## Sécurisation des communications

En tout premier lieu avant de faire de l'envoi de fichiers cryptés, il est important de sécuriser les différents échanges de clés entre notre serveur et notre client. L'intérêt de celui-ci est de faire en sorte qu'une personne qui pourrait entendre les différentes communications ne va avoir que des clés chiffrées, et les clés publiques des différentes personnes. Pour cela, nous utilisons RSA étant donné que c'est la méthode la plus sécurisée en termes d'échange de clés à l'heure actuelle du au fait que le coût computationnel pour déchiffrer cette clé est beaucoup trop important et ainsi elle ne permet pas à une personne qui écouterait le transfert de fichier de pouvoir déchiffrer celui-ci.

La force du chiffrement RSA est liée à la taille de la clé : plus la clé est grande, plus la signature est forte. Les progrès faits en analyse cryptographique ont largement influencé l'augmentation de la taille de clé utilisée avec cet algorithme. Les dernières recommandations visent à utiliser des clés de 2048 bits afin de se protéger de n'importe quel ordinateur n'étant pas un ordinateur quantique.

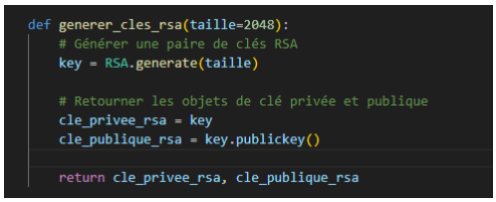
De ce fait, dans le code les générations de clés se font en suivant ces normes, il s'agit évidemment d'un compromis ; plus la taille des clés est importante, plus le chiffrage va demander des ressources et du temps mais plus il est sécurisé. A l'inverse, plus celui-ci est faible, et plus nos échanges seront vulnérables.

Pour effectuer ceci, l'utilisation d'une librairie python : Pycryptodome, s'occupera de toutes les étapes de ce transfert de fichiers. Cette librairie a été choisie suivant plusieurs critères déterminants :

- Elle devait être régulièrement mise à jour ce qui est le cas étant donné que sa dernière version date du 10 janvier 2024.
- Elle ne doit pas présenter de faille de sécurité et être en mesure d'offrir une liberté dans la taille des différents chiffrements (clés RSA/ AES / SHA, etc.)
- Être procédural, c'est-à-dire qu'elle ne doit pas générer constamment les mêmes clés ou suivre une liste de clés et ainsi être déterministe. Ceci poserait de réels problèmes de sécurité. Il suffirait à une personne de décrypter une clé, pour déchiffrer tous les fichiers qui lui sont liés.

C'est donc pour ces raisons que cette bibliothèque a été utilisée plutôt qu'une autre.

Toute la partie de génération de clés RSA réside en ces lignes, l'utilité d'une bibliothèque est multiple. Dans un premier temps, celle-ci permet de réduire fortement le code et se concentrer uniquement sur ce qui nous intéresse. Puis dans un second, généralement, les algorithmes utilisés ont été optimisés au maximum afin d'offrir les meilleures performances possibles.



```python
def generer_cles_rsa(taille=2048):
    # Générer une paire de clés RSA
    key = RSA.generate(taille)

    # Retourner les objets de clé privée et publique
    cle_privee_rsa = key
    cle_publique_rsa = key.publickey()

    return cle_privee_rsa, cle_publique_rsa
```

Ainsi, la fonction va faire appel à la génération de clé RSA dans la bibliothèque de PyCryptodome, puis nous allons retourner un tuple comprenant les deux clés, la publique et la privée.

## Vérification de l'intégrité du fichier

Pour l'envoi d'un fichier il est obligatoire de s'assurer de la qualité de celui-ci, c'est-à-dire est-ce qu'il est intact ou non. Pour cela, la méthode la plus sûre est de procéder au hachage du fichier. Une fonction de hachage cryptographique permet à partir d'un contenu (ici le contenu natif, donc déchiffré du message), de calculer son hash, c'est-à-dire son empreinte. Celle-ci est très rapide dans un sens, et en pratique infaisable dans le sens inverse. Ainsi, il est primordial que celle-ci soit déterministe, c'est-à-dire que pour une même donnée en entrée, le hash calculé soit le même, dans le cas échéant il serait impossible de comparer si le fichier reçu est bien le même que celui envoyé.

Dans la librairie de PyCryptodome, il existe de nombreuses variantes de SHA étant une fonction de hachage, allant de SHA-1 à SHA-3 ; conformément à l'énoncé et aux recommandations de la CNIL, la fonction SHA-1 a été éliminée car dite obsolète de nos jours. De plus, l'algorithme SHA-2 étant construit sur le même schéma que SHA-1 présentant une fragilité, il est recommandé d'utiliser la version la plus récente. Par élimination, SHA-3 avec un hash de 256 bits est la meilleure possibilité, et largement suffisante pour notre utilisation, il est important de noter, qu'à l'inverse de ses confrères SHA-1 et SHA-2, le principe de celui-ci est complètement différent ce qui est d'autant plus intéressant. Ceci est d'autant plus important car étant donné que des failles de sécurités ont été trouvées dans SHA-1 et que le principe de SHA-2 est le même, celui-ci pourrait à terme, être obsolète à son tour.

Dans le contexte d'une application de transfert de fichiers, SHA-3 avec une longueur de hash de 256 bits a été préféré à SHA-512 pour plusieurs raisons. Tout d'abord, l'option SHA-3-256 a été choisie en raison de son niveau de sécurité jugé adéquat pour la plupart des applications de transfert de fichiers. Bien que SHA-512 fournisse une longueur de hash plus grande, offrant ainsi une sécurité supplémentaire, la longueur de hash de 256 bits fournie par SHA-3-256 est généralement considérée comme suffisante pour garantir l'intégrité des fichiers transférés. En outre, SHA-3-256 présente l'avantage d'être une norme de hachage moderne et approuvée par le NIST, ce qui garantit sa robustesse et sa résistance aux attaques connues. En utilisant une norme de hachage plus récente comme SHA-3, il est assuré que l'application bénéficie des dernières avancées en matière de sécurité cryptographique, réduisant ainsi les risques associés à l'utilisation de versions plus anciennes ou potentiellement vulnérables des algorithmes de hachage. Enfin, bien que SHA-512 puisse offrir une sécurité supplémentaire, il convient de noter que cela peut entraîner un surdimensionnement pour une application de transfert de fichiers où une longueur de hash de 512 bits peut ne pas être nécessaire. En utilisant SHA-3-256, il est évité ce gaspillage de ressources tout en assurant un niveau de sécurité adéquat pour les besoins de l'application de transfert de fichiers. En résumé, SHA-3 avec une longueur de hash de 256 bits a été choisi pour l'application de transfert de fichiers en raison de son niveau de sécurité jugé adéquat, de son statut de norme moderne et approuvée, et de sa capacité à éviter un surdimensionnement des ressources tout en maintenant l'intégrité des fichiers transférés.

Ainsi, la fonction a calculé le hash du fichier avant son chiffrage côté serveur, puis côté client, une fois la réception du fichier, il suffit de faire le procédé inverse, le déchiffrage de celui-ci, puis de calculer le hash. Comme dit précédemment, étant donné qu'une fonction de hachage est déterministe, il suffit de comparer les hash. Si ceux-ci sont différents c'est que le contenu du fichier a été altéré, à l'inverse, si le hash est le même c'est que le fichier reçu correspond bien au fichier émis, et peut par conséquent être sauvegardé, dans le cas échéant, le fichier a été altéré et sera supprimé immédiatement côté client.

## Sécurisation du transfert de fichiers

Le chiffrement AES d'un fichier fonctionne comme suit : la fonction applique sur la donnée à chiffrer, 4 fonctions distinctes dans un ordre précis et de façon répétitive, chacune étant réversibles. La clé de chiffrement est utilisée pour chiffrer mais aussi pour déterminer le nombre de répétition. La clé utilisée pour AES dans le programme fera 32 bytes (256 bits), c'est-à-dire 14 répétitions. Les clés de 128 bits sont plus faibles que celles de 192, qui sont, elles-mêmes plus faibles que celle de 256 bits. Cela est justement dû au Key Schedule (création de sous-clés), il permet de contrer certaines attaques, et sa construction implique qu'une clé de plus grande taille est plus sûre.

Dans le cas de notre clé de 256 bits, il y a 14 répétitions de l'algorithme. Il faut donc générer 14 clés supplémentaires (une clé étant utilisée avant le début des répétitions).

Ainsi, c'est pour ces raisons que le choix s'est porté sur une fonction ayant pour taille une clé de 256 bits.

Pour le transfert de fichier, il a donc suffi de chiffrer le contenu du fichier avec une clé AES, puis de l'envoyer au client. Il est également important d'envoyer la clé AES en la chiffrant avec la clé publique du client pour qu'il puisse par la suite déchiffrer le contenu. Le fait de chiffrer la clé permet d'empêcher le décryptage du fichier si une personne écoute la communication entre le serveur et le client.

# Instructions pour l'exécution du script

L'installation d'un seul package est requis pour faire fonctionner le script, pycryptodome.

Pour ce faire taper dans le terminal « *pip install pycryptodome* » ou « *pip3 pycryptodome* » en fonction de la version de votre installeur pip.

Pour lancer le script :

- Il suffit de prendre deux fenêtres de terminal distinctes et faire dans cet ordre-ci (adaptez la ligne en fonction de votre répertoire courant) : *python serveur.py*
- Quant au second terminal entrez : *python client.py*

Quelques modifications ont été effectués sur le lancement du serveur. Précédemment si la personne lançait trop rapidement les requêtes de serveur étant donné que l'allocation du port était fixe, le terminal renvoyait une erreur en indiquant que le port était déjà utilisé. Désormais, en ayant modifié légèrement l'initialisation celle-ci va tout simplement permettre la réutilisation de l'adresse dans le cas où celle-ci est déjà prise. Ainsi, il n'y aura plus aucun risque de problème lié à cela.

```
class Server:
    def __init__(self, port):
        self.port = port
        self.server_socket = socket.socket()  # get instance
        self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        host = socket.gethostname()
        self.server_socket.bind((host, port))  # bind host address and port toget
        self.conn = None
```

# Sécurité supplémentaire et ajouts supplémentaires

Afin d'éviter des envois de fichiers bien trop importants, il est primordial de vérifier la taille de ceux-ci, à partir d'environ 500MB, les transferts mettant plusieurs minutes, ils sont considérés comme bien trop long et surchargeraient notre serveur. Par conséquent, il a été décidé de les bloquer.

```
# Vérifie la taille du fichier
file_size = os.path.getsize(filename)
if file_size > 500 * 1024 * 1024:  # 500 MB en octets
    print()
    raise Exception("Votre fichier a une taille trop importante. Veuillez la modifier.")
```

En plus de cela quelques ajouts ont été faits pour rendre l'application de transfert plus utilisable comme le temps total du transfert ou encore une barre de progression pour les longs fichiers.

```
Key decrypted successfully.
Message received.

Expected file size: 1407954
Progress: [##################################################] 100%
File received successfully.
File saved.
Fichier décrypté avec succès et modifié directement: output/img.jpg
Les valeurs de hachage correspondent. L'intégrité du fichier est vérifiée.
----------------------------------------------------------------------
Temps total pour réception du fichier : 0.8487718105316162
----------------------------------------------------------------------
```

# Diagramme de séquence