eCryptfs

This article describes basic usage of **eCryptfs (https://launchpad.net/ecryptfs)**. It guides you through the process of creating a private and secure encrypted directory within your `$HOME` directory to store sensitive files and private data.

Related articles

Disk encryption

In implementation eCryptfs differs from **dm-crypt**, which provides a *block device encryption layer*, while eCryptfs is an actual file-system – a **stacked cryptographic file system**. For comparison of the two you can refer to the **Disk encryption#Comparison table**. One distinguished feature is that the encryption is stacked on an existing filesystem; eCryptfs can be mounted onto any single existing directory and does not require a separate partition (or size pre-allocation).

Contents

- 1 Basics
  - 1.1 Deficiencies
- 2 Setup & mounting
  - 2.1 High-level tools (hardcoded paths)

- 2.1.1 Encrypting a data directory
- 2.1.2 Encrypting a home directory
- 2.1.3 Mounting
  - 2.1.3.1 Manually
  - 2.1.3.2 Auto-mounting
- 2.2 ecryptfs-simple
- 2.3 Manual setup
  - 2.3.1 With configuration files
  - 2.3.2 Raw mount command
    - 2.3.2.1 Mounting
    - 2.3.2.2 Auto-mounting
      - 2.3.2.2.1 Optional step
- 3 Usage
  - 3.1 Symlinking into the encrypted directory
  - 3.2 Removal of encryption
  - 3.3 Backup
- 4 See Also

Basics

As mentioned in the summary eCryptfs does not require special on-disk storage allocation effort, such as a separate partition or pre-allocated space. Instead, you can mount eCryptfs on top of any single directory to protect it. That includes, for example, a user's entire home directory or single dedicated directories within it. All cryptographic metadata is stored in the headers of files, so encrypted data can be easily moved, stored for backup and recovered. There are other advantages, but there are also drawbacks, for instance eCryptfs is not suitable for encrypting complete partitions which also means you cannot protect swap space with it (but you can, of course, combine it with **Dm-crypt/Swap encryption**). If you are just starting to set up disk encryption, swap encryption and other points to consider are covered in **Disk encryption#Preparation**.

To familiarize with eCryptfs a few points:

- As a stacked filesystem, a mounting of an eCryptfs directory refers to mounting a (stacked) encrypted directory to another **un**encrypted mount point (directory) at Linux kernel runtime.
- It is possible to share an encrypted directory between users. However, the encryption is linked to one passphrase so this must be shared as well. It is also possible to share a directory with differently encrypted files (different passphrases).
- A number of eCryptfs acronyms are used throughout the documentation:
  - The encrypted directory is referred to as the **lower** and the unencrypted as the **upper** directory throughout the eCryptfs documentation and this article. While not relevant for this article, the "overlay" filesystem introduced with Linux 3.18 uses (and **explains (https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Docum**

entation/filesystems/overlayfs.txt)) the same upper/lower nomenclature for the stacking of filesystems.
- the **mount** passphrase (or key) is what gives access to the encrypted files, i.e. unlocks the encryption. eCryptfs uses the term **wrapped** passphrase to refer to the cryptographically secured mount passphrase.
- a `FEFEK` refers to a **F**ile's **E**ncryption key **E**ncryption **Key** (see **kernel documentation (https://www.kernel.org/doc/Documentation/security/keys-ecryptfs.txt)**).
- a `FNEK` refers to a **F**ile **N**ame **E**ncryption **K**ey, a key to (optionally) encrypt the filenames stored in the encrypted directory.

Before using eCryptfs, the following disadvantages should be checked for applicability.

Deficiencies

- Ease of use

  The `ecryptfs-utils (https://www.archlinux.org/packages/?name=ecryptfs-utils)` package provides several different ways of setting up eCryptfs. The high-level "Ubuntu tools" are the easiest to use, but they hard-code the lower directory path and other settings, limiting their usefulness. The package also includes low-level tools which are fully configurable, but they are somewhat more difficult to use compared to alternatives like **EncFS**.

- File name length

  File names longer than 143 characters cannot be encrypted (with the `FNEK` option).[1] (https://bugs.launchpad.net/ecryptfs/+bug/344878) This can break some programs in your home directory (for example **Symfony** caching).

- Network storage mounts

  eCryptfs has long-standing **bugs (https://bugs.launchpad.net/ecryptfs/+bug/277578)** when used on top of NFS and possibly other networked filesystems. It is always possible to use eCryptfs on a local directory and then copy the encrypted files from the local directory to a network host. However, if you want to set up eCryptfs directly on top of an NFS mount, with no local copy of the files, eCryptfs may crash or behave incorrectly. If in doubt, **EncFS** may be a better choice in this case.

- Sparse files

  Sparse files written to eCryptfs will produce larger, non-sparse encrypted files in the lower directory. For example, in an eCryptfs directory running `truncate -s 1G file.img` creates a 1GB encrypted file on the underlying filesystem, with the corresponding resource (disk space, data throughput) requirements. If the same file were created on an unencrypted filesystem or a filesystem using **block device encryption**, it would only take a few kilobytes.

This should be considered before encrypting large portions of the directory structure, though in most cases the disadvantages will be minor. If you need to use large sparse files, you can work around this issue by putting the sparse files in an unencrypted directory or using block device encryption for them.

Setup & mounting

Before starting, check the eCryptfs documentation. It is distributed with a very good and complete set of **manual pages (http://ecryptfs.org/documentation.html)**.

eCryptfs has been included in Linux since version 2.6.19. Start by loading the `ecryptfs` module:

```
# modprobe ecryptfs
```

To actually mount an eCryptfs filesystem, you need to use userspace tools provided by the package **`ecryptfs-utils (https://www.archlinux.org/packages/?name=ecryptfs-utils)`** available in the **Official repositories**. Unfortunately, due to the poor design of these tools, you must choose between three ways of setting up eCryptfs with different tradeoffs:

1. Use the **high-level tools**, which set things up automatically but require the lower directory to be `~/.Private/`, and allow only one encrypted filesystem per user.

2. Use **ecryptfs-simple**, available from AUR, which is an easy way to mount eCryptfs filesystems using any lower directory and upper directory.
3. **#Manual setup**, which involves separate steps for loading the passphrase and mounting eCryptfs, but allows complete control over the directories and encryption settings.

High-level tools (hardcoded paths)

Most of the user-friendly convenience tools installed by the *ecryptfs-utils* package assume a very specific eCryptfs setup, namely the one that is officially used by Ubuntu (where it can be selected as an option during distro installation). Unfortunately, these choices are not just default options but are actually hard-coded in the tools. If this set-up does not suit your needs, then you can not use the convenience tools and will have to follow the steps at **#Manual setup** instead.

The set-up used by these tools is as follows:

- each user can have **only one encrypted directory** that is managed by these tools:
  - either full `$HOME` directory encryption, or
  - a single encrypted data directory (by default `~/Private/`, but this can be customized).
- the **lower directory** for each user is always `~/.Private/`
  (in the case of full home dir encryption, this will be a symlink to the actual location at `/home/.ecryptfs/$USER/.Private/`)
- the **encryption options** used are:

    - *cipher:* AES
    - *key length:* 16 bytes (128 bits)
    - *key management scheme:* passphrase
    - *plaintext passthrough:* enabled
- the **configuration / control info** for the encrypted directory is stored in a bunch of files at `~/.ecryptfs/`:

  (in the case of full home dir encryption, this will be a symlink to the actual location at `/home/.ecryptfs/$USER/.ecryptfs/`)
    - `Private.mnt` *[plain text file]* - contains the path where the upper directory should be mounted (e.g. `/home/lucy` or `/home/lucy/Private`)
    - `Private.sig` *[plain text file]* - contains the signature used to identify the mount passphrase in the kernel keyring
    - `wrapped-passphrase` *[binary file]* - the mount passphrase, encrypted with the login passphrase
    - `auto-mount`, `auto-umount` *[empty files]* - if they exist, the `pam_ecryptfs.so` module will (assuming it is loaded) automatically mount/unmount this encrypted directory when the user logs in/out

Encrypting a data directory

For a full `$HOME` directory encryption see **#Encrypting a home directory**

Before the data directory encryption is setup, decide whether it should later be mounted manually or automatically with the user log-in.

To encrypt a single data directory as a user and mount it manually later, run:

```
$ ecryptfs-setup-private --nopwcheck --noautomount
```

and follow the instructions. The option `--nopwcheck` enables you to choose a passphrase different to the user login passphrase and the option `--noautomount` is self-explanatory. So, if you want to setup the encrypted directory automatically on log-in later, just *leave out* both options right away.

The script will automatically create the `~/.Private/` and `~/.ecryptfs/` directory structures as described in the box above. It will also ask for two passphrases:

login passphrase
: This is the password you will have to enter each time you want to mount the encrypted directory. If you want auto-mounting on login to work, it has to be the same password you use to login to your user account.

mount passphrase
: This is used to derive the actual file encryption master key. Thus, you should not enter a custom one unless you know what you are doing - instead press Enter to let it auto-generate a secure random one. It will be encrypted using the login passphrase and stored in this encrypted form in `~/.ecryptfs/wrapped-passphrase`. Later it will automatically be decrypted ("unwrapped") again in RAM when needed, so you never have to enter it manually. Make sure this file does not get lost, otherwise you can never

access your encrypted folder again! You may want to run `ecryptfs-unwrap-passphrase` to see the mount passphrase in unencrypted form, write it down on a piece of paper, and keep it in a safe (or similar), so you can use it to recover your encrypted data in case the *wrapped-passphrase* file is accidentally lost/corrupted or in case you forget the login passphrase.

The mount point ("upper directory") for the encrypted folder will be at `~/Private` by default, however you can manually change this right after the setup command has finished running, by doing:

```
$ mv ~/Private /path/to/new/folder
$ echo /path/to/new/folder > ~/.ecryptfs/Private.mnt
```

To actually use your encrypted folder, you will have to mount it - see **#Mounting** below.

Encrypting a home directory

The wrapper script `ecryptfs-migrate-home` will set up an encrypted home directory for a user and take care of migrating any existing files they have in their not yet encrypted home directory.

To run it, the user in question must be logged out and own no processes. The best way to achieve this is to log the user out, log into a console as the root user, and check that `ps -U username` returns no output. You also need to ensure that you have **rsync (https://**

www.archlinux.org/packages/?name=rsync) and lsof (https://www.archlinux.org/packages/?name=lsof) installed. Once the prerequisites have been met, run:

```
# modprobe ecryptfs
# ecryptfs-migrate-home -u username
```

and follow the instructions. After the wrapper script is complete, follow the instructions for auto-mounting - see **#Auto-mounting** below. It is imperative that the user logs in *before* the next reboot, to complete the process.

Once everything is working, the unencrypted backup of the users home directory, which is saved to `/home/username.random_characters`, can and should be deleted.

Mounting

Manually

Executing the wrapper

```
$ ecryptfs-mount-private
```

and entering the passphrase is all needed to mount the encrypted directory to the **above** described *upper directory* `~/Private`.

Likewise, executing

```
$ ecryptfs-umount-private
```

will unmount it again.

Tip: If it is not required to access the private data permanently during a user session, maybe define an **alias** to speed the manual step up.

The tools include another script that can be very handy to access an encrypted `.Private` data or home directory. Executing `ecryptfs-recover-private` as root will search the system (or an optional specific path) for the directory, interactively query the passphrase for it and mount the directory. It can, for example, be used from a live-CD or different system to access the encrypted data in case of a recovery. Note that if booting from an Arch Linux ISO you must first install the **ecryptfs-utils (https://www.archlinux.org/packages/?name=ecryptfs-utils)** to it. Further, it will only be able to mount `.Private` directories created with the Ubuntu tools.

Auto-mounting

The default way to auto-mount an encrypted directory is via **PAM**. See **pam_ecryptfs(8) (https://jlk.fjfi.cvut.cz/arch/manpages/man/pam_ecryptfs.8)** and - for more details - 'PAM MODULE' in:

```
/usr/share/doc/ecryptfs-utils/README
```

For auto-mounting it is required that the passphrase to access the encrypted directory is synchronised with the user log-in.

The following steps set it up:

1. Check if `~/.ecryptfs/auto-mount`, `~/.ecryptfs/auto-umount` and `~/.ecryptfs/wrapped-passphrase` exist (these are automatically created by *ecryptfs-setup-private*).

2. Add *ecryptfs* to the pam-stack exactly as following to allow transparent unwrapping of the passphrase on login:

Open `/etc/pam.d/system-auth` and *after* the line containing `auth required pam_unix.so` add:

```
auth    required    pam_ecryptfs.so unwrap
```

Next, *above* the line containing `password required pam_unix.so` insert:

```
password    optional    pam_ecryptfs.so
```

And finally, *after* the line `session required pam_unix.so` add:

```
session    optional    pam_ecryptfs.so unwrap
```

3. Re-login and check output of *mount* which should now contain a mountpoint, e.g.:

```
/home/$USER/.Private on /home/$USER/Private type ecryptfs (...)
```

for the user's encrypted directory. It should be perfectly readable at `~$HOME/Private/`.

The latter should be automatically unmounted and made unavailable when the user logs off.

Note: The above changes to `system-auth` enable auto-mounting for normal login. If you switch users instead, using `su -` or `su -l`, you need to apply similar changes also to `/etc/pam.d/su-l`.

Warning: Unfortunately the automatic unmounting is susceptible to **break (https://bbs.archlinux.org/viewtopic.php?id=194509)** with systemd and bugs are filed against it.**[2] (https://bugs.freedesktop.org/show_bug.cgi?id=72759) [3] (https://nwrickert2.wordpress.com/2013/12/16/systemd-user-manager-ecryptfs-and-opensuse-13-1/) [4] (https://bugs.launchpad.net/ubuntu/+source/ecryptfs-utils/+bug/313812/comments/43) [5] (http://lists.alioth.debian.org/pipermail/pkg-systemd-maintainers/2014-October/004088.html)** If you experience this problem, you can test it by commenting out `-session optional pam_systemd.so` in `/etc/pam.d/system-login`. However, this is no solution because commenting out will break other systemd functionalities.

ecryptfs-simple

Use **ecryptfs-simple (http://xyne.archlinux.ca/projects/ecryptfs-simple/)** if you just want to use eCryptfs to mount arbitrary directories the way you can with **EncFS**. ecryptfs-simple does not require root privileges or entries in `/etc/fstab`, nor is it limited to hard-coded directories such as `~/.Private`. The package is available to be **installed** as **`ecryptfs-simple (https://aur.archlinux.org/packages/ecryptfs-simple/)`**[AUR] and from **Xyne's repos (http://xyne.archlinux.ca/repos/)**.

As the name implies, usage is simple:

```
# simple mounting
ecryptfs-simple /path/to/foo /path/to/bar
```

```
# automatic mounting: prompts for options on the first mount of a directory then reloads them next time
ecryptfs-simple -a /path/to/foo /path/to/bar
```

```
# unmounting by source directory
ecryptfs-simple -u /path/to/foo
```

```
# unmounting by mountpoint
ecryptfs-simple -u /path/to/bar
```

Manual setup

The following details instructions to set up eCryptfs encrypted directories manually. This involves two steps. First, the passphrase is processed and loaded into the kernel keyring. Second, the filesystem is actually mounted using the key from the keyring.

There are two ways to add the passphrase to the kernel keyring in the first step. The simpler option is `ecryptfs-add-passphrase`, which uses a single passphrase to encrypt the files. The disadvantage is that you cannot change the passphrase later. It works like this:

```
$ ecryptfs-add-passphrase
Passphrase:
Inserted auth tok with sig [78c6f0645fe62da0] into the user session keyring
```

You can also pipe a passphrase into `ecryptfs-add-passphrase -`. Keep in mind that if you leave your passphrase in a file, it will usually defeat the purpose of using encryption.

As an alternative to a plain passphrase, you can use a "wrapped passphrase", where the files are encrypted using a randomly generated key, which is itself encrypted with your passphrase and stored in a file. In this case, you can change your passphrase by unwrapping the key file with your old passphrase and rewrapping it using your new passphrase.

In the following we do a generation similar to the **source (http://bazaar.launchpad.net/~ecryptfs/ecryptfs/trunk/view/head:/src/utils/ecryptfs-setup-private#L96)** and then use *ecryptfs-wrap-passphrase* to wrap it with the password ("Arch") to `~/.ecryptfs/wrapped-passphrase`:

```
$ printf "%s\n%s" $(od -x -N 100 --width=30 /dev/random | head -n 1 | sed "s/^0000000//" | sed "s/\s*//g") "Arch" | ecryptfs-wrap-passphrase /home/username/.ecryptfs/wrapped-passphrase
```

Next, we can enter our passphrase "Arch" to load the key into the keyring:

```
$ printf "%s" "Arch" | ecryptfs-insert-wrapped-passphrase-into-keyring /home/username/.ecryptfs/wrapped-passphrase -
Inserted auth tok with sig [7c5d3dd8a1b49db0] into the user session keyring
```

In either case, when you successfully add the passphrase to the kernel keyring, you will get a "key signature" like `78c6f0645fe62da0` which you will need in the next step.

There are two different ways of manually mounting eCryptfs, described in the following sections. The first way, using `mount.ecryptfs_private`, can be run as a regular user and involves setting up some configuration files. This method does not allow you to change the encryption settings, such as key size. The second way is to use a raw `mount` command, which gives you complete control over all settings, but requires you to either run it as root, or add an entry to `/etc/fstab` which lets a user mount eCryptfs.

Tip: The following examples use an encrypted directory (`.secret`) different to the default, hard-coded `.Private` in the Ubuntu tools. This is on purpose to avoid problems of erroneous **#Auto-mounting** when the system has PAM setup for it, as well as problems with other tools using the hard-coded defaults.

With configuration files

This method involves running `mount.ecryptfs_private` from the **ecryptfs-utils (https://www.archlinux.org/packages/?name=ecryptfs-utils)** package, after first loading your passphrase. This binary requires no root privileges to work by default.

First choose a name for your configuration files in `~/.ecryptfs` and decide on the lower and upper directories. In this example we use `secret` for the configuration files, put in encrypted data in `~/.secret`, and mount the decrypted files at `~/secret`. Create the required directories:

```
$ mkdir ~/.secret ~/secret ~/.ecryptfs
```

Now specify the directories in `~/.ecryptfs/secret.conf`, using full paths. Its format looks like the one in `/etc/fstab` without the mount options:

```
$ echo "$HOME/.secret $HOME/secret ecryptfs" > ~/.ecryptfs/secret.conf
```

Write the key signature you got from `ecryptfs-add-passphrase` or `ecryptfs-insert-wrapped-passphrase-into-keyring` (see above) into `~/.ecryptfs/secret.sig`:

```
$ echo 78c6f0645fe62da0 > ~/.ecryptfs/secret.sig
```

If you also want to enable filename encryption, add a second passphrase to the keyring (or reuse the first passphrase) and **append** its key signature to `~/.ecryptfs/secret.sig`:

```
$ echo 326a6d3e2a5d444a >> ~/.ecryptfs/secret.sig
```

Finally, mount `~/.secret` on `~/secret`:

```
$ mount.ecryptfs_private secret
```

When you are done, unmount it:

```
$ umount.ecryptfs_private secret
```

Raw mount command

By running the actual `mount` command manually, you get complete control over the encryption options. The disadvantage is that you need to either run `mount` as root, or add an entry to `/etc/fstab` for each eCryptfs directory so users can mount them.

First create your private directories. In this example, we use the same ones as the previous section:

```
$ mkdir -m 700 ~/.secret
$ mkdir -m 500 ~/secret
```

To summarize:

- Actual encrypted data will be stored in the lower `~/.secret` directory
- While mounted, decrypted data will be available in `~/secret` directory
  - While not mounted nothing can be written to this directory
  - While mounted it has the same permissions as the lower directory

Now, supposed you have created the **wrapped keyphrase** above, you need to insert the encryption key once to the root user's keyring:

```
# printf "%s" "Arch" | ecryptfs-insert-wrapped-passphrase-into-keyring /home/username/.ecryptfs/wrapped-passphrase -
Inserted auth tok with sig [7c5d3dd8a1b49db0] into the user session keyring
```

so that the followng mount command succeeds:

```
# mount -i -t ecryptfs ~/.secret ~/secret -o ecryptfs_sig=7c5d3dd8a1b49db0,ecryptfs_fnek_sig=7c5d3dd8a1b49db0,ecryptfs_cipher=aes,ecryptfs_key_bytes
=32,ecryptfs_unlink_sigs
```

- `ecryptfs_sig` sets the data passphrase key signature.
- `ecryptfs_fnek_sig` sets the filename passphrase key signature; you can omit this option if you do not want to encrypt filenames.
- `ecryptfs_key_bytes` can be 16, 24, or 32 to change the encryption key size.
- `ecryptfs_unlink_sigs` will remove the passphrase(s) from the keyring when you unmount, so you have to add the passphrase(s) back again in order to re-mount the filesystem.
- There are a few other options listed in the `ecryptfs` man page.

Tip: There is a `mount.ecryptfs` tool, which you can run as root to enter the mount settings interactively. Once you have used it to mount eCryptfs, you can check `/etc/mtab` to find out what options it used.

Once you have chosen the right mount options, you can add an entry to `/etc/fstab` so regular users can mount eCryptfs on these directories. Copy the mount options to a new `/etc/fstab` entry and add the options `user` and `noauto`. The full entry will look similar to (bold entries added):

/etc/fstab

```
/home/username/.secret /home/username/secret ecryptfs noauto,user,ecryptfs_sig=7c5d3dd8a1b49db0,ecryptfs_fnek_sig=7c5d3dd8a1b49db0,ecryptfs_cipher=aes,ecryptfs_key_bytes=32,ecryptfs_unlink_sigs 0 0
```

- The `noauto` option is important, because otherwise systemd will error trying to mount the entry directly on boot.
- The `user` option enables to mount the directory as a user.
  - The user mount will default to option `noexec`. If you want to have at least executable files in your private directory, you can add `exec` to the fstab options.

The setup is now complete and the directory should be mountable by the user.

Mounting

To mount the encrypted directory as the user, the passphrase must be unwrapped and made available in the user's keyring. Following above section example:

```
$ ecryptfs-insert-wrapped-passphrase-into-keyring /home/username/.ecryptfs/wrapped-passphrase
Passphrase:
Inserted auth tok with sig [7c5d3dd8a1b49db0] into the user session keyring
```

Now the directory can be mounted without the mount helper questions:

```
$ mount -i /home/username/secret
```

and files be placed into the `secret` directory. The above two steps are necessary every time to mount the directory manually.

To unmount it again:

```
$ umount /home/username/secret
```

To finalize, the preliminary passphrase to wrap the encryption passphrase may be changed:

```
$ ecryptfs-rewrap-passphrase /home/username/.ecryptfs/wrapped-passphrase
Old wrapping passphrase:
New wrapping passphrase:
New wrapping passphrase (again):
```

The un-mounting should also clear the keyring, to check the user's keyring or clear it manually:

```
$ keyctl list @u
$ keyctl clear @u
```

Note: One should remember that `/etc/fstab` is for system-wide partitions only and should not generally be used for user-specific mounts

Auto-mounting

Different methods can be employed to automount the previously defined user-mount in `/etc/fstab` on login. As a first general step, follow point (1) and (2) of **#Auto-mounting**.

Then, if you login via console, a simple way is to specify the **user-interactive** *mount* and *umount* in the user's shell configuration files, for example **Bash#Configuration files**.

Another method is to automount the eCryptfs directory on user login using **pam_mount**. To configure this method, add the following lines to `/etc/security/pam_mount.conf.xml` :

```
<luserconf name=".pam_mount.conf.xml" />
<mntoptions require="" />
<lclmount>mount -i %(VOLUME) "%(before=\"-o\" OPTIONS)"</lclmount>
```

Please prefer writing manually these lines instead of simply copy/pasting them (especially the lclmount line), otherwise you might get some corrupted characters. Explanation:

- the first line indicates where the user-based configuration file is located (here `~/.pam_mount.conf.xml`)
- the second line overwrites the default required mount options which are unnecessary ("nosuid,nodev")
- the last line indicates which mount command to run (eCryptfs needs the `-i` switch).

Then set the volume definition, preferably to `~/.pam_mount.conf.xml` :

```
<pam_mount>
    <volume noroot="1" fstype="ecryptfs" path="/home/user/.secret/" mountpoint="/home/user/secret/"/>
</pam_mount>
```

"noroot" is needed because the encryption key will be added to the user's keyring.

Finally, edit `/etc/pam.d/login` as described in the **pam_mount** article.

Optional step

To avoid wasting time needlessly unwrapping the passphrase you can create a script that will check *pmvarrun* to see the number of open sessions:

```
#!/bin/sh
#
```

```
#    /usr/local/bin/doecryptfs

exit $(/usr/sbin/pmvarrun -u$PAM_USER -o0)
```

With the following line added before the eCryptfs unwrap module in your PAM stack:

```
auth    [success=ignore default=1]    pam_exec.so     quiet /usr/local/bin/doecryptfs
auth    required                      pam_ecryptfs.so unwrap
```

The article suggests adding these to `/etc/pam.d/login`, but the changes will need to be added to all other places you login, such as `/etc/pam.d/kde`.

Usage

Symlinking into the encrypted directory

Besides using your private directory as storage for sensitive files, and private data, you can also use it to protect application data. **Firefox** for example has an internal password manager, but the browsing history and cache can also be sensitive. Protecting it is easy:

```
$ mv ~/.mozilla ~/Private/mozilla
$ ln -s ~/Private/mozilla ~/.mozilla
```

Removal of encryption

There are no special steps involved, if you want to remove your private directory. Make sure it is un-mounted and delete the respective lower directory (e.g. `~/.Private`), along with all the encrypted files. After also removing the related encryption signatures and configuration in `~/.ecryptfs`, all is gone.

If you were **#Using the Ubuntu tools**[**broken link**: invalid section] to setup a single directory encryption, you can directly follow the steps detailed by:

```
$ ecryptfs-setup-private --undo
```

and follow the instructions.

Backup

If you want to move a file out of the private directory just move it to the new destination while `~/Private` is mounted.

With eCryptfs the cryptographic metadata is stored in the header of the files. Setup variants explained in this article separate the directory with encrypted data from the mount point. The unencrypted mount point is fully transparent and available for a backup. Obviously this has to be considered for automated backups, if one has to avoid leaking sensitive unencrypted data into a backup.

You can do backups, or incremental backups, of the encrypted (e.g. `~/.Private`) directory, treating it like any other directory.

Further points to note:

- If you used the Ubuntu tools for **#Encrypting a home directory**, be aware the location of the lower directory with the encrypted files is *outside* the regular user's `$HOME` at `/home/.ecryptfs/$USER/.Private`.
- It should be ensured to include the eCryptfs setup files (located in `~/.ecryptfs` usually) into the regular or a separate backup.
- If you use special filesystem mount options, for example `ecryptfs_xattr`, do extra checks on restore integrity.

See Also

- **eCryptfs (http://ecryptfs.org/documentation.html)** - Manpages and project home
- **Security audit (https://defuse.ca/audits/ecryptfs.htm)** of eCryptfs by Taylor Hornby (January 22, 2014).
- **eCryptfs and $HOME (http://sysphere.org/~anrxc/j/articles/ecryptfs/index.html)** by Adrian C. (anrxc) - Article with installation instructions and discussion of eCryptfs usage
- **Chromium data protection (http://www.chromium.org/chromium-os/chromiumos-design-docs/protecting-cached-user-data)** (November 2009) - Design document detailing

encryption options for Chromium OS, including explanation on its eCryptfs usage
- **eCryptfs design (http://ecryptfs.sourceforge.net/ecryptfs.pdf)** by Michael Halcrow (May 2005) - Original design document detailing and discussing eCryptfs

Retrieved from "https://wiki.archlinux.org/index.php?title=ECryptfs&oldid=510376"
