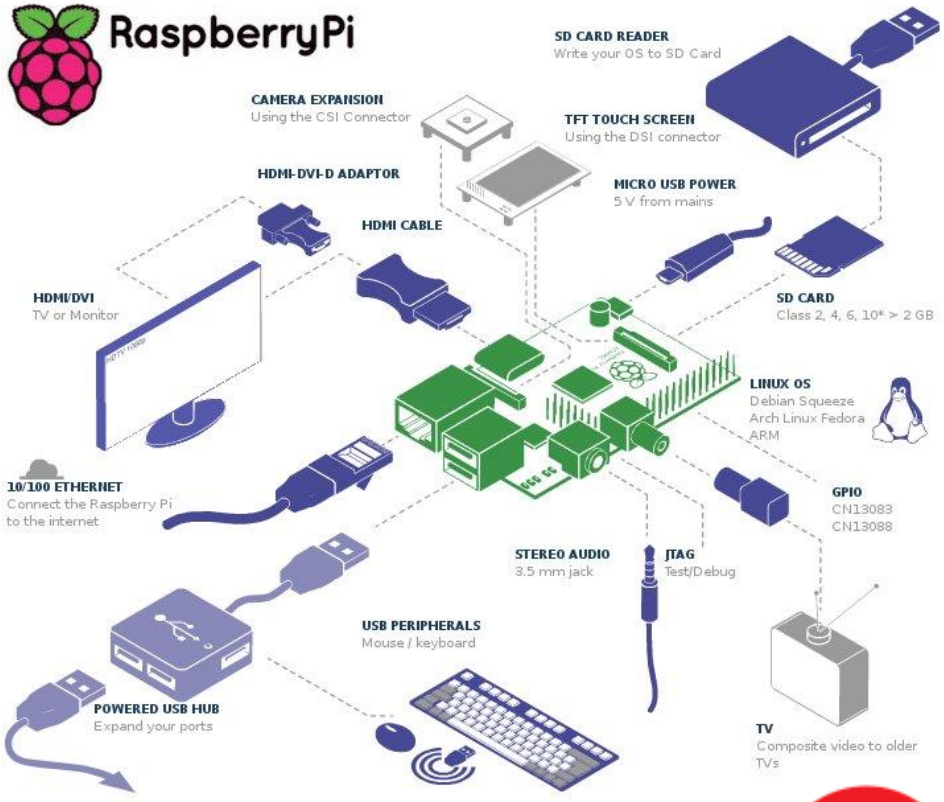
# 2013/2014

## M2 - Informatique SSI

**GULDNER Geoffrey**
**LACAVE Valentin**




CIRCL
Computer Incident
Response Center
Luxembourg

# [CryptoCIRCLEAN]

# SOMMAIRE

# I. Préambule

Dans le cadre de notre formation pour l'UE - Sécurité des Réseaux, nous avons effectué un projet sur Raspberry Pi appelée CryptoCIRCLEAN qui est une adaptation d'un projet existant : CIRCLEAN. Le projet existant est en réalité un système de copie de clef USB sécurisé : l'utilisateur va connecter une première clé USB avec des fichiers qui sont potentiellement dangereux pour ensuite connecter une seconde clé USB sûre. Le système va transférer les fichiers de la 1ère clé à la 2ieme en les convertissant dans un format passif et donc non dangereux pour l'utilisateur.

Notre travail consiste dans un premier temps à s'approprier le fonctionnement de CIRCLEAN et de le modifier pour mettre en place un système de cryptage de donnée afin d'éviter la fuite potentielle d'informations qui peuvent être sensibles. Puis dans un second temps, on mettra en place un système donnant la clef de cryptage via un synthétiseur de voix (possibilité d'écoute via la prise jack du raspberry pi). CryptoCIRCLEAN fonctionnera de la même manière que CIRCLEAN à la différence que quand l'utilisateur pluggera les clefs USB, le raspberry pi ne fera pas une conversion en fichier non actif mais vas les crypter dans un format spécial avec une clef de session.
On utilisera le logiciel GPG pour le cryptage des données et pour la génération de clef. Pour synthétiser la voix, on utilisera espeak.

Remerciements : Raphaël Vinot, Alexandre dulaunoy

# II. Génération de clef GPG

## A. Système de génération

Pour générer les clefs de session, on va utiliser la commande qui suit :

- `gpg --gen-random 0|1|2 count`

Emit *count* random bytes of the given quality level 0, 1 or 2. If *count* is not given or zero, an endless sequence of random bytes will be emitted. If used with `--armor` the output will be base64 encoded.

## B. Entropie

### *1) générateur d'entropie*

Pour avoir des clefs avec une entropie élevée, on peut utiliser divers systèmes qui génèrent de l'entropie :

- ***ls -R /*** : qui liste tous les fichiers (de façon récursive) présent à la racine afin de générer de l'entropie. L'avantage de ce système est qu'aucun package n'est à installer et aucune manipulation n'est à faire, l'entropie générée dépend des fichiers sur la carte SD mais aussi sur les clefs USB donc il peut y avoir des entropies différentes. Le désavantage de cette technique est qu'il liste les fichiers et les affiche, ce qui est plutôt lent.
- ***Havege*** : qui est un logiciel de création d'entropie (HArdware Volatile Entropy Gathering and Expansion). L'avantage est qu'il génère beaucoup d'entropie. Le désavantage c'est que d'après la description il n'est pas compatible processeur ARM (https://www.irisa.fr/caps/projects/hipsor/install.php) et il faut installer un package et faire des manipulations.
- ***/dev/random - /dev/urandom*** : lors d'une lecture, le périphérique /dev/random sera limité au nombre de bits de bruit contenus dans le réservoir d'entropie. /dev/random est particulièrement adapté pour les cas où on a ponctuellement besoin de nombres hautement aléatoires (création de clés par exemple). Lorsque le réservoir d'entropie est vide, les lectures depuis le périphérique /dev/random seront bloquantes jusqu'à l'obtention de suffisamment de bruit en provenance de l'environnement. Le désavantage de ce système est qu'il est déconseillé d'utiliser /dev/urandom pour les clés GPG / SSL / SSH à longue durée de vie.

Dans une optique de facilité, on utilisera ls -R /. On redirigera l'affichage dans le « trou noir » (/dev/null) pour un gain de temps pour la création de la clef de session.

### *2) Calcul d'entropie*

Pour connaitre l'entropie du Raspberry pi, on va générer un jeu de test ou on l'on créera 1000x clefs de session mais avec une taille différente 9 - 12 - 15 - 18 et un random différent 0|1|2 tous cela sur un OS Raspbian. Enfin pour vérifier l'efficacité du ls -R / on va faire 2 jeux de tests : une fois avec un ls -R / avant chaque création de clef (LS) et un jeu de test ou on effectuera un seul ls -R / en début de création de clefs (SLS). Puis ensuite on calcule l'entropie grâce à une série de calcul trouvée sur un site (http://gynvael.coldwind.pl/?id=162).
En résumé le calcul s'effectue en analysant la fréquence d'occurrence de chaque caractère dans la clé.
Entropie + = Occurrence_Number [X] ^ 2 / DATA_SIZE, où X sont les valeurs successives de l'octet (de 00 à FF), à la fin, nous pouvons diviser l'entropie par la taille des données (en octets), et l'inverser (X = 1-X). Enfin on prend l'entropie trouvée que l'on met au carré puis on multiplie par 100 pour avoir le pourcentage.

**KEY = ABCABCCABCABACBABABABABABABABCABACBACBABABACBABACBAB**

| | | |
|---|---|---|
| | E = 0 | (E=0) |
| A – 20 | E += 20 * 20 / 50 | ('A' E = 8) |
| B – 20 | E += 20 * 20 / 50 | ('B' E = 16) |
| C – 10 | E += 10 * 10 / 50 | ('C' E = 18) |
| other – 0 | E = 1 - (E / 50) | (E = 0.64) |
| | E'' = E^2 * 100 | (E'' = 40.96%) |

Donc ici on peut voir qu'il y a 40.96% d'entropie (de chaos) dans cette clef.

Comme on peut le voir ci-dessous, nous avons généré 1000 clefs (qui sont dans des fichiers situés dans le dossier entropy/KEY GPG du git) puis nous avons appliqué l'algorithme de calcul d'entropie sur les clefs (qui sont dans des fichiers situés dans le dossier entropie/KEY GPG ENTROPY du git). La création de clef c'est bien passée sauf pour le random 2 qui dut au caractère aléatoire de la création de clef a été impossible. Pour faire simple, il générerait 3-4 clefs puis après plus aucune par manque de bytes et des fois, il en crée même pas une seule, toujours dû au manque de bytes que soit avec le ls-R / ou tout autre système (nous avons quand même joint les fichiers txt avec les clefs qu'il arrive à générer et dans le fichier excel).

| LS | | | Count \| V | SLS | | |
|---|---|---|---|---|---|---|
| **0** | **1** | **2** | **<-Random->** | **0** | **1** | **2** |
| 96,7735475 | 96,7749133 | Impossible | **9** | 96,7724514 | 96,7687507 | Impossible |
| 96,7927855 | 96,7970457 | Impossible | **12** | 96,7956986 | 96,8050738 | Impossible |
| 96,8286276 | 96,8284671 | Impossible | **15** | 96,8234142 | 96,8257414 | Impossible |
| 96,8405005 | 96,8366348 | Impossible | **18** | 96,8370991 | 96,8468941 | Impossible |

| Récapitulatif | | | Count \| V |
|---|---|---|---|
| **0** | **1** | **2** | **<-Random->** |
| LS | LS | Impossible | **9** |
| SLS | SLS | Impossible | **12** |
| LS | LS | Impossible | **15** |
| LS | SLS | Impossible | **18** |

## 3) Script

Le script ci-dessous est celui qui permet la création de KEY GPG avec un ls-R / avant de créer chaque clef. Comme on peut le remarquer le script en bash prend 4 paramètres le premier est le fichier ou l'on va y stocker nos clef, le second est le nombre de clef que l'on veut générer, le troisième est le random et enfin le dernier est le count (taille de la clef). D'abord on initialise le compteur (cmpt) à 0, puis on crée le fichier grâce au touch, et on rentre dans la boucle qui va générer le nombre de clef souhaitée. Dans la boucle on trouve le ls-R / avec une redirection sur /dev/null, ensuite on crée la clef que l'on stocke dans le fichier txt on ajoute +1 au cmpt et on affiche le cmpt pour savoir où on se trouve dans la boucle.

```
1  #!/bin/bash
2  #$1 = nom du fichier
3  #$2 = nombre de bouclage
4  #$3 = type de random 0/1/2
5  #$4 = nombre de caractere
6
7  cmpt=0
8  touch $1
9  while test $cmpt != $2
10 do
11              ls -R / >> /dev/null
12              gpg --gen-random -armor $3 $4 >> $1
13              cmpt=$(($cmpt+1))
14              echo "$cmpt"
15 done
16 exit 0
```

*CreationClef.sh*

Le script ci-dessous est le même que celui du dessus mise à part que le ls-R / se fait 1 fois à l'extérieur de la boucle juste après le 'touch'. Pour le contenu de la boucle, il est semblable au script précédent..

```
1  #!/bin/bash
2  #$1 = nom du fichier
3  #$2 = nombre de bouclage
4  #$3 = type de random 0/1/2
5  #$4 = nombre de caractere
6
7  cmpt=0
8  touch $1
9  ls -R / >> /dev/null
10 while test $cmpt != $2
11 do
12              gpg --gen-random -armor $3 $4 >> $1
13              cmpt=$(($cmpt+1))
14              echo "$cmpt"
15 done
16 exit 0
```

*CreationCLefSansLS.sh*

Ce morceau de script est la fonction inArray() qui recherche le nombre d'occurrence d'une lettre dans le tableau de caractère. Tout d'abord cette fonction prend en compte 2 paramètres le premier étant la lettre à rechercher et le second la longueur du tableau. Ensuite on effectue un test pour vérifier qu'il y a bien 2 paramètres, sinon on va dans le 'else' qui lui compte les occurrences des lettres. Pour se faire on crée une variable compteur cmpt1 que l'on met à 0, une seconde variables nb qui renvoie le nombre d'élément dans le tableau, la dernière variable crée est ok initialisée à 0 qui permet de savoir s'il y a eu une modification de la variable occu. Ensuite on fait une boucle qui boucle (merci geoffrey…) tant que cmpt1 est différent de nb. On effectue un test pour savoir si la lettre et présente dans le tabCarac (initialisé dans la fonction dissection vue plus bas). Si c'est le cas on a occu qui prend cmpt1 et ok qui vaut 1, si ce n'est pas le cas ok reste a 0 et on renvoie occu= -1 pour plus tard.

```
1  #!/bin/bash
2  #$1 = nom du fichier de clef
3  #$2 = nom du fichier des entropie
4
5  inArray()
6  {
7          #$1 = lettre à rechercher
8          #$2 = longueur tableau
9          if [[ "$#" -ne 1 ]]; then
10                 exit
11         else
12                 cmpt1=0
13                 nb=${#tabCarac[*]}
14                 ok=0
15
16                 while test $cmpt1 != $nb
17                 do
18                         if [ "${1}" == "${tabCarac[${cmpt1}]}" ]; then
19                                 occu=$cmpt1
20                                 ok=$(($ok+1))
21                         fi
22                         cmpt1=$(($cmpt1+1))
23                 done
24
25                 if [ $ok -eq 0 ]; then
26                         occu=-1
27                 fi
28         fi
29 }
```

*EntropyClef.sh (fonction inArray())*

Ce morceau de script est la fonction Dissection() dont son but est d'initialiser les tableaux de caractères (tabCarac) et d'occurrence (tabOccu). Elle prend en paramètre une variable qui est la clef de cryptage. Puis on crée 3 variables la clef initialisé avec la clef de cryptage, longueur initialisé avec la taille de la clef (nombre de caractère) et un compteur à 0. Ensuite on a une boucle qui boucle tant que cmpt et diffèrent de la longueur de la clef. On crée une variable charatest (qui et le caractère à tester) initialisé part un caractère de la clef en fonction du compteur. D'abord on vérifie, si on est au premier élément si c'est le cas on l'ajoute au tableau de caractères et sont occurrence est de 1 vue que c'est le premier caractère. Puis pour les éléments suivant on vérifie d'abord s'ils sont déjà présents dans tabCarac si occu est supérieur à 0 alors on ajoute +1 a la bonne position dans le tabOccu. Si ce n'est pas le cas (le pourquoi occu -1 dans inArray()) on ajoute le caractère au tableau et on met son occu a 1.

```
31  Dissection()
32  {
33          #$1 = clef de cryptage
34          clef=$1
35          longueur=${#clef}
36          cmpt=0
37
38          while test $cmpt != $longueur
39          do
40                  charatest=${clef:$cmpt:1}
41
42                  if [ $cmpt -eq 0 ]; then
43                          tabCarac[$cmpt]=$charatest
44                          tabOccu[$cmpt]=1
45
46                  else
47                          inArray $charatest
48
49                          if [ $occu -ge 0 ]; then
50                          tabOccu[$occu]=$((${tabOccu[$occu]}+1))
51
52                          else
53                          pos=${#tabCarac[*]}
54                          tabCarac[$pos]=$charatest
55                          tabOccu[$pos]=1
56                          fi
57                  fi
58                  cmpt=$(($cmpt+1))
59          done
60  }
```

*EntropyClef.sh (fonction Dissection())*

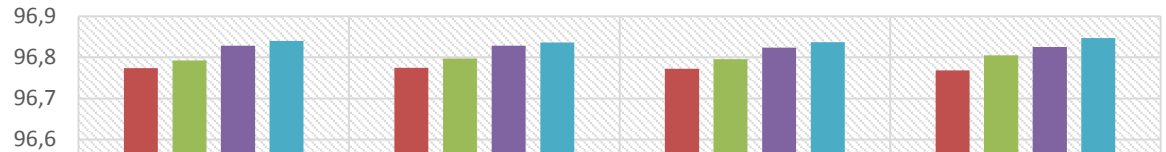
Cette partie du code est le main situé juste après les fonctions inArray() et Dissection(). Dans un premiers temps, on fait un touch (si le fichier n'existe pas il le crée sinon il ne fait rien) puis on le supprime pour éviter toute confusion ou tout ajout de clef où il ne faut pas. On crée une variable nbKey que l'on initialise à 0. Puis on crée une boucle qui va lire toute les lignes du fichier de clef. Dans un premier temps on utilise la fonction dissection pour disséquer la clef. Ensuite on crée une série de variable (pas d'explication car explicite). Enfin on applique l'algorithme de calcul d'entropie très simple. Tout à la fin on ajoute la clef et son entropie dans un nouveau fichier et on affiche à quel numéro de clef on en est.

```
62  touch $2
63  rm $2
64
65  nbKey=0
66
67  while read line
68  do
69          Dissection $line
70          tailleTabCarac=${#tabCarac[*]}
71          tailleTabOccu=${#tabOccu[*]}
72          iteent=0
73          itetailleclef=0
74          tailleclef=0
75          tmpmul=0
76          tmpdiv=0
77          entropy=0
78
79          while test $itetailleclef != $tailleTabOccu
80          do
81                  tailleclef=$(($tailleclef + ${tabOccu[$itetailleclef]}))
82                  itetailleclef=$(($itetailleclef+1))
83          done
84
85          while test $iteent != $tailleTabCarac
86          do
87                  tmpmul=$((${tabOccu[$iteent]}*${tabOccu[$iteent]}))
88                  tmpdiv=$(bc -l <<< "$tmpmul / $tailleclef")
89                  entropy=$(bc -l <<< "$entropy + $tmpdiv")
90                  iteent=$(($iteent+1))
91          done
92
93          entropy=$(bc -l <<< "1 - ($entropy / $tailleclef)")
94          entropyp=$(bc -l <<< "($entropy * $entropy)")
95          entropy=$(bc -l <<< "($entropy * 100)")
96          entropyp=$(bc -l <<< "($entropyp * 100)")
97          echo -e "$line \t $entropyp" >> $2
98
99          nbKey=$(($nbKey+1))
100         echo $nbKey
101
102 done < $1
103 exit 0
```

*EntropyClef.sh (main)*

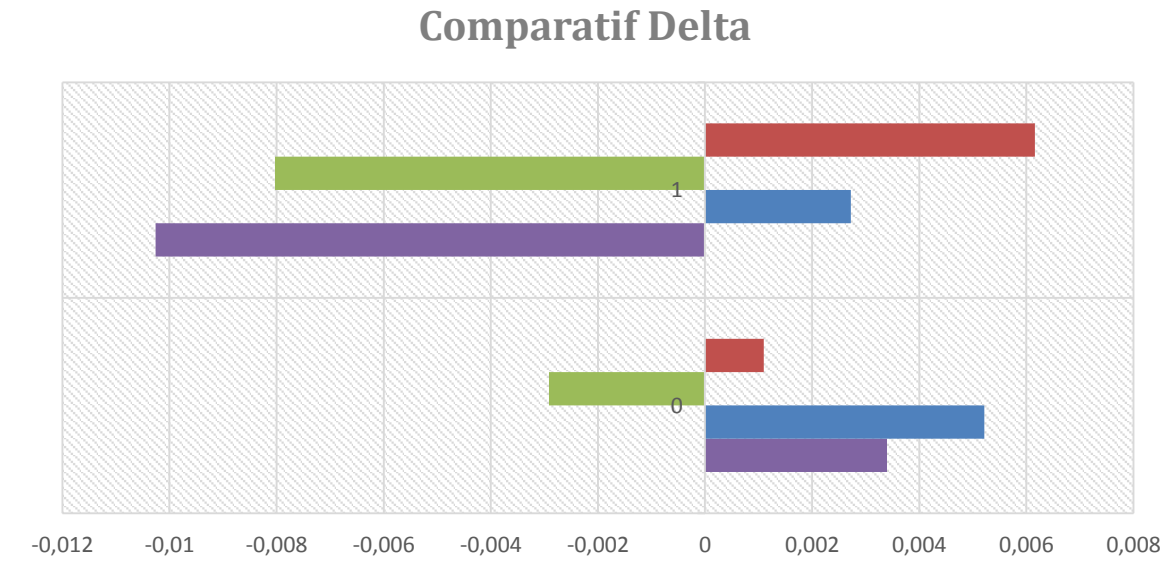
## C. Résultat

Dans cette partie, nous exposons les différents graphiques produits via nos séries de test. Comme expliqué plus haut nous avons produit 1000 clés ou tout du moins essayé car avec un **random** de 2 au bout de quelques clés l'OS nous donne un message d'erreur comme quoi, il manque des bytes pour l'entropie. Donc nous avons exclu des graphiques le random 2 (nombre de clé aléatoire et pas assez suffisante pour faire un vrai comparatif avec les autres clés).



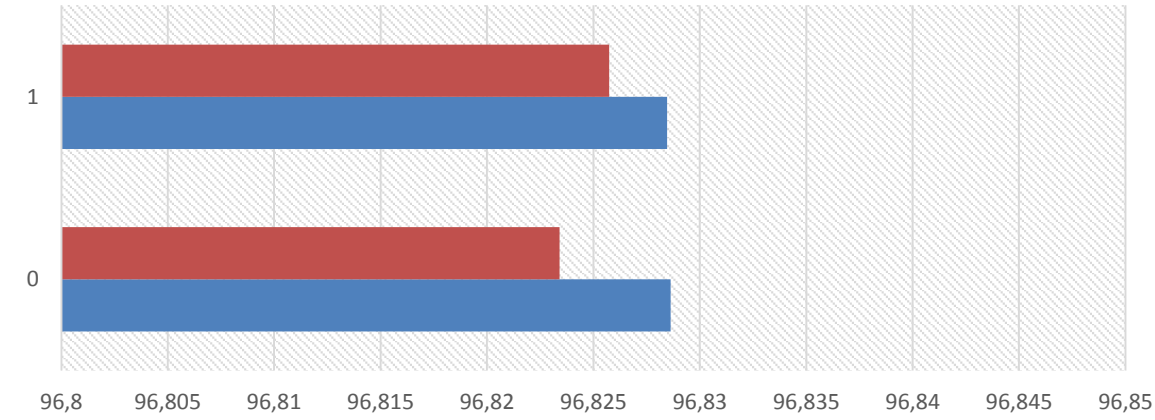

| | LS 0 | LS 1 | SLS 0 | SLS 1 |
|---|---|---|---|---|
| 9 | 96,77354749 | 96,77491327 | 96,77245139 | 96,76875068 |
| 12 | 96,79278554 | 96,79704571 | 96,79569862 | 96,80507377 |
| 15 | 96,82862762 | 96,82846713 | 96,82341423 | 96,82574144 |
| 18 | 96,84050049 | 96,83663479 | 96,83709914 | 96,84689406 |

Dans ce graphique, je fais le comparatif des différentes solutions en termes de **count** pour le nombre de caractères de la clé. Comme on peut le constater plus la clé a un **count** élevé plus celle-ci a une entropie forte (proche de 100%). On remarque aussi qu'avec une disparité entre les clés en fonction du au faite que l'on génère ou pas de l'entropie.




| | 0 | 1 |
|---|---|---|
| 9 | 0,001096097 | 0,006162591 |
| 12 | -0,002913078 | -0,008028057 |
| 15 | 0,00521339 | 0,002725693 |
| 18 | 0,003401355 | -0,010259266 |

Ce graphique nous montre le delta entre la génération d'entropie avant la création de clé ou sans (la disparité dite plus haut et rendu plus visible ici). Comme on peut le remarquer, le LS génère de l'entropie et donne donc des clés avec une meilleure entropie (chiffre positif) mais pas dans tous les cas. Ceci s'explique par le fait que le Raspberry utilise le max d'entropie possible dans l'OS. D'où le fait que les chiffres sont très proche et qu'une fois sans le LS les clés sont meilleur. Notre hypothèse est que le Raspberry régénère de l'entropie à chaque fois qu'il en utilise plus ou moins efficacement.




| | 0 | 1 |
|---|---|---|
| SLS | 96,82341423 | 96,82574144 |
| LS | 96,82862762 | 96,82846713 |

Dans le but de la génération de clé pour crypter les fichiers, nous avons décidé d'utiliser un count de 15 et un random de 1 avec une génération d'entropie. Avec un LS –R / on a une meilleur entropie et donc de meilleures clés (comme le démontre le graphique), le count de 15 a été choisi car il faut donner la clé oralement et cela semblait un bon compromis (18 aurait était trop long à dire et 12 trop faible) et pour finir un random de 1 pour avoir un bonne entropie (un random de 2 génère des meilleur clé mais des tests ont démontré qu'une fois sur 15 il ne génère pas de clé par manque de bytes que cela soit avec le LS –R / ou sans).

# III. Scripts

## A. Tests

Les scripts de test sont disponibles sur le git (/scripts), ces scripts nous ont permis de développer et de tester les fonctions nécessaires à implémenter par la suite sur le raspberry pi. Parmi ces scripts, on trouve :

- createKeyEnv.sh : nous as permis de tester la création de clé GPG et de la stocker dans une variable d'environnement, après plusieurs recherches et au vu de l'architecture de CIRCLEAN il n'a pas été très utile.
- createKeyTmp.sh : est basé sur le script createKeyEnv.sh à la différence près que la clé est stockée dans /tmp/key pour éviter toute écriture dans une mémoire morte. Le script est commenté et accompagné d'un 'ls –R /' pour générer plus d'entropie. Il est utilisé pour le raspberry pi (version adaptative, nous verrons le détail plus loin).
- encrypt.sh : ce script permet l'encryptions d'un fichier passé en paramètre, la ligne GPG est accompagnée de plusieurs arguments.
- espeak.sh : premier test avec espeak
- espealKey.sh: script final pour épeler la clé GPG avec un alphabet NATO
- stringToNato.sh : script trouvé sur git (https://gist.github.com/bradland/1523933) qui permet de convertir une string en alphabet NATO

Tous ces scripts nous ont donné une bonne base pour passer à la suite du développement : une fois toutes les fonctions connues, nous les avons adaptés à Circlean pour créer CryptoCirclean.
Cela nous a aussi permis de revoir le bash et les fonctions de bases. Tous les scripts de tests sont commentés et fournis (sur git), c'est pourquoi ils ne sont pas détaillés dans ce rapport.

## B. Raspberry Pi

Afin de pourvoir adapter les scripts de test au raspberry, nous avons d'abord regardé comment fonctionnai Circlean :

- Le fichier constraint.sh contient des chemins, nous y en avons rajouté 3 :
  - KEY="${TMP}key" pour le path de la clé GPG
  - GPG="${TMP}.gnupg" pour le répertoire temporaire de GPG
  - TRASH='/dev/null' pour la « poubelle affichage »
- Le fichier init.sh permet de lancer les scripts de base, nous y avons rajouté :
  - ./createKeyTmp.sh pour créer la clé GPG
  - ./espeakKey.sh pour dicter la clé GPG
- Les fichiers createTmpKey.sh et espeakKey.sh qui sont nos créations, ils permettent respectivement de créer la clé GPG dans KEY et de dicter la clé GPG avec espeak en alphabet NATO
- Le fichier functions.sh est le cœur du programme, il parcourt les fichiers dans la partition. Nous avons modifié presque entièrement le fichier (seule la structure qui permet de parcourir les fichiers a été gardée), pour chaque fichier, ont recréé le répertoire correspondant et on y met le fichier crypté.

Concernant les fichiers, nous avons préféré garder le nom d'origine pour que les utilisateurs s'y retrouvent mais rien n'est lisible sans décrypter. Nous avons aussi décidé d'utiliser espeak plutôt que festival car plus simple et plus audible. Car malgré que les 2 soient prévus pour une architecture ARM le son se détériore trop pour festival rendant la clé inaudible alors qu'espeak malgré la détérioration reste audible.

***IMPORTANT : il faut installer espeak avant de pouvoir utiliser nos scripts.***

# IV. Conclusion

Circlean est un projet très intéressant qui permet de transformer des fichiers dans un format « passif » afin d'éviter tout problème (virus par exemple). A partir de ce projet nous avons développé une version capable de crypter les fichiers au lieu de les transformer ; pour ce faire, nous générons de l'entropie sur un raspberry pi et nous réutilisons la structure de Circlean avec GPG : c'est le principe de CryptoCirclean.
Dans la version rendue, nous donnons la clé avec espeak, mais il reste possible de l'écrire sur la clé USB source.