# ASI331 - TP7 - Examen
# À rendre avant le 29 octobre 2020 à 23h
# scrypt

Quentin Byron

22 octobre 2020

Ce qui est attendu :
- Un code en C/C++ ou Python commenté, cohérent et aéré ;
- Un fichier README qui explique comment compiler et exécuter le code, ainsi que les réponses aux questions ;
- Une archive qui porte votre nom, contenant tous vos fichiers ;
- Un rendu avant le 29 octobre 2020 à 23h, par mail à `byron.ensta@gmail.com` ;
- Aucune bibliothèque extérieure ne peut-être utilisée sauf si spécifié autrement ;
- Les jeux de tests devront être également implémentés.

## 1 Définitions

L'objectif de ce TP sera de développer une fonction `scrypt` qui permet de dériver une clé à partir d'un mot de passe.
Il faut cependant complexifier cette dérivation de mot de passe pour éviter les attaques par force brute. Pour s'en protéger, on va pouvoir influer différents facteurs, pour augmenter le coût de la dérivation en :
- temps de calcul ;
- espace mémoire ;
- niveau de parallélisation.

Pour y parvenir, `scrypt` utilise différents mécanismes sont présentés ci-dessous. Cette fonction de dérivation de mot de passe est utilsée entre autre dans le chiffrement des données utilisateur sur Android, mais également comme *proof of work* dans certaines crypto-monnaies.

### 1.1 `uint32_t`

Nous allons travailler avec des entiers de 32 bits non signés.
En C/C++, cela correspond à des `uint32_t`, en Python il faut gérer le dépassement en faisant un *et bit-à-bit* : `x & 0xffffffff`.
La **somme** de deux mots $u, v$ est $u + v \mod 2^{32}$, et est noté $u + v$.
Le **ou-exclusif** de deux mots $u, v$ est noté $u \oplus v$ ou `u ^ v`.
La **rotation à gauche de $c$-bits** d'un mot $u$ est noté $u \lll c$ ou `R(x, c)`.

## 1.2 Salsa20/8

Soit $s$, un tableau de 16 `uint32_t`, la fonction Salsa20/8 est définie ainsi :

```
for (i = 0; i < 4; ++i)
   s[ 4] ^= R(s[ 0]+s[12], 7);  s[ 8] ^= R(s[ 4]+s[ 0], 9);
   s[12] ^= R(s[ 8]+s[ 4],13);  s[ 0] ^= R(s[12]+s[ 8],18);
   s[ 9] ^= R(s[ 5]+s[ 1], 7);  s[13] ^= R(s[ 9]+s[ 5], 9);
   s[ 1] ^= R(s[13]+s[ 9],13);  s[ 5] ^= R(s[ 1]+s[13],18);
   s[14] ^= R(s[10]+s[ 6], 7);  s[ 2] ^= R(s[14]+s[10], 9);
   s[ 6] ^= R(s[ 2]+s[14],13);  s[10] ^= R(s[ 6]+s[ 2],18);
   s[ 3] ^= R(s[15]+s[11], 7);  s[ 7] ^= R(s[ 3]+s[15], 9);
   s[11] ^= R(s[ 7]+s[ 3],13);  s[15] ^= R(s[11]+s[ 7],18);
   s[ 1] ^= R(s[ 0]+s[ 3], 7);  s[ 2] ^= R(s[ 1]+s[ 0], 9);
   s[ 3] ^= R(s[ 2]+s[ 1],13);  s[ 0] ^= R(s[ 3]+s[ 2],18);
   s[ 6] ^= R(s[ 5]+s[ 4], 7);  s[ 7] ^= R(s[ 6]+s[ 5], 9);
   s[ 4] ^= R(s[ 7]+s[ 6],13);  s[ 5] ^= R(s[ 4]+s[ 7],18);
   s[11] ^= R(s[10]+s[ 9], 7);  s[ 8] ^= R(s[11]+s[10], 9);
   s[ 9] ^= R(s[ 8]+s[11],13);  s[10] ^= R(s[ 9]+s[ 8],18);
   s[12] ^= R(s[15]+s[14], 7);  s[13] ^= R(s[12]+s[15], 9);
   s[14] ^= R(s[13]+s[12],13);  s[15] ^= R(s[14]+s[13],18);
```

## 1.3 `scryptBlockMix`

Entrée : Un tableau $B$ contenant $2 \cdot r$ tableaux de 16 `uint32_t`

1. $Y_{-1} = B_{2 \cdot r - 1}$
2. $\forall i \in [0, N-1], Y_i = \texttt{Salsa20\_8}(Y_{i-1} \oplus B_i)$
3. return $(Y_0, Y_2, \cdots, Y_{2 \cdot r - 2}, Y_1, Y_3, \cdots, Y_{2 \cdot r - 1})$

Sortie : Un tableau contenant $2 \cdot r$ tableaux de 16 `uint32_t`

## 1.4 `scryptROMix`

Entrée : Un tableau $B$ contenant $2 \cdot r$ tableaux de 16 `uint32_t`

1. $V_0 = B$
2. $\forall i \in [1, N-1], V_i = \texttt{scryptBlockMix}(V_{i-1})$
3. $X = V_{N-1}$
4. $N$ fois, $X = \texttt{scryptBlockMix}(X \oplus V_{X[2 \cdot r - 1][0] \bmod N})$
5. return $X$

Sortie : Un tableau contenant $2 \cdot r$ tableaux de 16 `uint32_t`

## 1.5 PBKDF2

PBKDF2 (Password Based Key Derivation Function v2) est une fonction qui permet d'obtenir une sortie de la taille souhaitée. Elle s'utilise avec une fonction à clé, ici HMAC-SHA256. On peut également rendre son exécution plus longue en augmentant le nombre d'itération, mais nous n'utiliserons pas cette propriété, et feront une seule itération.

### 1.6 `scrypt`

Entrée : Un mot de passe $P$, un sel $S$, des entiers $p$, $r$, $N$ et $cLen$

1. $B_0 \parallel B_1 \parallel ... \parallel B_{p-1}$ = PBKDF2-HMAC-SHA256($password = P, salt = S, iter = 1, dkLen = p \cdot 128 \cdot r$)
2. $\forall i \in [0, p-1], B_i =$ `scryptROMix`$(r, B_i, N)$
3. return PBKDF2-HMAC-SHA256($password = P, salt = B_0||B_1||...||B_{p-1}, iter = 1, dkLen = cLen$)

Sortie : Un tableau de $cLen$ octets

### 1.7 Padding PKCS#7

Le padding PKCS#7 consiste à compter le nombre $k$ d'octets manquant pour remplir le dernier bloc de message à chiffrer, et de le compléter avec $k$ octets de valeur $k$. S'il ne manque pas d'octet, alors on complète par un bloc complet de la même manière.
Ainsi, pour AES, les paddings possible sont (notation hexadécimale) :

- `XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX 01`
- `XX XX XX XX XX XX XX XX XX XX XX XX XX XX 02 02`
- `XX XX XX XX XX XX XX XX XX XX XX XX XX 03 03 03`
- ...
- `10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10`

## 2 A faire

1. Ecrire une fonction `Salsa20/8`, qui prend en entrée un tableau $s$ de 16 `uint32_t` et qui retourne un tableau de même type.
2. Ecrire une fonction `scryptBlockMix`, qui prend en entrée $r$ et un tableau B contenant $2 \cdot r$ tableaux de 16 `uint32_t`, et retourne un tableau de même type.
3. Ecrire une fonction `scryptROMix`, qui prend en entrée $r$, $N$ et un tableau B contenant $2 \cdot r$ tableaux de 16 `uint32_t`, et retourne un tableau de même type.
4. Ecrire une fonction `scrypt`, qui prend en entrée un mot de passe, un sel, $N$, $r$, $p$, $dklen$ et qui retourne un buffer de taille $dklen$.
5. Ecrire une fonction `pad`, qui ajoute un padding PKCS#7 à un message.
6. Ecrire une fonction `unpad`, qui vérifie et enlève un padding PKCS#7 à un message.
7. Ecrire une fonction `generate_salt`, qui génère un aléa de 16 octets.
8. Ecrire une fonction qui chiffre en AES-256-CBC des données à partir d'un mot de passe. Le sel sera généré avec la fonction `generate_salt`, la fonction retournera out, définie ainsi (les tailles indiquées sont en octets) :
   $\underbrace{\text{key}}_{\texttt{0x20}} \parallel \underbrace{\text{iv}}_{\texttt{0x10}} =$ `scrypt`$(\text{password}, \text{salt}, N = 2^{12}, r = 2^3, p = 2^1, dklen =$ `0x30`$)$
   $\text{out} = \underbrace{\text{salt}}_{\texttt{0x10}} \parallel \text{AES-256-CBC}(\text{key}, \text{iv}, \text{pad}(\text{data}))$
9. Ecrire une fonction qui déchiffre les données chiffrées avec la fonction précédente

10. Ecrire une fonction `main` qui va chiffrer ou déchiffrer un fichier à partir d'un mot de passe.

```
./encryptor enc plainfile encfile
./encryptor dec encfile plainfile
```

# 3 Bibliothèque extérieure

## 3.1 C/C++

Vous utiliserez la librairie OpenSSL (ajouter l'option de compilation -lcrypto), notamment les fonctions suivantes :

```c
#include <openssl/evp.h>

// PBKDF2-HMAC-SHA256
const EVP_MD *EVP_sha256(void)
int PKCS5_PBKDF2_HMAC(const char *pass, int passlen,
                      const unsigned char *salt, int saltlen, int iter,
                      const EVP_MD *digest,
                      int keylen, unsigned char *out)

// Random data generation
int RAND_bytes(unsigned char *buf, int num)

// AES-256-CBC
EVP_CIPHER_CTX *EVP_CIPHER_CTX_new(void)
void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *ctx)

int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding)

EVP_CIPHER *EVP_aes_256_cbc(void)

int EVP_CipherInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                   const unsigned char *key, const unsigned char *iv, int enc)
int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                     int *outl, const unsigned char *in, int inl)
int EVP_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl)
```

## 3.2 Python

Vous utiliserez les modules `PyCrypto`, `hashlib` et `secret`, notamment les fonctions suivantes :

```python
import hashlib
import secret
import Crypto.Cipher.AES

hashlib.pbkdf2_hmac(hash_name, password, salt, iterations, dklen=None)
secret.token_bytes(nbytes=None)
Crypto.Cipher.AES.new()
```