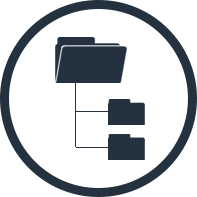
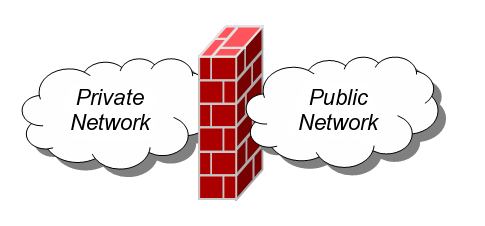
Secure Embedded System (SeS)

SSHD FileSystem Firewall

Auteur : Spinelli Isaia

Prof : [Schuler Jean-Roland](https://mse.hes-so.ch/consultation/master/horairesMSE.php?annee=2020&id=81&type=1)

Date : 26.10.2020

Salle : A4 – Lausanne

Classe : SeS

Table des matières

[Introduction - 2 -](#_Toc56434705)

[SSHD - 2 -](#_Toc56434706)

[Introduction - 2 -](#_Toc56434707)

[Installer et vérifier la signature d’OpenSSH - 2 -](#_Toc56434708)

[Configurer et installer sur intel - 4 -](#_Toc56434709)

[Configurer et installer sur Nano Pi - 6 -](#_Toc56434710)

[Création de clefs - 7 -](#_Toc56434711)

[Configuration sshd - 7 -](#_Toc56434712)

[Test final - 8 -](#_Toc56434713)

[Modification des sources - 9 -](#_Toc56434714)

[Conclusion SSHD - 11 -](#_Toc56434715)

[File system - 12 -](#_Toc56434716)

[Introduction - 12 -](#_Toc56434717)

[Kernel and rootfs configuration - 12 -](#_Toc56434718)

[EXT4 - 12 -](#_Toc56434719)

[Comment le noyau sait que le rootfs est sur la deuxième partition de la carte SD ? - 12 -](#_Toc56434720)

[Comment monter la première partition de la carte SD sur /mtn ? - 13 -](#_Toc56434721)

[Quel est le numéro majeur et mineur du fichier nœud gérant la carte SD ? - 13 -](#_Toc56434722)

[btrfs, f2fs, nilfs2, xfs - 13 -](#_Toc56434723)

[Créer deux nouvelles partitions (3 et 4) - 13 -](#_Toc56434724)

[Formater ces deux nouvelles partitions avec deux différents systèmes de fichier - 14 -](#_Toc56434725)

[Comparer les temps d’écriture entre les différents systèmes de fichier - 14 -](#_Toc56434726)

[LUKS, cryptsetup et dmcrypt - 15 -](#_Toc56434727)

[Créer une troisième partition LUKS - 15 -](#_Toc56434728)

[LUKS, cryptsetup options - 15 -](#_Toc56434729)

[LUKS test 1 - 15 -](#_Toc56434730)

[rootfs dans une partition LUKS - 18 -](#_Toc56434731)

[Initramfs - 21 -](#_Toc56434732)

[Conclusion - 25 -](#_Toc56434733)

[Annexes - 25 -](#_Toc56434734)

# Introduction

Ce rapport est composé de trois laboratoires distincts :

1. SSHD
2. FileSystem
3. Firewall

# SSHD

## Introduction

Durant ce laboratoire nous allons principalement nous familiariser avec le protocole SSH et les utilitaires OpenSSH. SSH est un protocole de connexion qui impose un échange de clés de chiffrement en début de connexion. Le port utilisé par défaut est le 22. OpenSSH est un ensemble d’outils open source utilisant le protocole SSH.

Durant ce laboratoire, plusieurs étapes nous a été demandé :

1. Installer et vérifier la signature d’OpenSSH
2. Configurer les packages sur notre machine Intel.
3. Configurer les packages sur la cible Nani Pi
4. Installer les différents utilitaires sur la cible.
5. Configurer des options de SSHD.
6. Modifier SSHD afin de ne plus indiquer la version utilisée.

## Installer et vérifier la signature d’OpenSSH

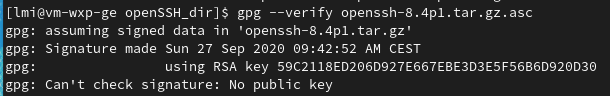
Afin d’installer OpenSSH il a fallu se rendre sur son site officiel (<https://www.openssh.com/>) et récupérer le lien http Suisse afin d’installer la dernière version :

*Wget https://mirror.ungleich.ch/pub/OpenBSD/OpenSSH/portable/*[*openssh-8.4p1.tar.gz*](https://mirror.ungleich.ch/pub/OpenBSD/OpenSSH/portable/openssh-8.4p1.tar.gz)

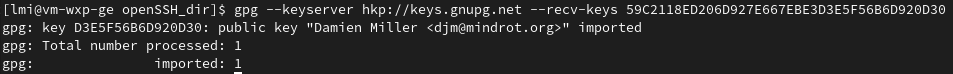
Dans le même temps, on peut télécharger la signature de celui-ci :

*Wget https://mirror.ungleich.ch/pub/OpenBSD/OpenSSH/portable/*[*openssh-8.4p1.tar.gz.asc*](https://mirror.ungleich.ch/pub/OpenBSD/OpenSSH/portable/openssh-8.4p1.tar.gz.asc)

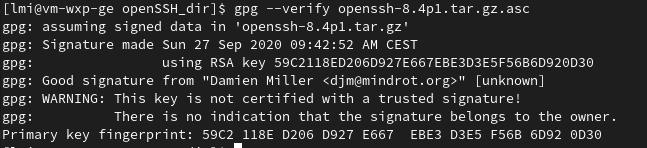
Pour commencer, il est important de vérifier l’authenticité et l’intégrité du logiciel :



On peut voir que le check n’est pas possible car la clef public n’est pas présente. Il faut importer cette clef sur notre PC comme suit :



On peut voir que la clef public de Damien Miller a correctement été importée. Il est maintenant possible de vérifier la signature du package :



## Configurer et installer sur intel

Maintenant que nous avons téléchargé OpenSSH, nous pouvons le configurer et l’installer sur notre machine hôte. Voici la commande à exécuter :

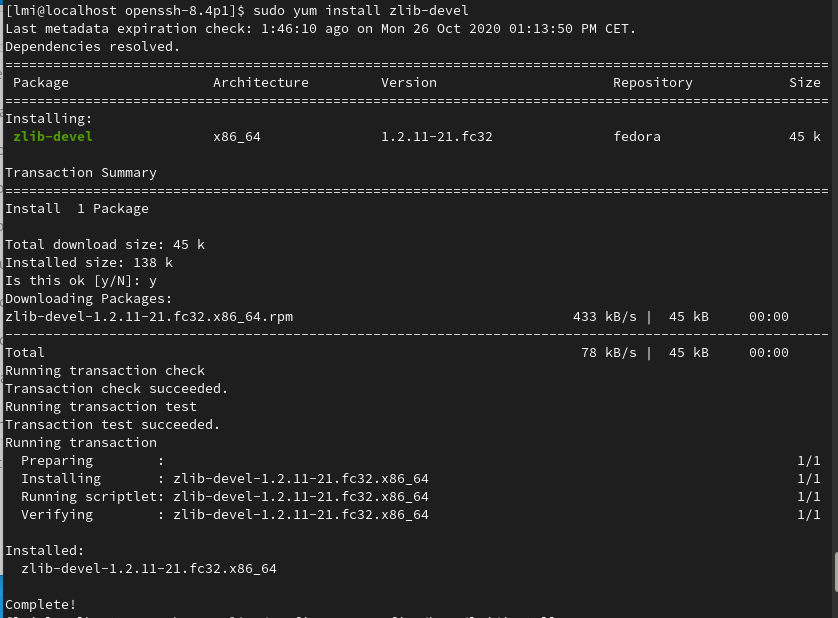
*./configure --prefix=/home/lmi/install*

L’option « –prefix » permet de modifier le répertoire d’installation.

Personnellement, j’ai eu l’erreur suivante :



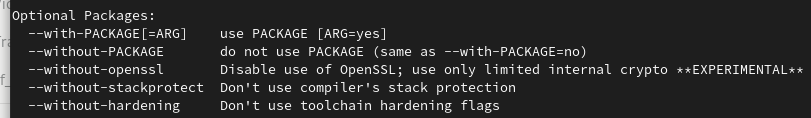
La librairie « zlib » n’est pas présente. Donc, je l’ai installée comme suit :



OpenSSH permet énormément de configuration et de différentes options, pour en savoir plus, il est intéressant de lire les fichiers « README » et « INSTALL ». Il est aussi possible d’utiliser la commande suivante :



Il nous est demandé d’activer l’option « hardening », cependant, grâce à la commande précédente, on peut apprendre que cette option est par défaut activée. Il est possible de la désactiver avec l’option « --without-hardening ».



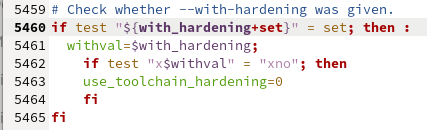
En m’intéressant aux différentes options proposées, j’ai pu voir celle-ci :



Il aurait été possible d’ajouter cette option dès le début afin de ne pas dépendre de la libraire « zlib ».

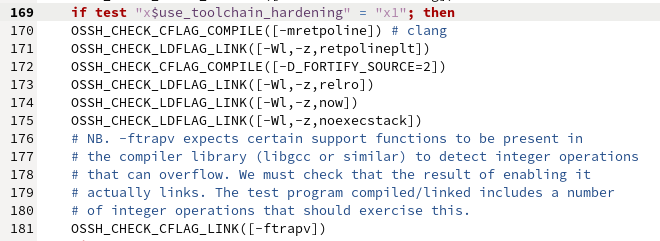
Afin de mieux comprendre l’impact de l’option « hardening », j’ai effectué une recherche de cette option et de son influence. Voici ce que j’ai pu trouver :

1. Dans le fichier « configure » :

7

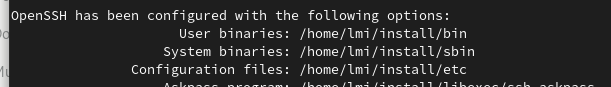
On peut voir que l’option « hardening » activera, ou pas, la variable « use\_toolchain\_hardening ».

1. Dans le fichier « configure.ac » :



Dans le cas où cette même variable est active, cela va ajouter plusieurs options de compilation de protection listées ci-dessus.

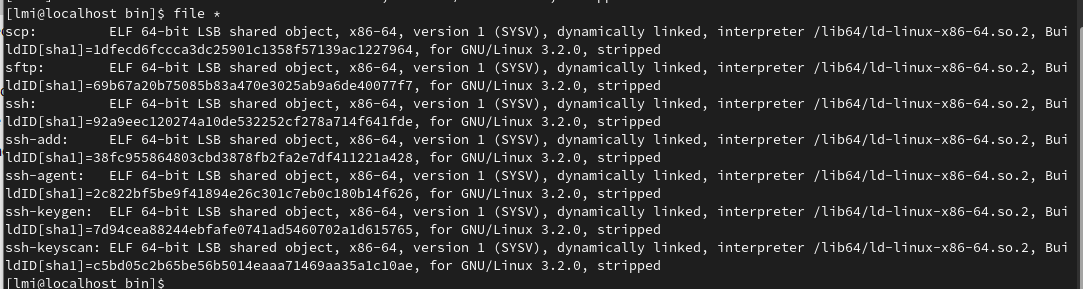
Une fois la configuration effectuée, un message résumant les informations de configuration est affiché. Voici une partie de ces informations :

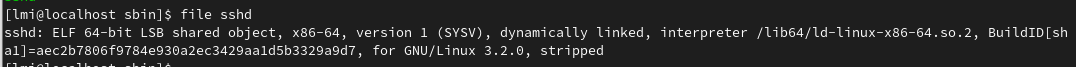


Ensuite, il est possible d’effectuer la compilation et l’installation avec les commandes suivantes :



Une fois tous les exécutables générés, il est possible de vérifier s’ils sont « stripped » :





On peut constater que tous les exécutables sont bien « stripped ».

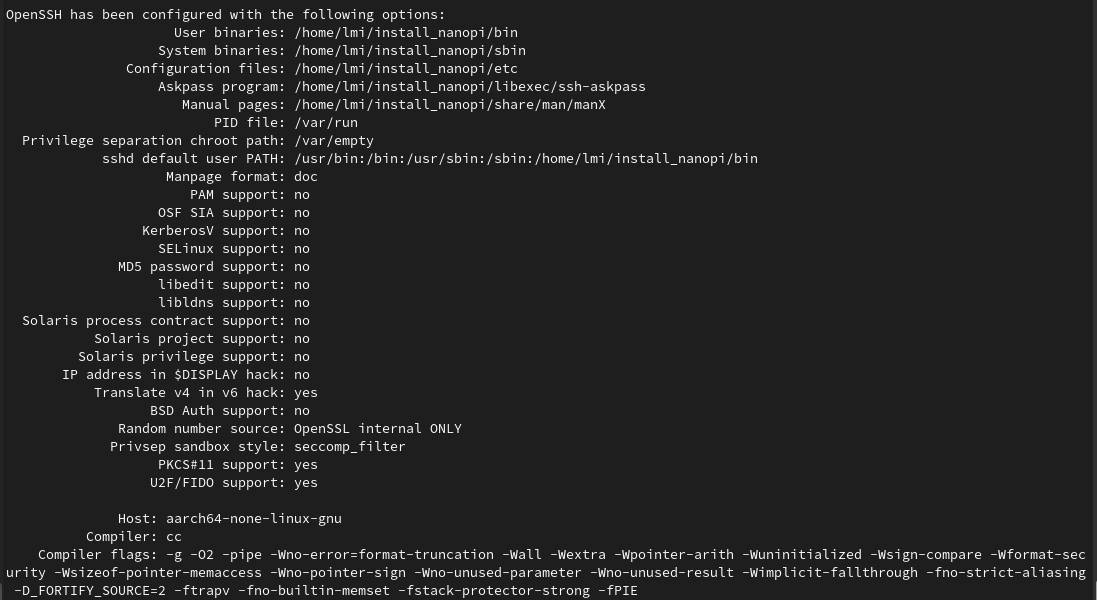
## Configurer et installer sur Nano Pi

L’objectif de cette étape est de configurer et installer OpenSSH sur la cible Nani Pi en utilisant la cross-compilation.

Étant donné cet utilitaire est relativement bien fait, il est facile de le cross-compiler. Il suffit d’ajouter l’option « --host » lors de la configuration des packages. On peut voir la commande à exécuter ci-dessous :

*./configure --host=aarch64-none-linux-gnu --prefix=/home/lmi/install\_nanopi*

Ci-dessous, la sortie de la configuration. On peut voir en bas, tous les flags de compilation activés.

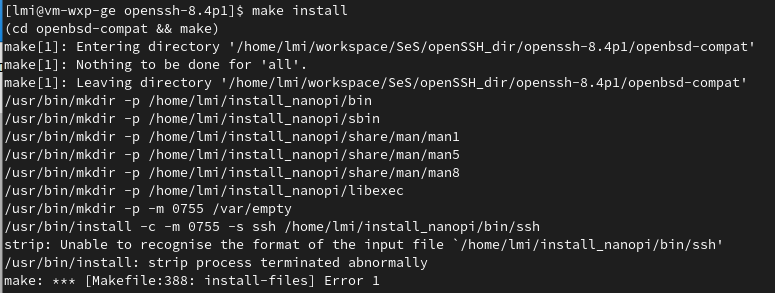


Ensuite, les deux commandes de compilation et d’installation peuvent être exécutées :

*make*

*make install*

Dans mon cas, la commande « make install » provoque une erreur :



La commande « strip » effectuée dans le script « /usr/bin/install » provoque une erreur. En faisant une recherche, voici ce que j’ai pu trouver :



La commande « strip » executée est celle par défaut avec un flag « -s ». Cependant, comme nous cross-compilons, il est nécessaire d’indiquer la commande « strip » adaptée à notre architecture grâce au flag « --strip-program » :



Après cela, l’installation a pu s’effectuer correctement.

Une fois que tous les fichiers exécutables sont correctement générés pour notre cible, il suffit de les transférer sur le système de fichier de la Nano PI. Il y a plusieurs manières de procéder, pour ma part, j’ai utilisé le système de fichier réseau (NFS) préalablement installé.

## Création de clefs

L’objectif est de créer différents types de clef sur la cible à l’aide de l’utilitaire « ssh-keygen » précédemment installé. Voici ci-dessous la liste des différents types de clef crées et les commandes permettant leur création :

1. rsa 4096 bits



1. dsa 1024 bits



1. ecdsa 521 bits



1. ed25519 256bits



## Configuration sshd

L’objectif est de créer un fichier permettant la configuration de SSHD. Celle-ci se fera via le fichier « sshd\_config ». Afin de répondre aux configurations demandées, voici les lignes à ajouter au fichier :

**Utilisation uniquement de IPv4 :**

AddressFamily inet

**Ne pas permettre la redirection de port :**

AllowTcpForwarding no

AllowStreamLocalForwarding no

GatewayPorts no

PermitTunnel no

**Autoriser des algorithmes de chiffrement de hachage :**

Ciphers aes128-ctr,aes192-ctr,aes256-ctr,aes256-cbc

MACs hmac-sha1,hmac-sha2-256

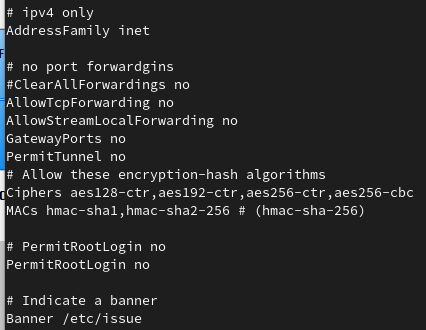
**Connexion root pas permis :**

PermitRootLogin no

**Indiquer une bannière :**

Banner /etc/issue

Voici plus précisément le fichier de configuration « sshd\_config ».



## Test final

L’objectif maintenant est de tester l’outil sshd sur la cile.

Afin de contrôler si un service ssh est en cours, on peut utiliser la commande « netstat -atu » afin de vérifier les programmes qui écoutent les ports. Ensuite, il faut désactiver le service ssh en cours par défaut avec la commande « /etc/init.d/S50dropbear stop ».

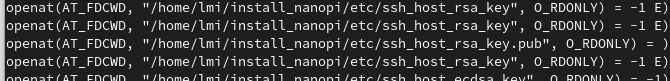
Afin de lancer le programme sshd précédemment configuré et installé, voici la commande à effectuer tout en précisant le fichier de configuration :



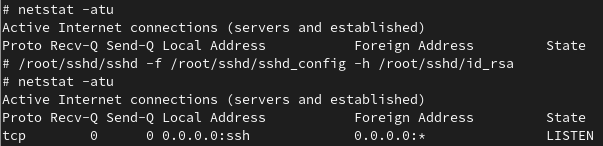
On peut remarquer qu’il nous indique qu’il ne dispose pas de clef hôte. Dans ce cas, il peut être intéressant d’utiliser la commande « strace » afin de mieux comprendre d’où vient le problème :



On peut voir ci-dessous que les clefs sont recherchées dans le dossier « /home/install\_nanopi/etc/ » :



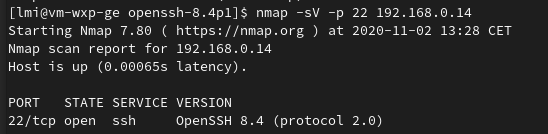
Il existe plusieurs façons d’indiquer l’emplacement des clefs. En analysant les paramètres du programme sshd, j’ai pu remarquer que l’option -h permet ceci. On peut voir ci-dessous la lancement de sshd en indiquant la clef rsa :



Grâce à la commande « netstat -atu », j’ai pu confirmer que le programme sshd c’était correctement lancé.

## Modification des sources

L’objectif de ce chapitre est de cacher la version de SSHD utilisée. En effet, il est possible à l’aide de la commande « nmap » d’avoir accès à cette information. On peut le voir ci-dessous :



Après plusieurs tentatives, je n’ai pas réussi. Cependant, j’ai fait des recherches sur internet et j’ai trouvé une page[[1]](#footnote-1) qui indique comment faire cette manipulation pour la version 4.3.

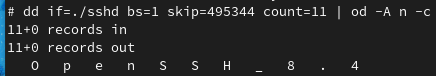
J’ai donc suivi ce tutoriel en adaptant quelques commandes. Voici les commandes exécutées pour y parvenir :

Premièrement, il faut trouver la string de la version utilisée dans le fichier binaire :



Il est possible d’afficher exactement ces caractères afin de s’assurer de l’offset et de la taille de ce string :

*dd if=./sshd bs=1 skip=495344 count=11 | od -A n -c*



L’objectif ensuite est de créer 3 fichiers contenant les informations du fichier binaire avant, pendant et après ce string :

*dd if=./sshd bs=1 count=495344 of=sshd.1 (Avant le string “OpenSSH\_8.4”)*

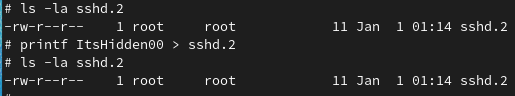
*dd if=./sshd bs=1 skip=495344 count=11 of=sshd.2 (le string “OpenSSH\_8.4”)*

*dd if=./sshd bs=1 skip=495355 count=999999999 of=sshd.3 (Après le string “OpenSSH\_8.4”)*

Nous pouvons contrôler que la découpe du string s’est bien réalisée :



Ensuite, il faut remplacer le texte de la version, tout en gardant le nombre de caractère :



Il est maintenant possible de recombiner les 3 fichiers afin de créer le nouveau binaire n’affichant plus la version sshd :

*cat sshd.\* > sshd.new*

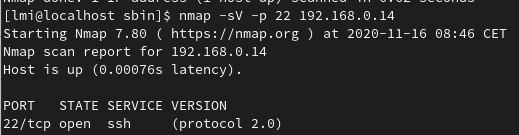
*chmod 755 ./sshd.new*

Ensuite, nous pouvons tester le nouveau binaire en l’exécutant et vérifiant si la version est encore affichée :

(Si un service ssh est déjà en cours, il est possible de le stopper avec les commandes « ps -e » & « kill -9 <PID> »)

*/root/sshd/sshd.new -f /root/sshd/sshd\_config -h /root/sshd/id\_rsa*

Finalement, on vérifie le bon fonctionnement de ces manipulations :



Nous constatons que la version n’est plus affichée.

## Conclusion SSHD

Ce laboratoire m’a permis de comprendre les principes et le fonctionnement des clefs et des signatures dans le domaine de la sécurité. De plus, il m’a fait prendre conscience que la cross-compilation pouvait rapidement être complexe. J’ai pu accroitre ma connaissance en ce qui concerne la configuration du protocole SSH et des packages en général. Finalement, j’ai pu découvrir quelques commandes pratiques tels que « strace » et « netstat ».

# File system

## Introduction

Durant ce laboratoire nous allons principalement nous familiariser avec ; les partitions, les différents systèmes de fichier, le cryptage de partition et l’utilisation d’un initramfs.

Ce laboratoire est composé de plusieurs étapes distinctes :

1. Kernel and rootfs configuration
2. EXT4
3. Btrfs, f2fs, nilfs2 et xfs
4. LUKS, cryptsetup et dmcrypt
5. initramfs
6. initramfs-LUKS partition

## Kernel and rootfs configuration

Ce chapitre permet de configurer le noyau et le rootfs grâce à l’interface menu-config. Plusieurs configurations ont été ajoutées :

1. mkfs.ext2 (busybox-menuconfig)
2. tune2fs (busybox-menuconfig)
3. Btrfs, NILFS2, F2FS, XFS (linux-menuconfig)
4. USB Mass Storage (linux-menuconfig)
5. Cryptsetup (buildroot et linux-menuconfig)
6. Initramfs (linux-menuconfig)

Toutes ces étapes de configurations sont détaillées dans la donnée du laboratoire en annexe [1].

Une fois que la nouvelle configuration est terminée, il faut générer et installer le nouveau kernel et rootfs sur la carte SD. Par exemple, en utilisant un script créé lors des laboratoires précédents.

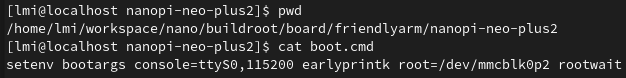
Typiquement, on peut vérifier que le package « cryptsetup » est disponible sur la cible NanoPi :



## EXT4

### Comment le noyau sait que le rootfs est sur la deuxième partition de la carte SD ?

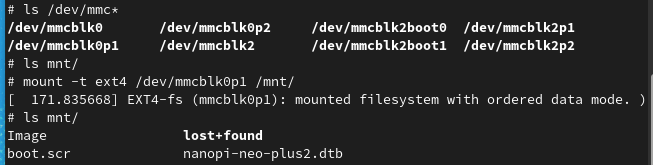
Car on l’indique dans les arguments avant de lancer le noyau avec la commande « setenv » dans le fichier « boot.cmd ». Ce fichier se trouve dans le répertoire associé à la board utilisée. On peut voir plus précisément cette ligne de commande ci-dessous :



« Mmcblk0 » inique la carte SD et « p2 » indique la deuxième partition.

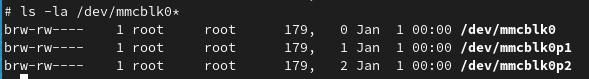
### Comment monter la première partition de la carte SD sur /mtn ?

On peut voir dans l’image ci-dessous comment monter la première partition de la carte sur un répertoire (/mnt) :



### Quel est le numéro majeur et mineur du fichier nœud gérant la carte SD ?

On peut voir ci-dessous trois différents nœuds qui sont en lien avec la carte SD :



La numéro majeur, indiquant le driver utilisé, est le 179. Ensuite, les numéro mineurs se suivent simplement de 0 à 2.

## btrfs, f2fs, nilfs2, xfs

### Créer deux nouvelles partitions (3 et 4)

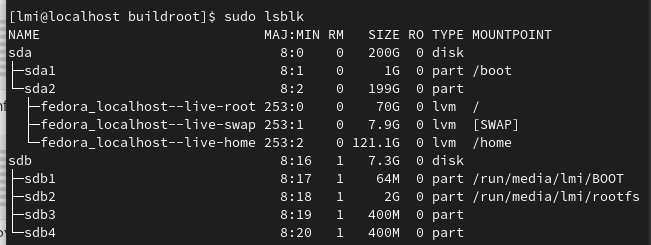
Voici les commandes que j’ai utilisées :

*sudo parted /dev/sdb mkpart primary ext4 4358144s 5177343s*

*sudo parted /dev/sdb mkpart primary ext4 5177344s 5996543s*

Le début en secteur (s) de la première partition créée est 4358144, qui est la suite (+1) de la fin de la partition précédant. Ensuite, il faut indiquer la fin de la partition en secteur. Afin de faire une partition de 400MB il faut ajouter 819 199 (Taille de la partition / Taille d’un secteur – 1 = 400MB / 512 - 1). Nous trouvons donc 5177343s. Le principe est le même pour la quatrième partition.

Il est de bonne habitude de vérifier le bon fonctionnement des commandes exécutées :



### Formater ces deux nouvelles partitions avec deux différents systèmes de fichier

Personnellement j’ai choisi les système de fichier btrfs et f2fs. Voici les commandes que j’ai dû exécuter pour formater la troisième partition en f2fs :

*sudo dnf install f2fs-tools.x86\_64*

*sudo mkfs.f2fs -l usr\_f2fs /dev/sdb3*

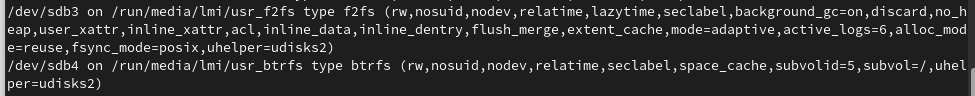
Voici celles pour formater la quatrième partition en btrfs :

*sudo dnf install btrfs-progs.x86\_64*

*sudo mkfs.btrfs /dev/sdb4*

*sudo btrfs filesystem label /dev/sdb4 usr\_btrfs*

Encore une fois, on peut contrôler le bon fonctionnement des commandes précédentes :



### Comparer les temps d’écriture entre les différents systèmes de fichier

Il peut arriver que le programme soit gêné par d’autres processus lors de la mesure. De ce fait, j’ai fait plusieurs mesures afin d’avoir une vision plus générale. Voici les différentes mesures observées lors des tests :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **btrfs** | | **ext4** | | **f2fs** | |
| small file | big file | small file | big file | small file | big file |
| **Mesures [ms]** | 65 | 36 | 13 | 8 | 16 | 6 |
| 58 | 10 | 18 | 10 | 14 | 6 |
| 54 | 527 | 17 | 10 | 19 | 8 |
| 67 | 13 | 14 | 6 | 12 | 9 |
| 102 | 1044 | 12 | 8 | 14 | 6 |
| 108 | 7 | 22 | 8 | 679 | 6 |
| 103 | 6 | 16 | 9 | 15 | 6 |
| 39 | 56 | 14 | 8 | 15 | 9 |
| 58 | 506 | 17 | 8 | 13 | 8 |
| 82 | 10 | 13 | 8 | 12 | 8 |
| 54 | 209 | 19 | 8 | 12 | 6 |
| 63 | 15 | 23 | 9 | 11 | 6 |
| **Moyenne** | 71.08 | 203.25 | 16.50 | 8.33 | 69.33 | 7.00 |
| **Médiane** | **64.00** | **25.50** | 16.50 | 8.00 | **14.00** | **6.00** |

On peut voir des mesures en rouge qui sont surement dû à des processus gênant lors de la mesure. Pour éviter de prendre en considération ce genre de mesures, la médiane a été privilégiée pour faire les comparaisons.

On peut constater que de manière générale, le système de fichier offrant l’écriture la plus rapide est f2fs, suivit de ext4. Le système de fichier btrfs offre l’écriture la moins rapide.

*Remarque : Afin d’avoir facilement accès aux trois partitions, j’ai créé et exécutée mon programme C dans le même répertoire d’accès : « /run/media/lmi ». Mon programme prenait un deuxième argument indiquant le chemin du répertoire où créer les fichiers (Ex : «./usr\_f2fs/test\_writing »). Malheureusement, en redémarrant ma machine virtuelle mon code source était introuvable. Je pense que c’était une mauvaise idée de travail dans un répertoire permettant le montage de disque externe. En effet, après un second test, j’ai pu confirmer que le répertoire « /run/media/lmi » était supprimé après un redémarrage.*

## LUKS, cryptsetup et dmcrypt

### Créer une troisième partition LUKS

Pour commencer, j’ai créé la troisième partition :

*sudo parted /dev/sdb mkpart primary ext4 4358144s 6455295s*

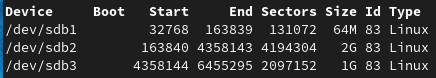
Le début en secteur (s) de la première partition créée est 4358144, qui est la suite (+1) de la fin de la partition précédant. Ensuite, il faut indiquer la fin de la partition en secteur. Afin de faire une partition de 1GB il faut ajouter *2 097 151* (Taille de la partition / Taille d’un secteur – 1 = 1GB / 512 - 1). Nous trouvons donc *6455295*s.

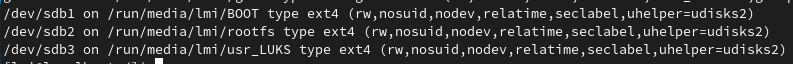
Ensuite, j’ai formaté la partition en ext4 :

*umount /dev/sdb3*

*sudo mkfs.ext4 /dev/sdb3 -L usr\_LUKS*

On peut vérifier le bon déroulement des commandes :





### LUKS, cryptsetup options

1. « Plain Mode » vs « LUKS extension mode » : La différence est que LUKS utilise un en-tête de métadonnées et peut donc offrir plus de fonctionnalités que dm-crypt (Plain Mode). D'autre part, la en-tête est visible et vulnérable aux dommages. À moins de très bien comprendre les contextes de cryptographie, il est en général conseillé d’utiliser LUKS.
2. *–hash* permet de remplacer la fonction de hachage de phrase de passe (passphrase) par défaut.
3. La valeur par défaut actuelle dans les sources distribuées pour le chiffrement est "aes-xts-plain64" pour LUKS.
4. *--key-file* permet principalement de fournir le passphrase par un fichier.

### LUKS test 1

***Initialiser une partition LUKS en format ext4 et la monter dans le répertoire /mtn/usr***

Voici les commandes effectuées pour initialiser la troisième partition LUKS :

*umount /dev/sdb3*

*sudo cryptsetup --debug --pbkdf-memory 256 luksFormat /dev/sdb3*

*sudo cryptsetup --debug open --type luks /dev/sdb3 luks\_test*

On peut vérifier le bon fonctionnement du mapping :



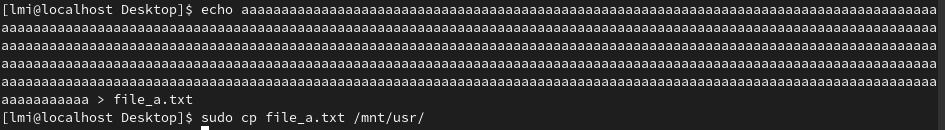
Voici les commandes effectuées pour formater et monter la troisième partition LUKS :

*sudo mkfs.ext4 /dev/mapper/luks\_test*

*sudo mkdir /mnt/usr*

*sudo mount /dev/mapper/luks\_test /mnt/usr/*

***Copier un fichier dans la partition LUKS***

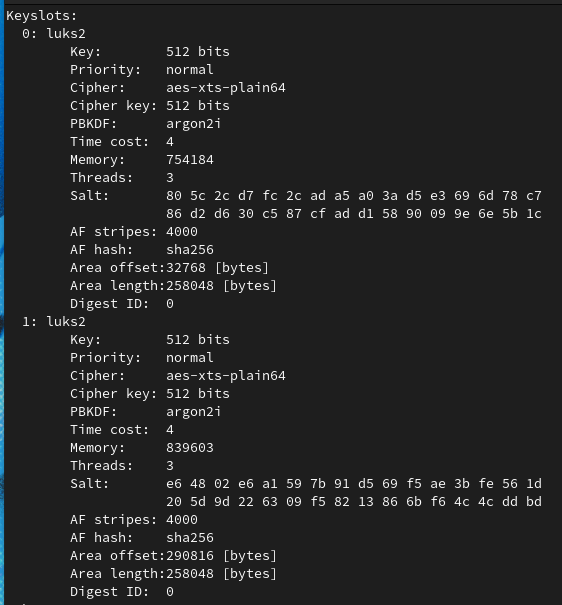


***Ajouter un nouveau passphrase à la partition LUKS***

Pour ce faire, il suffit d’utiliser la commande suivante :

*sudo cryptsetup luksAddKey /dev/sdb3*

Il est possible de vérifier qu’une second clef a été ajoutées en regardant l’en-tête du device LUKS avec cette commande : « sudo cryptsetup luksDump /dev/sdb3 » :

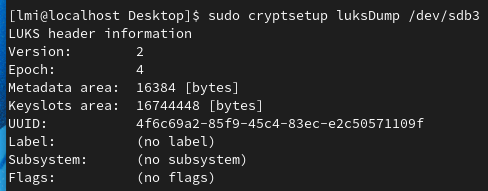


0

***Dumper l’en-tête de la partition et la clé principale cryptée***

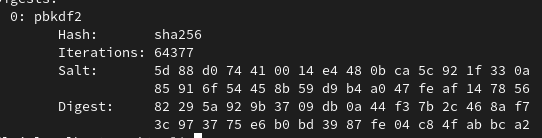
Comme nous l’avons fait précédemment, voici la commande pour afficher l’entête de la partition :

*sudo cryptsetup luksDump /dev/sdb3*



Il est aussi possible d’utiliser cette commande afin d’afficher la clé principale cryptée :

*sudo cryptsetup luksDump -dump-master-key /dev/sdb3*

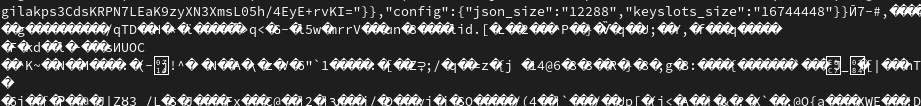


***Dumper 1 Mbytes de la partition /dev/sbd3 sur un fichier. Peut-on trouver l’en-tête de la partition et le clé principale cryptée ?***

Voici la commande à effectuer pour dumper et afficher la troisième partition :



On peut constater que nous avons accès à l’en-tête de la partition ainsi qu’à la clé principale cryptée. La suite du fichier est complétement indéchiffrable, alors qu’il contient le fichier précédemment inséré avec beaucoup de caractères ‘a’ :



Après utilisation, il ne faut pas oublier de libérer les ressources :

*sudo umount /dev/mapper/luks\_test*

*sudo cryptsetup close luks\_test*

***Activer la partition cryptée sur la NanoPi***

Voici les commande à effectuer afin de pouvoir utiliser la partition cryptée sur la cible NanoPi :

*cryptsetup --debug open --type luks /dev/mmcblk0p3 usr\_luks*

On peut vérifier le mapping :



*mkfs.ext2 /dev/mapper/usr\_luks*

*mkdir /mnt/usrfs*

*mount /dev/mapper/usr\_luks /mnt/usrfs/*

Maintenant il est possible d’utiliser la partition cryptée via /mnt/usrfs

Après utilisation, il ne faut pas oublier de libérer les ressources :

*umount /dev/mapper/usr\_luks*

*cryptsetup close usr\_luks*

### rootfs dans une partition LUKS

Le but de ce chapitre est d’avoir un rootfs crypté sur une partition LUKS.

***Générer un passphrase aléatoire dans un fichier***

Voici la commande permettant de générer un passphrase aléatoire dans un fichier :

*dd if=/dev/urandom of=file\_passphare bs=1 count=64*



***Initialise une partition LUKS sur la partition 3***

Voici la commande permettant d’initialiser une partition LUKS avec une fonction de hash « sha512 », une taille de clef de 512 et un « passphare » venant d’un fichier donné :

*sudo cryptsetup --debug --pbkdf-memory 256 --hash sha512 --key-size 512 --key-file ./file\_passphare luksFormat /dev/sdb3*

***Créer un mapping /dev/mapper/usrfs1***

Voici la commande permettant de créer un mapping entre la partition et un device :

*sudo cryptsetup --key-size 512 --key-file ./file\_passphare --debug open --type luks /dev/sdb3 rootfs\_luks*

Nous pouvons contrôler le bon fonctionnement de la commande :



***Formater la partition LUKS en format ext4***

Voici la commande permettant de formater la partition en format ext4 :

*sudo mkfs.ext4 /dev/mapper/rootfs\_luks*

***Copier le rootfs sur la partition LUKS***

Voici la commande permettant de copier le rootfs sur la partition :

*sudo dd if=~/workspace/nano/buildroot/output/images/rootfs.ext4 of=/dev/mapper/rootfs\_luks bs=4M*

Remarque : après utilisation, il ne faut pas oublier de libérer les ressources :

*umount /dev/mapper/usr\_luks*

*sudo cryptsetup close usr\_luks*

***Démarrer la NanoPi et monter manuellement la partition LUKS***

Premièrement, il est nécessaire d’avoir le fichier contenant le passphare aléatoire créé précédemment. Pour ce faire, j’ai utilisé le nfs afin de transmettre le fichier « file\_passphare » à la cible NanoPi.

Ensuite, j’ai créé un device afin d’avoir accès à la partition encryptée :

*cryptsetup --debug --key-size 512 --key-file ./file\_passphare open --type luks /dev/mmcblk0p3 usr\_luks*

Nous pouvons vérifier le bon fonctionnement de la commande précédente :

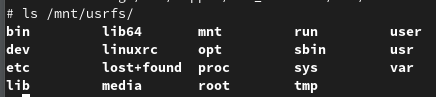


Puis, j’ai monté la partition LUKS contenant le rootfs sur le répertoire /mnt/usrfs :

*mkdir /mnt/usrfs*

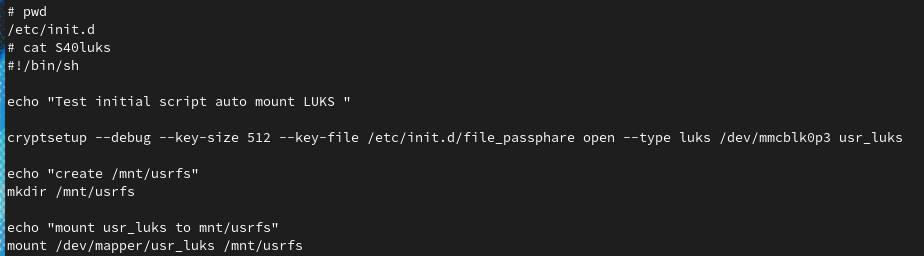
*mount /dev/mapper/usr\_luks /mnt/usrfs*

Nous pouvons constater que le montage c’est correctement déroulé :



***Écrire un script (/etc/init.d/S40luks) afin de monter automatiquement la partition***

Pour ce faire, il suffit d’écrire les commandes précédentes dans un scirpt nommé « S40luks » et de placer ce scirpt dans le dossier « /etc/init.d ». Cependant, il est nécessaire d’indiquer le chemin correcte du fichier contenant le « passphare ». Voici à quoi ressemble mon scirpt :

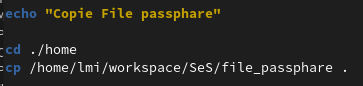


En ajoutant ce script, la cible NanoPi se comporte comme attendue et monte automatiquement la partition encryptée.

### Initramfs

***Générer un initramfs***

Afin de générer un initramfs, j’ai suivi les directives données dans le document « File Systems, LUKS, InitRamFS » du cours SeS. De plus, il a fallu ajouter au initramfs notre fichier contenant le « passphare » pour décrypter la partition LUKS. Voici les quelques lignes qui permettent de copier le fichier passphare :

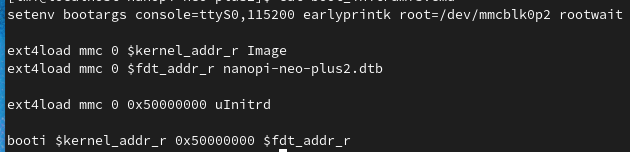


Finalement, la création d’un script permettant de construire complètement l’initramfs est créé. Ce script est disponible dans les annexes [[2]](#_Annexes).

Il est intéressant d’exécuter la commande « tree /home/lmi/ramfs/ » afin de s’assurer que tous les répertoires, devices, libraires, exécutables et scripts soient correctement créent.

***Initialiser la NanoPi afin de démarrer l’initramfs***

Il est nécessaire de modifier le fichier « boot.cmd » correspondant à notre carte (~/workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/boot.cmd) afin de charger l’initramfs fraichement créé à une adresse disponible (0x5000 0000). De plus, il faut indiquer cette adresse grâce au deuxième argument de la commande « booti » :

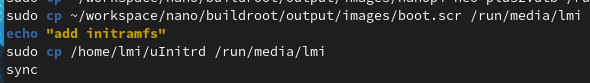


Une fois ce fichier modifié, il faut générer l’image de ce fichier pour U-boot via la commande « mkimage » :

*mkimage -C none -A arm64 -T script -d board/friendlyarm/nanopi-neo-plus2/boot.cmd ./output/images/boot.scr*

Il faut bien entendu, mettre à jour ce fichier sur la carte SD.

De plus, il faut aussi modifier notre script qui permet de générer notre carte SD afin d’ajouter l’initramfs sur la partition 1 (BOOT).



***Démarrer la NanoPi et démarrer manuellement***

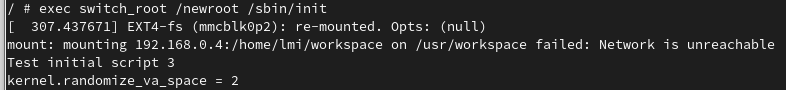
Premièrement, on peut s’assurer que la modification précédente est bien appliquée :



Ensuite, pour démarrer manuellement, en changeant de système de fichier racine, il faut exécuter la commande suivante :

*exec switch\_root /newroot /sbin/init*

Nous pouvons voir après cela que le lancement s’effectue correctement et commence par exécuter le script « S00KernelParameter » qui exécute « sysctl -p » :



***Depuis le shell sur l’initramfs, monter la partition LUKS***

Pour monter la partition LUKS, une fois sur le shell de l’initramfs, il suffit d’exécuter la commande suivante :

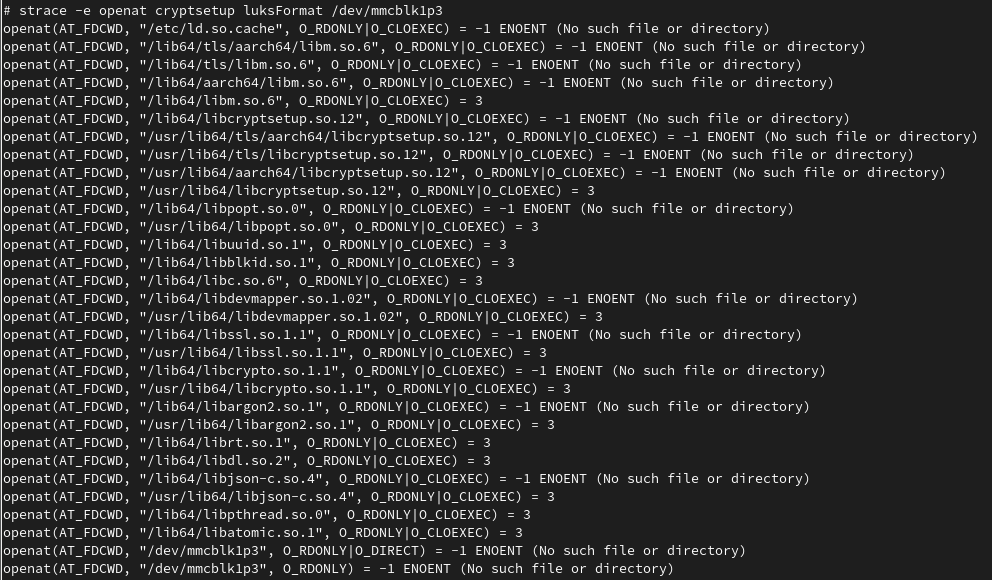
*cryptsetup --debug --key-size 512 --key-file /home/file\_passphare open --type luks /dev/mmcblk0p3 usr\_luks*

Cependant, cette commande n’est pas connue par l’initramfs :



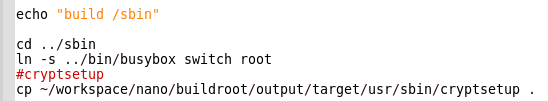
Il faut donc ajouter l’exécutable dans le système de fichier du initramfs ainsi que toute les librairies dépendantes. Pour connaitre toutes les libraires dépendantes, nous pouvons utiliser la commande strace depuis le rootfs :

*strace -e openat cryptsetup luksFormat /dev/mmcblk1p3*

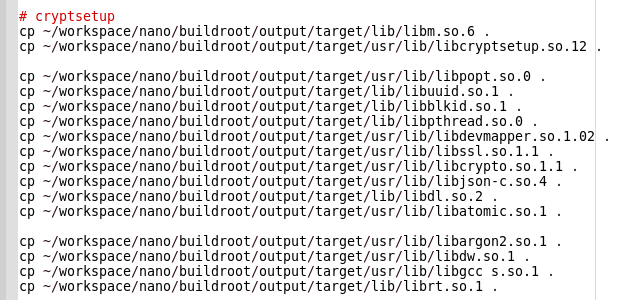


On peut voir ci-dessus, toutes les librairies utilisées par l’exécutable « cryptsetup ».

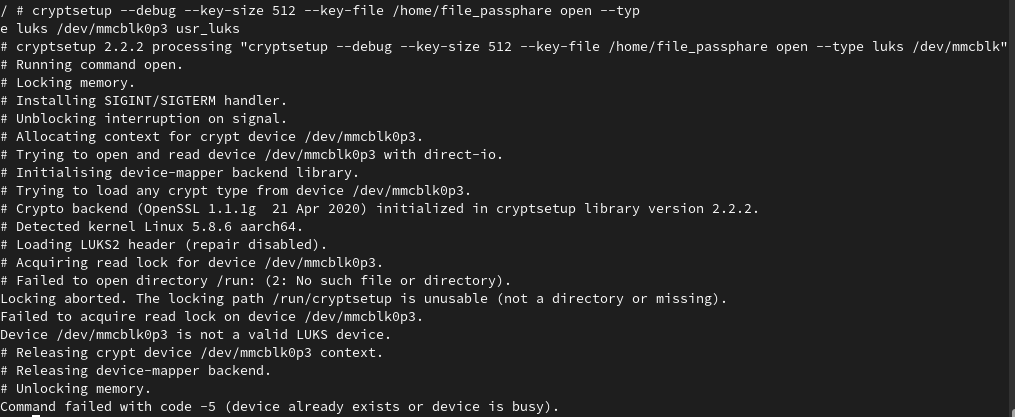
Afin d’ajouter tous ces fichiers, j’ai utilisé la commande « find » pour localiser les différents fichiers et j’ai modifier mon script construisant l’initramfs. Voici l’ajout du fichier exécutable :



Et voici les librairies dépendantes :



Une fois l’exécutable et toutes les librairies ajoutées au système de fichier, une deuxième tentative a été réalisée :



On peut constater qu’un dossier « /run » est nécessaire au bon fonctionnement. Dans un premier temps, il est possible de simplement exécuter la commande « mkdir /run ». Cependant, il est nécessaire de modifier notre script de construction de l’initramfs afin d’avoir ce répertoire par défaut :



Finalement, le montage de la partition LUKS est possible :



*Remarque : Durant ce chapitre, plusieurs commandes a été utilisées afin de mettre à jour l’initramfs ;*

1. *geany build\_initramfs : Pour modifier mon script de construction*
2. *./build\_initramfs : Pour appliquer mes modifications*
3. *tree /home/lmi/ramfs/ : Pour valider mes modifications*
4. *sudo cp /home/lmi/uInitrd /run/media/lmi/BOOT/ : Pour mettre à jour le système de fichier sur la carte SD.*

***Démarrer manuellement la partition roofs encryptée***

Le but est de changer de système de fichier racine avec la partition LUKS précédemment montée. Pour ce faire, voici les commandes à exécuter :

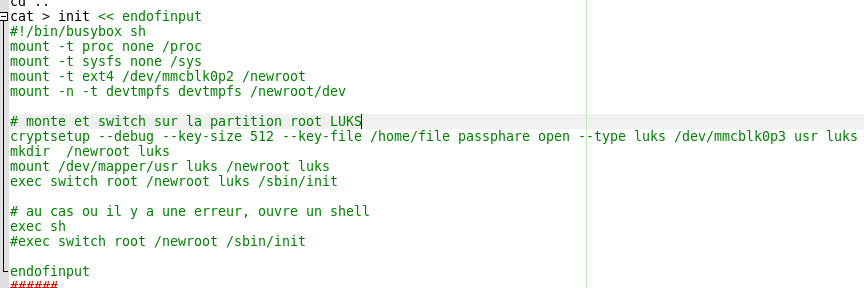
mkdir /newroot\_luks

mount /dev/mapper/usr\_luks /newroot\_luks

exec switch\_root /newroot\_luks /sbin/init

***Écrire un scrpit afin de démarrer automatiquement la partition rootfs encryptée.***

Pour automatiser les étapes précédentes, il suffit d’ajouter les quatre commandes utilisées dans le fichier « init », à la racine du initramfs. Personnellement, j’ai ajouté ceci directement dans mon script permettant de construire l’initramfs :



En appliquant ces modifications, la cible NanoPi se comporte comme attendue et démarre automatiquement sur la partition rootfs encryptée.

## Conclusion File system

Ce laboratoire m’a permis de mieux comprendre et surtout d’appliquer les différents principes sur les initramfs et les partitions LUKS. Personnellement, je n’avais aucune connaissance sur l’encryptage des partitions. Maintenant je vois plus clairement comment cela peut être mis en place. J’ai pu prendre conscience de la construction d’un initramfs et de l’importance de libraires dépendantes. J’ai pu apprendre l’outil « cryptsetup » et l’utilisation devices de type « mapper ». Finalement, j’ai découvert le principe général sur la génération des clefs et leurs utilités.

Durant ce laboratoire, le passphare est stocké dans un fichier sur la cible. Cette méthode fonctionne mais pour un cas réel, je pense qu’elle n’est pas idéale. Il serait plus judicieux d’utiliser une trust zone matériel si possible. Sinon, il est envisageable de récupérer le passphare via un serveur ssh ou une clef USB.

# Firewall

## Introduction

Durant ce laboratoire nous allons principalement nous familiariser avec ;

# Annexes

1. Donnée laboratoire « FileSystem »
2. Script de construction d’un initramfs

Date : x.yxx.20

Nom de l’étudiant : Spinelli Isaia

1. <http://kb.ictbanking.net/article.php?id=668&oid=2> [↑](#footnote-ref-1)