]
WARNING: Commented out your unencrypted swap from /etc/fstab
swapon: stat of /dev/mapper/cryptswap1 failed: No such file or directory
donnie@ubuntu:~$
```

Don't mind the warning about the missing `/dev/mapper/cryptswap1` file. It will get created the next time you reboot the machine.

Using VeraCrypt for cross-platform sharing of encrypted containers

Once upon a time, there was TrueCrypt, a cross-platform program that allowed the sharing of encrypted containers across different operating systems. But the project was always shrouded in mystery, because its developers would never reveal their identities. And then, right out of the blue, the developers released a cryptic message about how TrueCrypt was no longer secure, and shut down the project.

VeraCrypt is the successor to TrueCrypt, and it allows the sharing of encrypted containers across Linux, Windows, MacOS, and FreeBSD machines. Although LUKS and eCryptfs are good, VeraCrypt does offer more flexibility in certain ways:

- As mentioned, VeraCrypt offers cross-platform sharing, whereas LUKS and eCryptfs don't
- VeraCrypt allows you to encrypt either whole partitions or whole storage devices, or to create virtual encrypted disks
- Not only can you create encrypted volumes with VeraCrypt, you can also hide them, giving you plausible deniability
- VeraCrypt comes in both command-line and GUI variants, so it's appropriate for either server use or for the casual desktop user
- Like LUKS and eCryptfs, VeraCrypt is free open source software, which means that it's free to use, and that the source code can be audited for either bugs or backdoors

Getting and installing VeraCrypt

The Linux version of VeraCrypt comes as a set of universal installer scripts that should work on any Linux distribution. Once you extract the `.tar.bz2` archive file, you'll see two scripts for GUI installation, and two for console-mode installation. One of each of those is for 32-bit Linux, and one of each is for 64-bit Linux:

```
donnie@linux-0ro8:~/Downloads> ls -l vera*
-r-xr-xr-x 1 donnie users  2976573 Jul  9 05:10 veracrypt-1.21-setup-
console-x64
-r-xr-xr-x 1 donnie users  2967950 Jul  9 05:14 veracrypt-1.21-setup-
console-x86
-r-xr-xr-x 1 donnie users  4383555 Jul  9 05:08 veracrypt-1.21-setup-gui-
x64
-r-xr-xr-x 1 donnie users  4243305 Jul  9 05:13 veracrypt-1.21-setup-gui-
```

```
x86
-rw-r--r-- 1 donnie users 14614830 Oct 31 23:49 veracrypt-1.21-
setup.tar.bz2
donnie@linux-0ro8:~/Downloads>
```

For the server demo, I used `scp` to transfer the 64-bit console-mode installer to one of my Ubuntu virtual machines. The executable permission is already set, so all you have to do to install is:

```
donnie@ubuntu:~$ ./veracrypt-1.21-setup-console-x64
```

You'll need sudo privileges, but the installer will prompt you for your sudo password. After reading and agreeing to a rather lengthy license agreement, the installation only takes a few seconds.

Creating and mounting a VeraCrypt volume in console mode

I haven't been able to find any documentation for the console-mode variant of VeraCrypt, but you can see a list of the available commands just by typing `veracrypt`. For this demo, I'm creating a 2 GB encrypted volume in my own home directory. But you can just as easily do it elsewhere, such as on a USB memory stick.

To create a new encrypted volume, type:

```
veracrypt -c
```

This will take you into an easy-to-use, interactive utility. For the most part, you'll be fine just accepting the default options:

```
donnie@ubuntu:~$ veracrypt -c
Volume type:
 1) Normal
 2) Hidden
Select [1]:

Enter volume path: /home/donnie/good_stuff

Enter volume size (sizeK/size[M]/sizeG): 2G

Encryption Algorithm:
 1) AES
 2) Serpent
 3) Twofish
```

```
 4) Camellia
 5) Kuznyechik
 6) AES(Twofish)
 7) AES(Twofish(Serpent))
 8) Serpent(AES)
 9) Serpent(Twofish(AES))
 10) Twofish(Serpent)
Select [1]:

Hash algorithm:
 1) SHA-512
 2) Whirlpool
 3) SHA-256
 4) Streebog
Select [1]:
. . .
. . .
```

For the filesystem, the default option of FAT will give you the best cross-platform compatibility between Linux, MacOS, and Windows:

```
Filesystem:
 1) None
 2) FAT
 3) Linux Ext2
 4) Linux Ext3
 5) Linux Ext4
 6) NTFS
 7) exFAT
Select [2]:
```

You'll then select your password and a PIM, which stands for Personal Iterations Multiplier. (For the PIM, I entered `8891`. High PIM values give better security, but they will also cause the volume to take longer to mount.) Then, type at least 320 random characters in order to generate the encryption key. (This is where it would be handy to have my cats walking across my keyboard.):

```
Enter password:
Re-enter password:

Enter PIM: 8891

Enter keyfile path [none]:

Please type at least 320 randomly chosen characters and then press Enter:
```

After you hit *Enter*, be patient, because the final generation of your encrypted volume will take a few moments. Here, you see that my 2 GB `good_stuff` container has been successfully created:

```
donnie@ubuntu:~$ ls -l good_stuff
-rw------- 1 donnie donnie 2147483648 Nov  1 17:02 good_stuff
donnie@ubuntu:~$
```

To use this container, I have to mount it. I'll begin by creating a mount point directory; the same as I would for mounting normal partitions:

```
donnie@ubuntu:~$ mkdir good_stuff_dir
donnie@ubuntu:~$
```

Use the `veracrypt` utility to mount your container on this mount point:

```
donnie@ubuntu:~$ veracrypt good_stuff good_stuff_dir
Enter password for /home/donnie/good_stuff:
Enter PIM for /home/donnie/good_stuff: 8891
Enter keyfile [none]:
Protect hidden volume (if any)? (y=Yes/n=No) [No]:
Enter your user password or administrator password:
donnie@ubuntu:~$
```

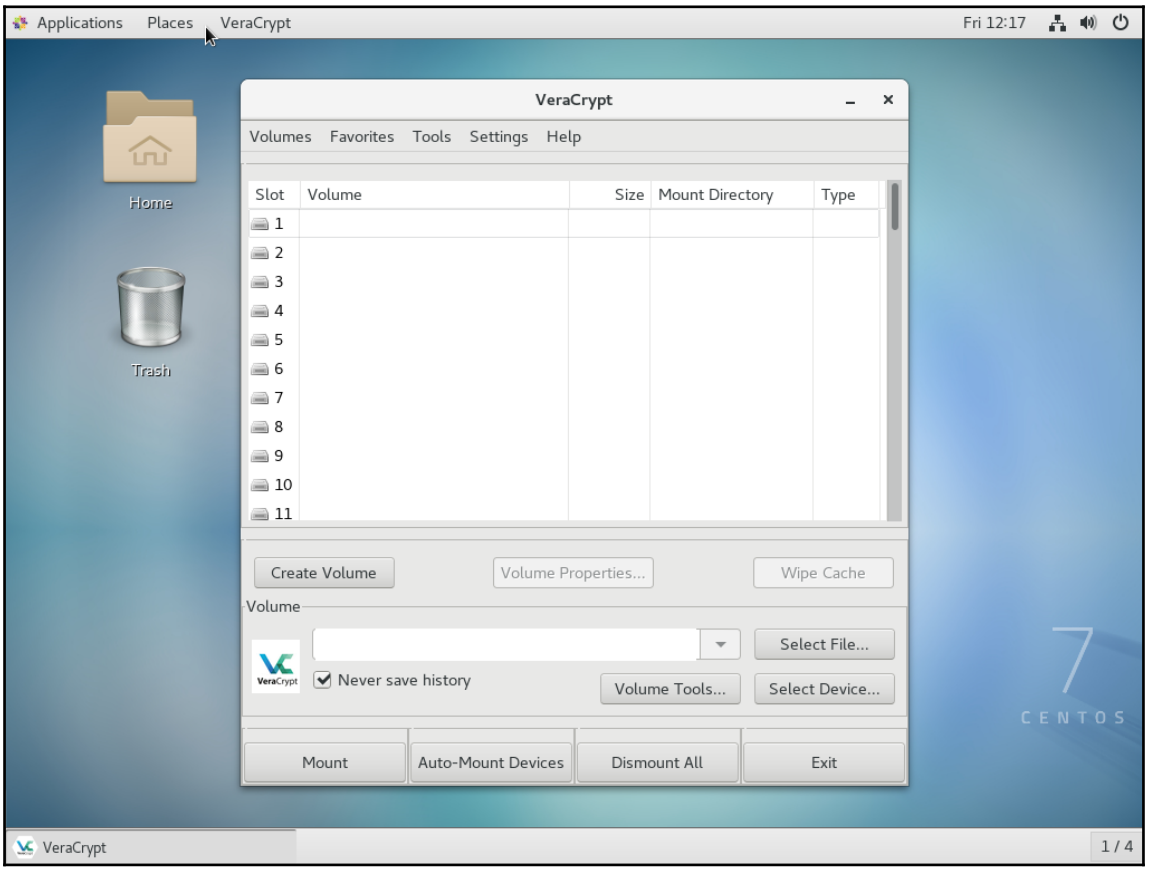
To see what VeraCrypt volumes you have mounted, use `veracrypt -l`:

```
donnie@ubuntu:~$ veracrypt -l
1: /home/donnie/secret_stuff /dev/mapper/veracrypt1
/home/donnie/secret_stuff_dir
2: /home/donnie/good_stuff /dev/mapper/veracrypt2
/home/donnie/good_stuff_dir
donnie@ubuntu:~$
```

And, that's all there is to it.

Using VeraCrypt in GUI mode

Desktop users of any of the supported operating systems can install the GUI variant of VeraCrypt. Be aware though, that you can't install both the console-mode variant and the GUI variant on the same machine, because one will overwrite the other. Here, you see the GUI version running on my CentOS 7 virtual machine:



Since the main focus of this book is sever security, I won't go into the details of the GUI version here. But, it's fairly self-explanatory, and you can view the full VeraCrypt documentation on their website.

You can get VeraCrypt from here: `https://www.veracrypt.fr/en/Home.html`

For the rest of this chapter, we'll turn our attention to the subject of protecting data in transit, by locking down Secure Shell.

Ensuring that SSH protocol 1 is disabled

By this stage in your Linux career, you should already know how to use Secure Shell, or SSH, to do remote logins and remote file transfers. What you may not know is that a default configuration of SSH is actually quite insecure.

SSH protocol version 1, the original SSH protocol, is severely flawed, and should never be used. It's still in most Linux distributions, but fortunately, it's always disabled by default. But, if you ever open your `/etc/ssh/sshd_config` file and see this:

```
Protocol 1
```

Or this:

```
Protocol 1, 2
```

Then you have a problem.

The Ubuntu main page for the `sshd_config` file says that protocol version 1 is still available for use with `legacy devices`. But, if you're still running devices that are that old, you need to start seriously thinking about doing some upgrades.

As Linux distributions get updated, you'll see SSH protocol 1 gradually being completely removed, as has happened with Red Hat and CentOS 7.4.

Creating and managing keys for password-less logins

The Secure Shell Suite, or SSH, is a great set of tools that provides secure, encrypted communications with remote servers. You can use the SSH component to remotely log into the command-line of a remote machine, and you can use either `scp` or `sftp` to securely transfer files. The default way to use any of these SSH components is to use the username and password of a person's normal Linux user account. So, logging into a remote machine from the terminal of my OpenSUSE workstation would look something like:

```
donnie@linux-0ro8:~> ssh donnie@192.168.0.8
donnie@192.168.0.8's password:
```

While it's true that the username and password go across the network in an encrypted format, making it hard for malicious actors to intercept, it's still not the most secure way of doing business. The problem is that attackers have access to automated tools that can perform brute-force password attacks against an SSH server. Botnets, such as the Hail Mary Cloud, perform continuous scans across the internet to find internet-facing servers with SSH enabled. If a botnet finds that the servers allow SSH access via username and password, it will then launch a brute-force password attack. Sadly, such attacks have been successful quite a few times, especially when the server operators allow the root user to log in via SSH.

This older article gives more details about the Hail Mary Cloud botnet: `http://futurismic.com/2009/11/16/the-hail-mary-cloud-slow-but-steady-brute-force-password-guessing-botnet/`

In the next section, we'll look at two ways to help prevent these types of attacks:

- Enable SSH logins through an exchange of public keys
- Disable the root user login through SSH

Creating a user's SSH key set

Each user has the ability to create his or her own set of private and public keys. It doesn't matter whether the user's client machine is running Linux, MacOS, or Cygwin on Windows. In all three cases, the procedure is exactly the same. To demo, I'll create keys on my OpenSUSE workstation and transfer the public key to one of my virtual machines. It doesn't matter which virtual machine I use, but since I haven't shown much love to the CentOS machine lately, I'll use it.

I'll begin by creating the keys on my OpenSUSE workstation:

```
donnie@linux-0ro8:~> ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/donnie/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/donnie/.ssh/id_rsa.
Your public key has been saved in /home/donnie/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:oqDpCvAptbE8srN6Z4FNXxgkhPhjh1sEKazfMpxhVI8 donnie@linux-0ro8
The key's randomart image is:
+---[RSA 2048]----+
|...*+..          |
|o.+ .+.          |
|.+ oE .o         |
|. B + . .        |
|.=+% ...S        |
|.*O*+...         |
|* Bo..           |
|++..o            |
|B= o             |
+----[SHA256]-----+
donnie@linux-0ro8:~>
```

There are several different types of keys that you can create, but the default 2048-bit RSA keys are considered as plenty strong enough for the foreseeable future. The private and public SSH keys work the same as we saw with GPG. You'll keep your private keys to yourself, but you can share the public key with the world, if you so desire. In this case though, I'm only going to share my public key with just one server.

When prompted for the location and name of the keys, I'll just hit *Enter* to accept the defaults. You could just leave the private key with a blank passphrase, but that's not a recommended practice.

Note that if you choose an alternative name for your key files, you'll need to type in the entire path to make things work properly. For example, in my case, I would specify the path for `donnie_rsa` keys as:

`/home/donnie/.ssh/donnie_rsa`

In the `.ssh` directory in my home directory, I can see the keys that I created:

```
donnie@linux-0ro8:~/.ssh> ls -l
total 12
-rw------- 1 donnie users 1766 Nov  2 17:52 id_rsa
-rw-r--r-- 1 donnie users  399 Nov  2 17:52 id_rsa.pub
-rw-r--r-- 1 donnie users 2612 Oct 31 18:40 known_hosts
donnie@linux-0ro8:~/.ssh>
```

The `id_rsa` key is the private key, with read and write permissions only for me. The `id_rsa.pub` public key has to be world-readable.

Transferring the public key to the remote server

Transferring my public key to a remote server allows the server to readily identify both me and my client machine. Before I can transfer the public key to the remote server, I need to add the private key to my session keyring. This requires two commands. (One command is to invoke the `ssh-agent`, and the other command actually adds the private key to the keyring.):

```
donnie@linux-0ro8:~> exec /usr/bin/ssh-agent $SHELL
donnie@linux-0ro8:~> ssh-add
Enter passphrase for /home/donnie/.ssh/id_rsa:
Identity added: /home/donnie/.ssh/id_rsa (/home/donnie/.ssh/id_rsa)
donnie@linux-0ro8:~>
```

Finally, I can transfer my public key to my CentOS server, which is at address `192.168.0.101`:

```
donnie@linux-0ro8:~> ssh-copy-id donnie@192.168.0.101
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to
filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
prompted now it is to install the new keys
```

```
donnie@192.168.0.101's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'donnie@192.168.0.101'"
and check to make sure that only the key(s) you wanted were added.

donnie@linux-0ro8:~>
```

The next time that I log in, I'll use the key exchange, and I won't have to enter a password:

```
donnie@linux-0ro8:~> ssh donnie@192.168.0.101
Last login: Wed Nov  1 20:11:20 2017
[donnie@localhost ~]$
```

So, now you're wondering, "*How is that secure if I can log in without entering my password?*" The answer is that once you close the client machine's terminal window that you used for logging in, the private key will be removed from your session keyring. When you open a new terminal and try to log in to the remote server, you'll see this:

```
donnie@linux-0ro8:~> ssh donnie@192.168.0.101
Enter passphrase for key '/home/donnie/.ssh/id_rsa':
```

Now, every time I log into this server, I'll need to enter the passphrase for my private key. (That is, unless I add it back to the session keyring with the two commands that I showed you in the preceding section.)

Disabling root user login

A few years ago, there was a somewhat celebrated case where malicious actors had managed to plant malware on quite a few Linux servers somewhere in southeast Asia. There were three reasons that the bad guys found this so easy to do:

- The internet-facing servers involved were set up to use username/password authentication for SSH
- The root user was allowed to log in through SSH
- User passwords, including the root user's password, were incredibly weak

All this meant that it was easy for Hail Mary to brute-force its way in.

Different distributions have different default settings for root user login. In the `/etc/ssh/sshd_config` file of your CentOS machine, you'll see this line:

```
#PermitRootLogin yes
```

Unlike what you have in most configuration files, the commented-out lines in `sshd_config` define the default settings for the Secure Shell daemon. So, this line indicates that the root user is indeed allowed to log in through SSH. To change that, I'll remove the comment symbol and change the setting to `no`:

```
PermitRootLogin no
```

To make the new setting take effect, I'll restart the SSH daemon, which is named `sshd` on CentOS, and is named `ssh` on Ubuntu:

```
sudo systemctl restart sshd
```

On the Ubuntu machine, the default setting looks a bit different:

```
PermitRootLogin prohibit-password
```

This means that the root user is allowed to log in, but only via a public key exchange. That's probably secure enough, if you really need to allow the root user to log in. But in most cases, you'll want to force admin users to log in with their normal user accounts, and to use `sudo` for their admin needs. So, in most cases, you can still change this setting to `no`.

Be aware that if you deploy an instance of Ubuntu Server on a cloud service, such as Azure, Rackspace, or Vultr, the service owners will have you log into the virtual machine with the root user account. The first thing you'll want to do is to create your own normal user account, log back in with that account, disable the root user account, and disable the root user login in `sshd_config`.

Disabling username/password logins

This is something that you'll only want to do after you've set up the key exchange with your clients. Otherwise, clients will be locked out of doing remote logins.

For both Ubuntu and CentOS machines, look for this line in the `sshd_config` file:

```
#PasswordAuthentication yes
```

Remove the comment symbol, change the parameter value to `no`, and restart the SSH daemon. The line should now look like this:

```
PasswordAuthentication no
```

Now, when the botnets scan your system, they'll see that doing a brute-force password attack would be useless. They'll then just go away and leave you alone.

Setting up a chroot environment for SFTP users

Secure File Transfer Protocol (**SFTP**) is a great tool for performing secure file transfers. There is a command-line client, but users will most likely use a graphical client, such as Filezilla. A common use-case for SFTP is to allow website owners to upload web content files to the proper content directories on a web server. With a default SSH setup, anyone who has a user account on a Linux machine can log in through either SSH or SFTP, and can navigate through the server's entire filesystem. What we really want for SFTP users is to prevent them from logging into a command-prompt via SSH, and to confine them to their own designated directories.

Creating a group and configuring the sshd_config file

With the exception of the slight difference in user-creation commands, this procedure works the same for either CentOS or Ubuntu. So, you can use either one of your virtual machines to follow along. We'll begin by creating an `sftpusers` group.

```
sudo groupadd sftpusers
```

Create the user accounts, and add them to the `sftpusers` group. We'll do both operations in one step. On your CentOS machine, the command for creating Max's account would be:

```
sudo useradd -G sftpusers max
```

On your Ubuntu machine, it would be:

```
sudo useradd -m -d /home/max -s /bin/bash -G sftpusers max
```

Open the `/etc/ssh/sshd_config` file in your favorite text editor. Find the line that says:

```
Subsystem sftp /usr/lib/openssh/sftp-server
```

Change it to:

```
Subsystem sftp internal-sftp
```

This setting allows you to disable normal SSH login for certain users.

At the bottom of the `sshd_config` file, add a `Match Group` stanza:

```
Match Group sftpusers
        ChrootDirectory /home
        AllowTCPForwarding no
        AllowAgentForwarding no
        X11Forwarding no
        ForceCommand internal-sftp
```

An important consideration here is that the `ChrootDirectory` has to be owned by the root user, and it can't be writable by anyone other than the root user. When Max logs in, he'll be in the `/home` directory, and will then have to `cd` into his own directory. This also means that you want for all users' home directories to have the restrictive `700` permissions settings, in order to keep everyone out of everyone else's stuff.

Save the file and restart the SSH daemon. Then, try to log on as Max through normal SSH, just to see what happens:

```
donnie@linux-0ro8:~> ssh max@192.168.0.8
max@192.168.0.8's password:
This service allows sftp connections only.
Connection to 192.168.0.8 closed.
donnie@linux-0ro8:~>
```

Okay, so he can't do that. Let's now have him try to log in through SFTP, and verify that he is in the `/home` directory:

```
donnie@linux-0ro8:~> sftp max@192.168.0.8
max@192.168.0.8's password:
Connected to 192.168.0.8.
drwx------    7 1000     1000         4096 Nov  4 22:53 donnie
drwx------    5 1001     1001         4096 Oct 27 23:34 frank
drwx------    3 1003     1004         4096 Nov  4 22:43 katelyn
drwx------    2 1002     1003         4096 Nov  4 22:37 max
sftp>
```

Now, let's see him try to `cd` out of the `/home` directory:

```
sftp> cd /etc
Couldn't stat remote file: No such file or directory
sftp>
```

So, our chroot jail does indeed work.

Hands-on lab – setting up a chroot directory for sftpusers group

For this lab, you can use either the CentOS virtual machine or the Ubuntu virtual machine. You'll add a group, then configure the `sshd_config` file to allow group members to only be able to log in via SFTP, and to confine them to their own directories. For the simulated client machine, you can use the terminal of your MacOS or Linux desktop machine, or Cygwin from your Windows machine:

1. Create the `sftpusers` group:

   ```
   sudo groupadd sftpusers
   ```

2. Create a user account for Max, and add him to the `sftpusers` group. On CentOS, do:

   ```
   sudo useradd -G sftpusers max
   ```

   On Ubuntu, do:

   ```
   sudo useradd -m -d /home/max -s /bin/bash -G sftpusers max
   ```

3. For Ubuntu, ensure that the users' home directories are all set with read, write, and execute permissions for only the directory's user. If that's not the case, do:

   ```
   sudo chmod 700 /home/*
   ```

4. Open the `/etc/ssh/sshd_config` file in your preferred text editor. Find the line that says:

   ```
   Subsystem sftp /usr/lib/openssh/sftp-server
   ```

   Change it to:

   ```
   Subsystem sftp internal-sftp
   ```

5. At the end of the `sshd_config` file, add the following stanza:

```
Match Group sftpusers
         ChrootDirectory /home
         AllowTCPForwarding no
         AllowAgentForwarding no
         X11Forwarding no
         ForceCommand internal-sftp
```

6. Restart the SSH daemon. On CentOS, do:

```
sudo systemctl sshd restart
```

   On Ubuntu, do:

```
sudo systemctl ssh restart
```

7. Have Max try to log in through normal SSH, to see what happens:

```
ssh max@IP_Address_of_your_vm
```

8. Now, have Max log in through SFTP. Once he's in, have him try to `cd` out of the `/home` directory:

```
sftp max@IP_Address_of_your_vm
```

9. End of Lab.

Summary

In this chapter, we've seen how to work with various encryption technologies that can help us safeguard our secrets. We started with GNU Privacy Guard for encrypting individual files. We then moved on to the disk, partition, and directory encryption utilities. LUKS and eCryptfs are specific to Linux, but we also looked at VeraCrypt, which can be used on any of the major operating systems.

In the next chapter, we'll take an in-depth look at the subject of discretionary access control. I'll see you there.