dm-crypt/Device encryption

< Dm-crypt
This section covers how to manually utilize *dm-crypt* from the command line to encrypt a system.

1 Preparation

Before using `cryptsetup (https://archlinux.org/packages/?name=cryptsetup)`, always make sure the `dm_crypt` **kernel module** is loaded.

2 Cryptsetup usage

Cryptsetup is the command line tool to interface with *dm-crypt* for creating, accessing and managing encrypted devices. The tool was later expanded to support different encryption types that rely on the Linux kernel **d**evice-**m**apper and the **crypt**ographic modules. The most notable expansion was for the Linux Unified Key Setup (LUKS) extension, which stores all of the needed setup information for dm-crypt on the disk itself and abstracts partition and key management in an attempt to improve ease of use. Devices accessed via the device-mapper are called block devices. For further information see **Data-at-rest encryption#Block device encryption**.

The tool is used as follows:

```
# cryptsetup OPTIONS action action-specific-options device dmname
```

It has compiled-in defaults for the options and the encryption mode, which will be used if no others are specified on the command line. Have a look at

```
$ cryptsetup --help
```

which lists options, actions and the default parameters for the encryption modes in that order. A full list of options can be found on the man page. Since different parameters are required or optional, depending on encryption mode and action, the following sections point out differences further. Block device encryption is fast, but speed matters a lot too. Since changing an encryption cipher of a block device after setup is difficult, it is important to check *dm-crypt* performance for the individual parameters in advance:

```
$ cryptsetup benchmark
```

can give guidance on deciding for an algorithm and key-size prior to installation. If certain AES ciphers excel with a considerable higher throughput, these are probably the ones with hardware support in the CPU.

Tip: You may want to practise encrypting a virtual hard drive in a **virtual machine** when learning.

2.1 Cryptsetup passphrases and keys

An encrypted block device is protected by a key. A key is either:

- a passphrase: see Security#Passwords.
- a keyfile, see #Keyfiles.

Both key types have default maximum sizes: passphrases can be up to 512 characters and keyfiles up to 8192 KiB.

An important distinction of *LUKS* to note at this point is that the key is used to unlock the master-key of a LUKS-encrypted device and can be changed with root access. Other encryption modes do not support changing the key after setup, because they do not employ a master-key for the encryption. See Data-at-rest encryption#Block device encryption for details.

3 Encryption options with dm-crypt

Cryptsetup supports different encryption operating modes to use with *dm-crypt*:

- `--type luks` for using the default LUKS format version (LUKS1 with cryptsetup (https://archlinux.org/packages/?name=cryptsetup) < 2.1.0, LUKS2 with cryptsetup (https://archlinux.org/packages/?name=cryptsetup) ≥ 2.1.0),
- `--type luks1` for using LUKS1, the most common version of LUKS,
- `--type luks2` for using LUKS2, the latest available version of LUKS that allows additional extensions,
- `--type plain` for using dm-crypt plain mode,
- `--type loopaes` for a loopaes legacy mode,
- `--type tcrypt` for a TrueCrypt compatibility mode.
- `--type bitlk` for a BitLocker compatibility mode. See cryptsetup(8) § BITLK (WINDOWS BITLOCKER COMPATIBLE) EXTENSION (https://man.archlinux.org/man/cryptsetup.8#BITLK_(WINDOWS_BITLOCKER_COMPATIBLE)_EXTENSION).

The basic cryptographic options for encryption cipher and hashes available can be used for all modes and rely on the kernel cryptographic backend features. All that are loaded and available to use as options at runtime can be viewed with:

```
$ less /proc/crypto
```

Tip: If the list is short, execute `$ cryptsetup benchmark` which will trigger loading available modules.

The following introduces encryption options for the `luks`, `luks1`, `luks2` and `plain` modes. Note that the tables list options used in the respective examples in this article and not all available ones.

3.1 Encryption options for LUKS mode

The *cryptsetup* action to set up a new dm-crypt device in LUKS encryption mode is `luksFormat`. Unlike what the name implies, it does not format the device, but sets up the LUKS device header and encrypts the master-key with the desired cryptographic options.

In order to create a new LUKS container with the compiled-in defaults listed by `cryptsetup --help`, simply execute:

```
# cryptsetup luksFormat device
```

As of cryptsetup 2.4.0, this is equivalent to:

```
# cryptsetup --type luks2 --cipher aes-xts-plain64 --hash sha256 --iter-time 2000 --key-size 256 --pbkdf
argon2id --use-urandom --verify-passphrase luksFormat device
```

Defaults are compared with a cryptographically higher specification example in the table below, with accompanying comments:

Options	Cryptsetup 2.1.0 defaults	Example	Comment
--cipher **-c**	`aes-xts-plain64`	`aes-xts-plain64`	**Release 1.6.0 (https://www.kernel.org/pub/linux/utils/cryptsetup/v1.6/v1.6.0-ReleaseNotes)** changed the defaults to an AES **cipher** in **XTS** mode (see item 5.16 **of the FAQ (https://gitlab.com/cryptsetup/cryptsetup/wiki/FrequentlyAskedQuestions#5-security-aspects)**). It is advised against using the previous default `--cipher aes-cbc-essiv` because of its known **issues** and practical **attacks (https://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/)** against them.
--key-size **-s**	`256` (`512` for XTS)	`512`	By default a 512 bit key-size is used for XTS ciphers. Note however that **XTS splits the supplied key in half**, so this results in AES-256 being used.
--hash **-h**	`sha256`	`sha512`	Hash algorithm used for **key derivation**. Release 1.7.0 changed defaults from `sha1` to `sha256` "*not for security reasons [but] mainly to prevent compatibility problems on hardened systems where SHA1 is already [being] phased out*"**[1] (https://www.kernel.org/pub/linux/utils/cryptsetup/v1.7/v1.7.0-ReleaseNotes)**. The former default of `sha1` can still be used for compatibility with older versions of *cryptsetup* since it is **considered secure (https://gitlab.com/cryptsetup/cryptsetup/wiki/FrequentlyAskedQuestions#5-security-aspects)** (see item 5.20).
--iter-time **-i**	`2000`	`5000`	Number of milliseconds to spend with PBKDF passphrase processing. Release 1.7.0 changed defaults from `1000` to `2000` to "*try to keep PBKDF2 iteration count still high enough and also still acceptable for users.*"**[2] (https://www.kernel.org/pub/linux/utils/cryptsetup/v1.7/v1.7.0-ReleaseNotes)**. This option is only relevant for LUKS operations that set or change passphrases, such as `luksFormat` or `luksAddKey`. Specifying 0 as parameter selects the compiled-in default.
--use-urandom	`--use-urandom`	`--use-random`	Selects which **random number generator** to use. Note that **/dev/random blocking pool has been removed (https://lwn.net/Articles/808575/)**. Therefore, `--use-random` flag is now equivalent to `--use-urandom`.
--verify-passphrase **-y**	Yes	-	Enabled by default in Arch Linux for `luksFormat` and `luksAddKey`.

The properties of LUKS features and options are described in the **LUKS1 (https://gitlab.com/cryptsetup/cryptsetup/wikis/Specification)** (pdf) and **LUKS2 (https://gitlab.com/cryptsetup/cryptsetup/blob/master/docs/on-disk-format-luks2.pdf)** (pdf) specifications.

Tip: The project developers' **devconfcz2016 (https://mbroz.fedorapeople.org/talks/DevConf2016/devconf2016-luks2.pdf)** (pdf) presentation summarizes the motivation for the major specification update to LUKS2.

3.1.1 Iteration time

From **cryptsetup FAQ§2.1 (https://gitlab.com/cryptsetup/cryptsetup/-/wikis/FrequentlyAskedQuestions#2-setup)** and **§3.4 (https://gitlab.com/cryptsetup/cryptsetup/-/wikis/FrequentlyAskedQuestions#3-common-problems)**:

> The unlock time for a key-slot [...] is calculated when setting a passphrase. By default it is 1 second (2 seconds for LUKS2). [...]
> Passphrase iteration count is based on time and hence security level depends on CPU power of the system the LUKS container is created on. [...]
> If you set a passphrase on a fast machine and then unlock it on a slow machine, the unlocking time can be much longer.

As such, it is better to always create a container on the machine where it will be most often accessed.

Read the rest of those sections for advice on how to correctly adjust the iteration count should the need arise.

3.1.2 Sector size

See **Advanced Format#dm-crypt**.

3.2 Encryption options for plain mode

In dm-crypt *plain* mode, there is no master-key on the device, hence, there is no need to set it up. Instead the encryption options to be employed are used directly to create the mapping between an encrypted disk and a named device. The mapping can be created against a partition or a full device. In the latter case not even a partition table is needed.

To create a *plain* mode mapping with cryptsetup's default parameters:

```
# cryptsetup options open --type plain device dmname
```

Executing it will prompt for a password, which should have very high entropy, and the `--verify-passphrase` option can be used but is not a default. In general it is advisable to make exact note of the encryption options used for the creation, because they can not be derived from the encrypted device, or an optional key-file, and upstream defaults may change.

Below a comparison of default parameters with the example in **dm-crypt/Encrypting an entire system#Plain dm-crypt**.

Option	Cryptsetup 2.1.0 defaults	Example	Comment
-h **--hash**	`ripemd160`	-	The hash is used to create the key from the passphrase; it is not used on a keyfile.
-c **--cipher**	`aes-cbc-essiv:sha256`	`aes-xts-plain64`	The cipher consists of three parts: cipher-chainmode-IV generator. Please see **Data-at-rest encryption#Ciphers and modes of operation** for an explanation of these settings, and the **DMCrypt documentation (https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt)** for some of the options available.
-s **--key-size**	`256`	`512`	The key size (in bits). The size will depend on the cipher being used and also the chainmode in use. Xts mode requires twice the key size of cbc.
-b **--size**	real size of target disk	`2048` (mapped device will be 512B×2048=1MiB)	Limit the maximum size of the device (in 512-byte sectors).
-o **--offset**	`0`	`0`	The offset from the beginning of the target disk (in 512-byte sectors) from which to start the mapping.
-p **--skip**	`0`	`2048` (512B×2048=1MiB will be skipped)	The number of 512-byte sectors of encrypted data to skip at the beginning.
-d **--key-file**	default uses a passphrase	`/dev/sdZ` (or e.g. `/boot/keyfile.enc`)	The device or file to be used as a key. See **#Keyfiles** for further details.
--keyfile-offset	`0`	`0`	Offset from the beginning of the file where the key starts (in bytes). This option is supported from *cryptsetup* 1.6.7 onwards.
-l **--keyfile-size**	`8192kB`	- (default applies)	Limits the bytes read from the key file. This option is supported from *cryptsetup* 1.6.7 onwards.

Using the device `/dev/sdX`, the above right column example results in:

```
# cryptsetup --cipher=aes-xts-plain64 --offset=0 --key-file=/dev/sdZ --key-size=512 open --type=plain /dev/sdX enc
```

Unlike encrypting with LUKS, the above command must be executed *in full* whenever the mapping needs to be re-established, so it is important to remember the cipher, hash and key file details. We can now check that the mapping has been made:

```
# fdisk -l
```

An entry should now exist for `/dev/mapper/enc`.

4 Encrypting devices with cryptsetup

This section shows how to employ the options for creating new encrypted block devices and accessing them manually.

Warning: GRUB's support for LUKS2 is limited; see **GRUB#Encrypted /boot** for details. Use LUKS1 (`cryptsetup luksFormat --type luks1`) for partitions that GRUB will need to unlock.

4.1 Encrypting devices with LUKS mode

4.1.1 Formatting LUKS partitions

In order to setup a partition as an encrypted LUKS partition execute:

```
# cryptsetup luksFormat device
```

You will then be prompted to enter a password and verify it.

See **#Encryption options for LUKS mode** for command line options.

You can check the results with:

```
# cryptsetup luksDump device
```

You will note that the dump not only shows the cipher header information, but also the key-slots in use for the LUKS partition.

The following example will create an encrypted root partition on `/dev/sda1` using the default AES cipher in XTS mode with an effective 256-bit encryption

```
# cryptsetup -s 512 luksFormat /dev/sda1
```

4.1.1.1 Using LUKS to format partitions with a keyfile

When creating a new LUKS encrypted partition, a keyfile may be associated with the partition on its creation using:

```
# cryptsetup luksFormat device /path/to/mykeyfile
```

See **#Keyfiles** for instructions on how to generate and manage keyfiles.

4.1.2 Unlocking/Mapping LUKS partitions with the device mapper

Once the LUKS partitions have been created, they can then be unlocked.

The unlocking process will map the partitions to a new device name using the device mapper. This alerts the kernel that *`device`* is actually an encrypted device and should be addressed through LUKS using the `/dev/mapper/`*`dm_name`* so as not to overwrite the encrypted data. To guard against accidental overwriting, read about the possibilities to **backup the cryptheader** after finishing setup.

In order to open an encrypted LUKS partition execute:

```
# cryptsetup open device dm_name
```

You will then be prompted for the password to unlock the partition. Usually the device mapped name is descriptive of the function of the partition that is mapped. For example the following unlocks a root luks partition `/dev/sda1` and maps it to device mapper named `root` :

```
# cryptsetup open /dev/sda1 root
```

Once opened, the root partition device address would be `/dev/mapper/root` instead of the partition (e.g. `/dev/sda1`).

For setting up LVM ontop the encryption layer the device file for the decrypted volume group would be anything like `/dev/mapper/root` instead of `/dev/sda1` . LVM will then give additional names to all logical volumes created, e.g. `/dev/lvmpool/root` and `/dev/lvmpool/swap` .

In order to write encrypted data into the partition it must be accessed through the device mapped name. The first step of access will typically be to **create a file system**. For example:

```
# mkfs -t ext4 /dev/mapper/root
```

The device `/dev/mapper/root` can then be **mounted** like any other partition.

To close the LUKS container, unmount the partition and do:

```
# cryptsetup close root
```

4.1.3 Using a TPM to store keys

See **Trusted Platform Module#Data-at-rest encryption with LUKS**.

4.2 Encrypting devices with plain mode

The creation and subsequent access of a *dm-crypt* plain mode encryption both require not more than using the *cryptsetup* `open` action with correct **parameters**. The following shows that with two examples of non-root devices, but adds a quirk by stacking both (i.e. the second is created inside the first). Obviously, stacking the encryption doubles overhead. The usecase here is simply to illustrate another example of the cipher option usage.

A first mapper is created with *cryptsetup's* plain-mode defaults, as described in the table's left column above

```
# cryptsetup --type plain -v open /dev/sdxY plain1
```

```
Enter passphrase:
Command successful.
```

Now we add the second block device inside it, using different encryption parameters and with an (optional) offset, create a file system and mount it

```
# cryptsetup --type plain --cipher=serpent-xts-plain64 --hash=sha256 --key-size=256 --offset=10  open /de
v/mapper/plain1 plain2
```

```
Enter passphrase:
```

```
# lsblk -p
```

```
 NAME
 /dev/sda
 ├─/dev/sdxY
 │ └─/dev/mapper/plain1
 │   └─/dev/mapper/plain2
 ...
```

```
# mkfs -t ext2 /dev/mapper/plain2
# mount -t ext2 /dev/mapper/plain2 /mnt
# echo "This is stacked. one passphrase per foot to shoot." > /mnt/stacked.txt
```

We close the stack to check access works

```
# cryptsetup close plain2
# cryptsetup close plain1
```

First, let us try to open the file system directly:

```
# cryptsetup --type plain --cipher=serpent-xts-plain64 --hash=sha256 --key-size=256 --offset=10 open /de
v/sdxY plain2
```

```
# mount -t ext2 /dev/mapper/plain2 /mnt
```

```
mount: wrong fs type, bad option, bad superblock on /dev/mapper/plain2,
       missing codepage or helper program, or other error
```

Why that did not work? Because the "plain2" starting block (`10`) is still encrypted with the cipher from "plain1". It can only be accessed via the stacked mapper. The error is arbitrary though, trying a wrong passphrase or wrong options will yield the same. For *dm-crypt* plain mode, the `open` action will not error out itself.

Trying again in correct order:

```
# cryptsetup close plain2    # dysfunctional mapper from previous try
```

```
# cryptsetup --type plain open /dev/sdxY plain1
```

```
Enter passphrase:
```

```
# cryptsetup --type plain --cipher=serpent-xts-plain64 --hash=sha256 --key-size=256 --offset=10 open /de
v/mapper/plain1 plain2
```

```
Enter passphrase:
```

```
# mount /dev/mapper/plain2 /mnt && cat /mnt/stacked.txt
```

```
This is stacked. one passphrase per foot to shoot.
```

dm-crypt will handle stacked encryption with some mixed modes too. For example LUKS mode could be stacked on the "plain1" mapper. Its header would then be encrypted inside "plain1" when that is closed.

Available for plain mode only is the option `--shared`. With it a single device can be segmented into different non-overlapping mappers. We do that in the next example, using a *loopaes* compatible cipher mode for "plain2" this time:

```
# cryptsetup --type plain --offset 0 --size 1000 open /dev/sdxY plain1
```

```
Enter passphrase:
```

```
# cryptsetup --type plain --offset 1000 --size 1000 --shared --cipher=aes-cbc-lmk --hash=sha256 open /de
v/sdxY plain2
```

```
Enter passphrase:
```

```
# lsblk -p
```

```
NAME
dev/sdxY
├─/dev/sdxY
│ ├─/dev/mapper/plain1
│ └─/dev/mapper/plain2
...
```

As the device tree shows both reside on the same level, i.e. are not stacked and "plain2" can be opened individually.

5 Cryptsetup actions specific for LUKS

5.1 Key management

It is possible to define additional keys for the LUKS partition. This enables the user to create access keys for safe backup storage In so-called key escrow, one key is used for daily usage, another kept in escrow to gain access to the partition in case the daily passphrase is forgotten or a keyfile is lost/damaged. A different key-slot could also be used to grant access to a partition to a user by issuing a second key and later revoking it again.

Once an encrypted partition has been created, the initial keyslot 0 is created (if no other was specified manually). Additional keyslots are numbered from 1 to 7. Which keyslots are used can be seen by issuing

```
# cryptsetup luksDump /dev/device
```

Where *device* is the block device containing the LUKS header. This and all the following commands in this section work on header backup files as well.

5.1.1 Adding LUKS keys

Adding new keyslots is accomplished with the `luksAddKey` action. For safety it will always, even for already unlocked devices, ask for a valid existing key (a passphrase for any existing slot) before a new one may be entered:

```
# cryptsetup luksAddKey /dev/device [/path/to/additionalkeyfile]
```

```
Enter any passphrase:
Enter new passphrase for key slot:
Verify passphrase:
```

If */path/to/additionalkeyfile* is given, cryptsetup will add a new keyslot for *additionalkeyfile*. Otherwise it prompts for a new passphrase. To authorize the action with an existing *keyfile*, the `--key-file` or `-d` option followed by the "old" *keyfile* will try to unlock all available keyfile keyslots:

```
# cryptsetup luksAddKey /dev/device [/path/to/additionalkeyfile] -d /path/to/keyfile
```

If it is intended to use multiple keys and change or revoke them, the `--key-slot` or `-S` option may be used to specify the slot:

```
# cryptsetup luksAddKey /dev/device -S 6
```

```
Enter any passphrase:
Enter new passphrase for key slot:
Verify passphrase:
```

```
# cryptsetup luksDump /dev/sda8 | grep 'Slot 6'
```

```
Key Slot 6: ENABLED
```

To show an associated action in this example, we decide to change the key right away:

```
# cryptsetup luksChangeKey /dev/device -S 6
```

```
Enter LUKS passphrase to be changed:
Enter new LUKS passphrase:
```

before continuing to remove it.

5.1.2 Removing LUKS keys

There are three different actions to remove keys from the header:

- `luksRemoveKey` removes a key by specifying its passphrase/key-file.
- `luksKillSlot` removes a key by specifying its slot (needs another valid key). Obviously, this is extremely useful if you have forgotten a passphrase, lost a key-file, or have no access to it.
- `luksErase` removes **all** active keys.

Warning:

- All above actions can be used to irrevocably delete the last active key for an encrypted device!
- The `luksErase` command was added in version 1.6.4 to quickly nuke access to the device. This action **will not** prompt for a valid passphrase! It will not **wipe the LUKS header**, but all keyslots at once and you will, therefore, not be able to regain access unless you have a valid backup of the LUKS header.

For above warning it is good to know the key we want to **keep** is valid. An easy check is to unlock the device with the `-v` option, which will specify which slot it occupies:

```
# cryptsetup --test-passphrase -v open /dev/device
```

```
Enter passphrase for /dev/device:
Key slot 1 unlocked.
Command successful.
```

Now we can remove the key added in the previous subsection using its passphrase:

```
# cryptsetup luksRemoveKey /dev/device
```

```
Enter LUKS passphrase to be deleted:
```

If we had used the same passphrase for two keyslots, the first slot would be wiped now. Only executing it again would remove the second one.

Alternatively, we can specify the key slot:

```
# cryptsetup luksKillSlot /dev/device 6
```

```
Enter any remaining LUKS passphrase:
```

Note that in both cases, no confirmation was required.

```
# cryptsetup luksDump /dev/sda8 | grep 'Slot 6'
```

```
Key Slot 6: DISABLED
```

To re-iterate the warning above: If the same passphrase had been used for key slots 1 and 6, both would be gone now.

5.2 Backup and restore

If the header of a LUKS encrypted partition gets destroyed, you will not be able to decrypt your data. It is just as much of a dilemma as forgetting the passphrase or damaging a key-file used to unlock the partition. Damage may occur by your own fault while re-partitioning the disk later or by third-party programs misinterpreting the partition table. Therefore, having a backup of the header and storing it on another disk might be a good idea.

Note: If one of the LUKS-encrypted partitions' passphrases becomes compromised, you must revoke it on *every* copy of the cryptheader, even those you have backed up. Otherwise, a copy of the backed-up cryptheader that uses the compromised passphrase can be used to determine the master key which in turn can be used to decrypt the associated partition (even your actual partition, not only the backed-up version). On the other hand, if the master key gets compromised, you have to reencrypt your whole partition. See **LUKS FAQ (https://gitlab.com/cryptsetup/cryptsetup/wikis/FrequentlyAskedQuestions#6-backup-and-data-recovery)** for further details.

5.2.1 Backup using cryptsetup

Cryptsetup's `luksHeaderBackup` action stores a binary backup of the LUKS header and keyslot area:

```
# cryptsetup luksHeaderBackup /dev/device --header-backup-file /mnt/backup/file.img
```

where *device* is the partition containing the LUKS volume.

You can also back up the plain text header into ramfs and encrypt it with e.g. **GPG** before writing it to persistent storage:

```
# mount --mkdir -t ramfs ramfs /root/tmp
# cryptsetup luksHeaderBackup /dev/device --header-backup-file /root/tmp/file.img
# gpg2 --recipient User_ID --encrypt /root/tmp/file.img
# cp /root/tmp/file.img.gpg /mnt/backup/
# umount /root/tmp
```

Warning: **tmpfs** can swap to the disk in low memory situations, so it is not recommended here.

5.2.2 Restore using cryptsetup

Warning: Restoring the wrong header or restoring to an unencrypted partition will cause data loss! The action can not perform a check whether the header is actually the *correct* one for that particular device.

In order to evade restoring a wrong header, you can ensure it does work by using it as a remote `--header` first:

```
# cryptsetup -v --header /mnt/backup/file.img open /dev/device test
```

```
Key slot 0 unlocked.
Command successful.
```

```
# mount /dev/mapper/test /mnt/test && ls /mnt/test
# umount /mnt/test
# cryptsetup close test
```

Now that the check succeeded, the restore may be performed:

```
# cryptsetup luksHeaderRestore /dev/device --header-backup-file ./mnt/backup/file.img
```

Now that all the keyslot areas are overwritten; only active keyslots from the backup file are available after issuing the command.

5.2.3 Manual backup and restore

The header always resides at the beginning of the device and a backup can be performed without access to *cryptsetup* as well. First you have to find out the payload offset of the crypted partition:

```
# cryptsetup luksDump /dev/device | grep "Payload offset"
```

```
Payload offset: 4040
```

Second check the sector size of the drive

```
# fdisk -l /dev/device | grep "Sector size"
```

```
Sector size (logical/physical): 512 bytes / 512 bytes
```

Now that you know the values, you can backup the header with a simple **dd** command:

```
# dd if=/dev/device of=/path/to/file.img bs=512 count=4040
```

and store it safely.

A restore can then be performed using the same values as when backing up:

```
# dd if=./file.img of=/dev/device bs=512 count=4040
```

5.3 Re-encrypting devices

The **cryptsetup (https://archlinux.org/packages/?name=cryptsetup)** package features two options for re-encryption.

cryptsetup reencrypt
: Argument to `cryptsetup` itself: Preferred method. Currently LUKS2 devices only. Actions can be performed online. Supports multiple parallel re-encryption jobs. Resilient to system failures. See **cryptsetup(8) (https://man.archlinux.org/man/cryptsetup.8)** for more information.

cryptsetup-reencrypt
: Legacy tool, supports LUKS1 in addition to LUKS2. Actions can be performed on unmounted devices only. Single process at a time. Sensitive to system failures. See **cryptsetup-reencrypt(8) (https://man.archlinux.org/man/cryptsetup-reencrypt.8)** for more information.

Both can be used to convert an existing unencrypted file system to a LUKS encrypted one or permanently remove LUKS encryption from a device (using `--decrypt`). As its name suggests it can also be used to re-encrypt an existing LUKS encrypted device, though, re-encryption is not possible for a detached LUKS header or other encryption modes (e.g. plain-mode). For re-encryption it is possible to change the **#Encryption options for LUKS mode**.

One application of re-encryption may be to secure the data again after a passphrase or **keyfile** has been compromised *and* one cannot be certain that no copy of the LUKS header has been obtained. For example, if only a passphrase has been shoulder-surfed but no physical/logical access to the device happened, it would be enough to change the respective passphrase/key only (**#Key management**).

Warning: Always make sure a **reliable backup** is available and double-check options you specify before using the tool!

The following shows an example to encrypt an unencrypted file system partition and a re-encryption of an existing LUKS device.

5.3.1 Encrypt an existing unencrypted file system

Tip: If you are trying to encrypt an existing root partition, you might want to create a separate and unencrypted boot partition which will be mounted to `/boot` (see **Dm-crypt/Encrypting an entire system#Preparing the boot partition**). It is not strictly necessary but has a number of advantages:

- If `/boot` is located inside an encrypted root partition, the system will ask for the passphrase twice when the machine is powered on. The first time will happen when the boot loader attempts to read the files located inside encrypted `/boot`, the second time will be when the kernel tries to mount the encrypted partition **[5] (https://opencraft.com/blog/tutorial-encrypting-an-existing-root-partition-in-ubuntu-with-dm-crypt-and-luks/)**. This might not be the desired behaviour and can be prevented by having a separate and unencrypted boot partition.
- Some system restore applications (e.g., **`timeshift` (https://archlinux.org/packages/?name=timeshift)**) will not work if `/boot` is located inside an encrypted partition **[6] (https://github.com/teejee2008/timeshift/issues/280)**.

In short, create a partition with the size of at least 260 MiB if needed. See **Partitioning#/boot**.

A LUKS encryption header is always stored at the beginning of the device. Since an existing file system will usually be allocated all partition sectors, the first step is to shrink it to make space for the LUKS header.

The **default** LUKS2 header requires 16 MiB. If the current file system occupies all the available space, we will have to shrink it at least that much. To shrink an existing `ext4` file system on `/dev/sdxY` to its current possible minimum:

```
# umount /mnt
```

```
# e2fsck -f /dev/sdxY
```

```
e2fsck 1.46.5 (30-Dec-2021)
Pass 1: Checking inodes, blocks, and sizes
...
/dev/sda6: 12/166320 files (0.0% non-contiguous), 28783/665062 blocks
```

```
# resize2fs -p -M /dev/sdxY
```

```
e2fsck 1.46.5 (30-Dec-2021)
Resizing the filesystem on /dev/sdxY to 26347 (4k) blocks.
The filesystem on /dev/sdxY is now 26347 (4k) blocks long.
```

Tip: Shrinking to the minimum size with `-M` might take a long time. You might want to calculate a size just 32 MiB smaller than the current size instead of using `-M`.

Warning: The file system should be shrunk while the underlying device (e.g., a partition) should be kept at its original size. Some graphical tools (e.g., **GParted**) may resize both the file system and the partition, and data loss may occur after encryption.

Now we encrypt it, using the default cipher we do not have to specify it explicitly:

```
# cryptsetup reencrypt --encrypt --reduce-device-size 16M /dev/sdxY
```

```
WARNING!

========

This will overwrite data on LUKS2-temp-12345678-9012-3456-7890-123456789012.new irrevocably.

Are you sure? (Type 'yes' in capital letters): YES
Enter passphrase for LUKS2-temp-12345678-9012-3456-7890-123456789012.new:
Verify passphrase:
```

After it finished, the whole `/dev/sdxY` partition is encrypted, not only the space the file system was shrunk to. As a final step we extend the original `ext4` file system to occupy all available space again, on the now encrypted partition:

```
# cryptsetup open /dev/sdxY recrypt
```

```
Enter passphrase for /dev/sdxY:
...
```

```
# resize2fs /dev/mapper/recrypt
```

```
resize2fs 1.43-WIP (18-May-2015)
Resizing the filesystem on /dev/mapper/recrypt to 664807 (4k) blocks.
The filesystem on /dev/mapper/recrypt is now 664807 (4k) blocks long.
```

```
# mount /dev/mapper/recrypt /mnt
```

The file system is now ready to use. You may want to add it to your **crypttab**.

Tip: If you have just encrypted your root partition, you might need to perform a number of post-encryption adjustments.

1. Configure mkinitcpio and kernel parameters. See **dm-crypt/System configuration#Unlocking in early userspace**.
2. Update the entry for `/` in **fstab** to use the unlocked volume's specifier (e.g. **UUID**).

5.3.2 Re-encrypting an existing LUKS partition

In this example an existing LUKS device is re-encrypted.

Warning: Double-check you specify encryption options for correctly and *never* re-encrypt without a **reliable backup**!

In order to re-encrypt a device with its existing encryption options, they do not need to be specified:

```
# cryptsetup reencrypt /dev/sdxY
```

Note: For LUKS1 we will need to use the legacy tool:

```
# cryptsetup-reencrypt /dev/sdxY
```

Existing keys are retained when re-encrypting a device with a different cipher and/or hash. Another use case is to re-encrypt LUKS devices which have non-current encryption options. Apart from above warning on specifying options correctly, the ability to change the LUKS header may also be limited by its size. For example, if the device was initially encrypted using a CBC mode cipher and 128 bit key-size, the LUKS header will be half the size of above mentioned `4096` sectors:

```
# cryptsetup luksDump /dev/sdxY | grep -e "mode" -e "Payload" -e "MK bits"
```

```
Cipher mode:    cbc-essiv:sha256
Payload offset: 2048
MK bits:        128
```

While it is possible to upgrade the encryption of such a device, it is currently only feasible in two steps. First, re-encrypting with the same encryption options, but using the `--reduce-device-size` option to make further space for the larger LUKS header. Second, re-encypt the whole device again with the desired cipher. For this reason and the fact that a backup should be created in any case, creating a new, fresh encrypted device to restore into is always the faster option.

5.4 Conversion from LUKS1 to LUKS2 and back

The **cryptsetup (https://archlinux.org/packages/?name=cryptsetup)** package has `convert` option that needed for conversion between LUKS1 and LUKS2 container types. The argument `--type` is **required**.

Migration from LUKS1 to LUKS2:

```
# cryptsetup convert --type luks2 /dev/sdxY
```

Note: The LUKS header size will be 16 MiB instead of 2 MiB.

Rollback to LUKS1 (for example, to boot from **GRUB with encrypted /boot**):

```
# cryptsetup convert --type luks1 /dev/sdxY
```

Note: Conversion from LUKS2 to LUKS1 is **not** always possible. You may get the following error:

```
Cannot convert to LUKS1 format - keyslot 0 is not LUKS1 compatible.
```

If the container is using Argon2, it needs to be converted to PBKDF2 to be LUKS1-compatible.

```
# cryptsetup luksConvertKey --pbkdf pbkdf2 /dev/sdXY
```

6 Resizing encrypted devices

If a storage device encrypted with dm-crypt is being cloned (with a tool like dd) to another larger device, the underlying dm-crypt device must be resized to use the whole space.

The destination device is /dev/sdX2 in this example, the whole available space adjacent to the partition will be used:

```
# cryptsetup luksOpen /dev/sdX2 sdX2
# cryptsetup resize sdX2
```

Then the underlying file system must be resized.

6.1 Loopback file system

Assume that an encrypted loopback file system is stored in a file `/bigsecret`, looped to `/dev/loop0`, mapped to `secret` and mounted on `/mnt/secret`, as in the example at **dm-crypt/Encrypting a non-root file system#File container**.

If the container file is currently mapped and/or mounted, unmount and/or close it:

```
# umount /mnt/secret
# cryptsetup close secret
# losetup -d /dev/loop0
```

Next, expand the container file with the size of the data you want to add. In this example, the file will be expanded with 1M * 1024, which is 1G.

Warning: Make absolutely sure to use **two** `>`, instead of just one, or else you will overwrite the file instead of appending to it. Making a backup before this step is strongly recommended.

```
# dd if=/dev/urandom bs=1M count=1024 | cat - >> /bigsecret
```

Now map the container to the loop device:

```
# losetup /dev/loop0 /bigsecret
# cryptsetup open /dev/loop0 secret
```

After this, resize the encrypted part of the container to the new maximum size of the container file:

```
# cryptsetup resize secret
```

Finally, perform a file system check and, if it is ok, resize it (example for ext2/3/4):

```
# e2fsck -f /dev/mapper/secret
# resize2fs /dev/mapper/secret
```

You can now mount the container again:

```
# mount /dev/mapper/secret /mnt/secret
```

6.2 Integrity protected device

If the device was formatted with integrity support (e.g., `--integrity hmac-sha256`) and the backing block device is shrinked, it cannot be opened with this error: `device-mapper: reload ioctl on failed: Invalid argument`.

To fix this issue without wiping the device again, it can be formatted with the previous master key (keeping the per-sector tags valid).

```
# cryptsetup luksDump /dev/sdX2 --dump-master-key --master-key-file=/tmp/masterkey-in-tmpfs.key
# cryptsetup luksFormat /dev/sdX2 --type luks2 --integrity hmac-sha256 --master-key-file=/tmp/masterkey-i
n-tmpfs.key --integrity-no-wipe
# rm /tmp/masterkey-in-tmpfs.key
```

7 Keyfiles

Note: This section describes using a plaintext keyfile. If you want to encrypt your keyfile giving you two factor authentication see **Using GPG or OpenSSL Encrypted Keyfiles** for details, but please still read this section.

What is a keyfile?

A keyfile is a file whose data is used as the passphrase to unlock an encrypted volume. That means if such a file is lost or changed, decrypting the volume may no longer be possible.

Tip: Define a passphrase in addition to the keyfile for backup access to encrypted volumes in the event the defined keyfile is lost or changed.

Why use a keyfile?

There are many kinds of keyfiles. Each type of keyfile used has benefits and disadvantages summarized below:

7.1 Types of keyfiles

7.1.1 passphrase

This is a keyfile containing a simple passphrase. The benefit of this type of keyfile is that if the file is lost the data it contained is known and hopefully easily remembered by the owner of the encrypted volume. However the disadvantage is that this does not add any security over entering a passphrase during the initial system start.

Example: `1234`

Note: The keyfile containing the passphrase must not have a newline in it. One option is to create it using

```
# echo -n 'your_passphrase' > /path/to/keyfile
# chown root:root /path/to/keyfile; chmod 400 /path/to/keyfile
```

If the file contains special characters such as a backslash, rather than escaping these, it is recommended to simply edit the key file directly entering or pasting the passphrase and then remove the trailing newline with a handy perl one-liner:

```
# perl -pi -e 'chomp if eof' /path/to/keyfile
```

7.1.2 randomtext

This is a keyfile containing a block of random characters. The benefit of this type of keyfile is that it is much more resistant to dictionary attacks than a simple passphrase. An additional strength of keyfiles can be utilized in this situation which is the length of data used. Since this is not a string meant to be memorized by a person for entry, it is trivial to create files containing thousands of random characters as the key. The disadvantage is that if this file is lost or changed, it will most likely not be possible to access the encrypted volume without a backup passphrase.

Example: `fjqweifj830149-57 819y4my1-38t1934yt8-91m 34co3;t8y;9p3y-`

7.1.3 binary

This is a binary file that has been defined as a keyfile. When identifying files as candidates for a keyfile, it is recommended to choose files that are relatively static such as photos, music, video clips. The benefit of these files is that they serve a dual function which can make them harder to identify as keyfiles. Instead of having a text file with a large amount of random text, the keyfile would look like a regular image file or music clip to the casual observer. The disadvantage is that if this file is lost or changed, it will most likely not be possible to access the encrypted volume without a backup passphrase. Additionally, there is a theoretical loss of randomness when compared to a randomly generated text file. This is due to the fact that images, videos and music have some intrinsic relationship between neighboring bits of data that does not exist for a random text file. However this is controversial and has never been exploited publicly.

Example: images, text, video, ...

7.2 Creating a keyfile with random characters

7.2.1 Storing the keyfile on a file system

A keyfile can be of arbitrary content and size.

Here **dd** is used to generate a keyfile of 2048 random bytes, storing it in the file `/etc/mykeyfile`:

```
# dd bs=512 count=4 if=/dev/random of=/etc/mykeyfile iflag=fullblock
```

If you are planning to store the keyfile on an external device, you can also simply change the outputfile to the corresponding directory:

```
# dd bs=512 count=4 if=/dev/random of=/media/usbstick/mykeyfile iflag=fullblock
```

To deny any access for other users than `root` :

```
# chmod 600 /etc/mykeyfile
```

7.2.1.1 Securely overwriting stored keyfiles

If you stored your temporary keyfile on a physical storage device, and want to delete it, remember to not just remove the keyfile later on, but use something like

```
# shred --remove --zero mykeyfile
```

to securely overwrite it. For overaged file systems like FAT or ext2 this will suffice while in the case of journaling file systems, flash memory hardware and other cases it is highly recommended to **wipe the entire device**.

7.2.2 Storing the keyfile in ramfs

Alternatively, you can mount a ramfs for storing the keyfile temporarily:

```
# mount --mkdir -t ramfs ramfs /root/myramfs
# cd /root/myramfs
```

The advantage is that it resides in RAM and not on a physical disk, therefore it can not be recovered after unmounting the ramfs. After copying the keyfile to another secure and persistent file system, unmount the ramfs again with

```
# umount /root/myramfs
```

7.3 Configuring LUKS to make use of the keyfile

Add a keyslot for the keyfile to the LUKS header:

```
# cryptsetup luksAddKey /dev/sda2 /etc/mykeyfile
```

```
Enter any LUKS passphrase:
key slot 0 unlocked.
Command successful.
```

7.4 Manually unlocking a partition using a keyfile

Use the `--key-file` option when opening the LUKS device:

```
# cryptsetup open /dev/sda2 dm_name --key-file /etc/mykeyfile
```

7.5 Unlocking the root partition at boot

This is simply a matter of configuring **mkinitcpio** to include the necessary modules or files and configuring the **cryptkey kernel parameter** to know where to find the keyfile.

Two cases are covered below:

1. Using a keyfile stored on an external medium (e.g. a USB stick)
2. Using a keyfile embedded in the initramfs

7.5.1 With a keyfile stored on an external media

7.5.1.1 Configuring mkinitcpio

You have to add the kernel module for the drive's **file system** to the **MODULES array** in `/etc/mkinitcpio.conf`. For example, add `ext4` if the file system is **Ext4** or `vfat` in case it is **FAT**:

```
MODULES=(vfat)
```

If there are messages about bad superblock and bad codepage at boot, then you need an extra codepage module to be loaded. For instance, you may need `nls_iso8859-1` module for `iso8859-1` codepage.

Regenerate the initramfs.

7.5.1.2 Configuring the kernel parameters

- For a busybox-based initramfs using the **encrypt** hook, see **dm-crypt/System configuration#cryptkey**.
- For a systemd based initramfs using the **sd-encrypt** hook, see **dm-crypt/System configuration#rd.luks.key**.

7.5.2 With a keyfile embedded in the initramfs

Warning: Use an embedded keyfile **only** if you protect the keyfile sufficiently by:

- Using some form of authentication earlier in the boot process. Otherwise auto-decryption will occur, defeating completely the purpose of block device encryption.
- `/boot` is encrypted. Otherwise root on a different installation (including the **live environment**) can extract your key from the initramfs, and unlock the device without any other authentication.

This method allows to use a specially named keyfile that will be embedded in the **initramfs** and picked up by the `encrypt` **hook** to unlock the root file system (`cryptdevice`) automatically. It may be useful to apply when using the **GRUB early cryptodisk** feature, in order to avoid entering two passphrases during boot.

The `encrypt` hook lets the user specify a keyfile with the `cryptkey` kernel parameter: in the case of initramfs, the syntax is `rootfs:/path/to/keyfile`. See **dm-crypt/System configuration#cryptkey**. Besides, this kernel parameter defaults to use `/crypto_keyfile.bin`, and if the initramfs contains a valid key with this name, decryption will occur automatically without the need to configure the `cryptkey` parameter.

If using `sd-encrypt` instead of `encrypt`, specify the location of the keyfile with the `rd.luks.key` kernel parameter: in the case of initramfs, the syntax is */path/to/keyfile*. See **dm-crypt/System configuration#rd.luks.key**. This kernel parameter defaults to using `/etc/cryptsetup-keys.d/`*name*`.key` (where *name* is the *dm_name* used for decryption in **#Encrypting devices with cryptsetup**) and can be omitted if initramfs contains a valid key with this path.

Generate the keyfile, give it suitable permissions and **add it as a LUKS key**:

```
# dd bs=512 count=4 if=/dev/random of=/crypto_keyfile.bin iflag=fullblock
# chmod 600 /crypto_keyfile.bin
# cryptsetup luksAddKey /dev/sdX# /crypto_keyfile.bin
```

Note: The initramfs is generated by mkinitcpio with `600` **permissions** by default, so regular users are not able to read the keyfile via the generated initramfs.

Include the key in **mkinitcpio's FILES array**:

/etc/mkinitcpio.conf

```
FILES=(/crypto_keyfile.bin)
```

Finally **regenerate the initramfs**.

On the next reboot you should only have to enter your container decryption passphrase once.

(**source (https://www.pavelkogan.com/2014/05/23/luks-full-disk-encryption/#bonus-login-once)**)

-