Write-Up Merry & Pippin

Pseudo : xbquo

1. TL; DR

Ces challenges sont issus des épreuves de préselection pour rejoindre l'équipe de France de Cybersécurité lors du challenge européen. Ils étaient évalués à 500 et 200 points, ce qui est relativement élevé. Dans le premier challenge, on nous donne un script Python d'un serveur qui nous demande en entrée les matrices permettant de retrouver une certaine clé. Le second challenge est basé sur le même principe excepté le fait que le nombre de requêtes sur le serveur distant est limité.

2. Analyse du code source

Je vous donne d'abord un exemplaire du code source tel que je l'ai commenté, puis je vous expliquerai en détail les lignes importantes qui nous intéressent. Voici le code source :

```
1  import sys
2  import numpy as np
3  from flag import flag
4  from zlib import compress, decompress
5  from base64 import b64encode as b64e, b64decode as b64d
6
7  class Server:
8      def __init__(self, q, n, n_bar, m_bar):
9          # initialisation de constantes
10         self.q     = q
11         self.n     = n
12         self.n_bar = n_bar
13         self.m_bar = m_bar
14         # on initialise des matrices
15         # dans les 2 premiers cas, elles comportent des chiffres entre -1 et
       2
16         # et sont de tailles n*n_bar (n lignes et n_bar colonnes)
17         self.__S_a = np.matrix(np.random.randint(-1, 2, size = (self.n, self
       .n_bar)))
18         self.__E_a = np.matrix(np.random.randint(-1, 2, size = (self.n, self
       .n_bar)))
19         # dans le dernier cas, elle comporte des nb random de 0 a q
20         # et la matrice est de taille n*n
21         self.A     = np.matrix(np.random.randint( 0, q, size = (self.n, self
       .n)))
22         # equivalent de l'operation % entre 2 nombres en python
23         # ici on fait A*__S_a + __E_a mod q
24         # Exemple avec 2 matrices 2x2 :
```

```
        """
        >>> m1
        matrix([[4, 7],
                [4, 7]])
        >>> m2
        matrix([[2, 3],
                [3, 2]])
        >>> np.mod(m1,m2)
        matrix([[0, 1],
                [1, 1]])
        """
        # Soient nos matrices m1 et m2, de coordonn\ es  m1{_i,j} (resp. m2{_i,j})
        # Soit mf notre matrice finale. mf = mod(m1,m2) :
        # On realise m1_{1,1} % m2_{1,1} = mf_{1,1} etc
        self.B      = np.mod(self.A * self.__S_a + self.__E_a, self.q)

    ### Private methods
    def __decode(self, mat):
        """[summary]

        Args:
            mat (matrice)

        Returns:
            matrice: tous les nombres de la matrices sont passes dans recenter puis mult_and_round
        """

        # fonction qui prends un x et le renvoie uniquement s'il est compris  entre 0 et q//2
        # sinon on renvoie la diff rence de x - q (comprise entre -q//2 et +\infty)
        def recenter(x):
            if x > self.q // 2:
                return x - self.q
            else:
                return x

        # fonction qui prends un nb en params et retourne la valeur entiere la plus proche de
        # x / (q/4)
        def mult_and_round(x):
            return round((x / (self.q / 4)))

        # pour les nombres de la matrice mat, on les passe 1 a 1 dans la fonction recenter
        out = np.vectorize(recenter)(mat)
        # pour les nombres de la matrice out, on les passe 1 a 1 dans la fonction mult_and_round
        out = np.vectorize(mult_and_round)(out)
        return out

    def __decaps(self, U, C):
        # pour recapituler, on prends les params U et C de la fonction
```

```
73          # np.dot(U, self.__S_a) ; cette ligne effectue le produit des
    matrices U et __S_a
74          # notons P ce produit. On le soustrait a la matrice C, on note S ce
    r suktat
75          # on fait alors S mod q
76          key_a = self.__decode(np.mod(C - np.dot(U, self.__S_a), self.q))
77          return key_a
78
79      ### Public methods
80      def pk(self):
81          return self.A, self.B
82
83      def check_exchange(self, U, C, key_b):
84          """[summary]
85
86          Args:
87              U ([matrice])
88              C ([matrice])
89              key_b ([boll en]): True si key_a est  gal  a key_b, False sinon
90
91          Returns:
92              [type]: [description]
93          """
94          key_a = self.__decaps(U, C)
95          return (key_a == key_b).all()
96
97
98      def check_sk(self, S_a, E_a):
99          """[summary]
100
101         Args:
102             S_a ([matrice])
103             E_a ([matrice])
104
105         Returns:
106             [bool en]: True si S_a = __S_a et E_a == __E_a, False sinon
107         """
108         return (S_a == self.__S_a).all() and (E_a == self.__E_a).all()
109
110 def menu():
111     print("Possible actions:")
112     print("  [1] Key exchange")
113     print("  [2] Get flag")
114     print("  [3] Exit")
115     return int(input(">>> "))
116
117 if __name__ == "__main__":
118
119     q     = 2 ** 11
120     n     = 280
121     n_bar = 4
122     m_bar = 4
123     flag = ""
124
125     # cr ation du serveur avec ces params
```

```
    server = Server(q, n, n_bar, m_bar)

    # recevoir les matrices A et B
    A, B = server.pk()
    # on nous donne les matrices A et B
    print("Here are the server public parameters:")
    print("A = {}".format(b64e(compress(A.tobytes())).decode()))
    print("B = {}".format(b64e(compress(B.tobytes())).decode()))

    nbQueries = 0
    while True:
        try:
            choice = menu()
            # si on veut faire un  change  de  cl
        if choice == 1:
            nbQueries += 1
            print("Key exchange #{}".format(nbQueries), file = sys.stderr)
            #  entr e des deux matrices U et C et de la  cl  b
            U     = np.reshape(np.frombuffer(decompress(b64d(input("U = "))), dtype = np.int64), (m_bar, n))
            C     = np.reshape(np.frombuffer(decompress(b64d(input("C = "))), dtype = np.int64), (m_bar, n_bar))
            key_b = np.reshape(np.frombuffer(decompress(b64d(input("key_b = "))), dtype = np.int64), (m_bar, n_bar))
            # si c'est  gal  a la  cl  du serveur, il print le msg de succ s , sinon il envoie celui de fail.
            if server.check_exchange(U, C, key_b):
                print("Success, the server and the client share the same key!")
            else:
                print("Failure.")

            elif choice == 2:
                S_a = np.reshape(np.frombuffer(decompress(b64d(input("S_a = "))), dtype = np.int64), (n, n_bar))
                E_a = np.reshape(np.frombuffer(decompress(b64d(input("E_a = "))), dtype = np.int64), (n, n_bar))
                # si les matrices de d part S_a et E_a sont  gales celles du programme, on obtient le flag
                if server.check_sk(S_a, E_a):
                    print("Correct key, congratulations! Here is the flag: {}".format(flag))
                else:
                    print("Sorry, this is not the correct key.")
                    print("Bye bye.")
                    exit(1)

            elif choice == 3:
                print("Bye bye.")
                break

        except:
            pass
```

L'énoncé du premier challenge est le suivant: ***Un serveur a été conçu pour utiliser un algorithme d'échange de clés avec ses clients. Cet algorithme génère et garde le***

même bi-clé pour plusieurs requêtes. Il notifie aussi ses clients quand l'échange a échoué et que la clé partagée n'est pas la même. Votre but est de retrouver la clé secrète du bi-clé généré par le serveur. Service : nc challenges1.france-cybersecurity-challenge.fr 2001.

L'algorithme dispose d'une classe contenant plusieurs fonctions. Celles dont nous prêterons le plus attention seront les fonctions **decode** et quelques lignes de la fonction **__init**. Le challenge consiste à renvoyer deux matrices E_a et S_a qui contiennent des valeurs générées aléatoirement appartenant à l'ensemble $\{-1, 0, 1\}$, à l'aide d'un service permettant de vérifier si nos clés sont égales. Venons-en maintenant à la génération de la clé du serveur. Comme le montre le bout de code suivant :

```
key_a = self.__decode(np.mod(C - np.dot(U, self.__S_a) self.q))
```

La clé key_a, notée ici matrice $k_{a_{4,4}}$ est générée à partir d'une matrice $U_{4,280}$, d'une matrice $C_{4,4}$ et de la matrice que l'on souhaite retrouver, $S_{a_{280,4}}$. Nous noterons D la fonction **decode**. Le serveur réalise donc le calcul suivant :

$$
\begin{aligned}
k_{a_{4,4}} &= D\left(C_{4,4} - U_{4,280} \cdot S_{a_{280,4}} \mod q\right) \\
&= D\left(\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{2,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix} - \begin{pmatrix} u_{1,1} & \cdots & u_{1,280} \\ \vdots & \ddots & \vdots \\ u_{4,1} & \cdots & u_{4,280} \end{pmatrix} \cdot \begin{pmatrix} s_{1,1} & \cdots & s_{1,4} \\ \vdots & \ddots & \vdots \\ s_{280,1} & \cdots & s_{280,4} \end{pmatrix} \mod q\right)
\end{aligned}
$$

Que fait exactement la fonction **decode** ? Regardons le code source de plus près. On initialise deux fonctions que voici :

```
def recenter(x):
    if x > self.q // 2:
        return x - self.q
    else:
        return x

def mult_and_round(x):
    return round((x / (self.q / 4)))
```

La première est assez explicite, on renvoie x, ou $x - q$ si $x > E\left(\frac{q}{2}\right)$, où $E()$ désigne la partie entière. La seconde fonction renvoie l'entier le plus proche de $\frac{x}{\frac{q}{4}} = \frac{4x}{q}$. Cela sera important pour la suite. Ensuite, les deux lignes suivantes de la fonction **decode** s'occupent de réaliser les opérations pour chaque coefficient de la matrice. Prenons par exemple une matrice M aux coefficients suivants :

$$
M = \begin{pmatrix} 100 & 200 & 300 & 400 \\ 500 & 600 & 700 & 800 \\ 900 & 1000 & 1100 & 1200 \\ 1300 & 1400 & 1500 & 1600 \end{pmatrix}
$$

Si on la passe dans la fonction D, toutes les valeurs de la matrice vont d'abord passer dans la fonction **recenter** :
- pour les valeurs inférieurs à 1024 ($\frac{2048}{2}$, car $q = 2048$), on retourne ces valeurs.
- pour les valeurs supérieurs à 1024, on retourne $x - 2048$. Dans notre matrice, les valeurs comprises entre 1100 et 1600, qu'on note x_i, renverront $x_i - 2048$.

Après être passé dans la fonction **recenter**, on obtient la matrice suivante :

$$M_1 = \begin{pmatrix} 100 & 200 & 300 & 400 \\ 500 & 600 & 700 & 800 \\ 900 & 1000 & -948 & -848 \\ -748 & -648 & -548 & -448 \end{pmatrix}$$

Ensuite, il faut encore que les valeurs de la matrice passent dans la fonction **mult_and_round**. Pour tous les coefficients $a_{i,j}$ de la matrice, on calcule $round\left(\frac{4a_{i,j}}{2048}\right)$. On constate que les petites valeurs tel que $a_{i,j} \leq 255$, donneront 0 car si on prend par exemple 100, $\frac{4*100}{2048} = \frac{400}{2048} < \frac{1024}{2048} = 0.5$, la fonction *round* arrondira les valeurs à 0 (l'entier le plus proche de notre quotient). Il en va de même pour les valeurs comprises entre -255 et 0. Donc quand les valeurs $a_{i,j}$ sont dans $[\![-255, 255]\!]$, la fonction D renverra des valeurs nulles, donc inexploitables. Cependant, si les coefficients $a_{i,j}$ sont ≤ -256 ou ≥ 256, la fonction D renverra des coefficients < 0 (resp. > 0). Ainsi, après que la matrice M_1 soit passée dans la fonction **mult_and_round**, on obtient cette dernière matrice :

$$M_2 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 2 & 2 & -2 & -2 \\ -1 & -1 & -1 & -1 \end{pmatrix}$$

Sachant que parmi ces valeurs, on doit retrouver les valeurs des coefficients de la matrice S_a. Maintenant, vous êtes parés pour attaquer le serveur ! L'attaque se déroule en 2 parties pour pouvoir récupérer le flag.

3. Leaker S_a

Regardons maintenant les services dont nous disposons:

```
if choice == 1:
    nbQueries += 1
    print("Key exchange #{}".format(nbQueries), file = sys.stderr)
    # entrée des deux matrices U et C et de la clé b
    U     = np.reshape(np.frombuffer(decompress(b64d(input("U = "))), dtype = np.int64), (m
    C     = np.reshape(np.frombuffer(decompress(b64d(input("C = "))), dtype = np.int64), (m
    key_b = np.reshape(np.frombuffer(decompress(b64d(input("key_b = "))), dtype = np.int64)
    # si c'est égal a la clé du serveur, il print le msg de succès, sinon il envoie celui d
    if server.check_exchange(U, C, key_b):
        print("Success, the server and the client share the same key!")
    else:
        print("Failure.")
```

On peut rentrer les valeurs des matrices $U_{4,280}$, $C_{4,4}$ et $k_{b_{4,4}}$. On contrôle donc entièrement l'input de la clé b, qui sera comparé à celle de la clé a. Qui plus est, on maîtrise partiellement la génération de la clé a, qui dépend notamment des matrices $U_{4,280}$ et $C_{4,4}$. On rappelle la relation qui relie $k_{a_{4,4}}$, $U_{4,280}$ et $C_{4,4}$:

$$k_{a_{4,4}} = D\left(C_{4,4} - U_{4,280} \cdot S_{a_{280,4}} \mod q\right)$$

Etant donné qu'on cherche S_a, on peut prendre pour nous faciliter la tâche une matrice nulle pour $C_{4,4}$. On considère à présent l'égalité suivante :

$$k_{a_{4,4}} = D\left(-U_{4,280} \cdot S_{a_{280,4}} \mod q\right)$$

On sait aussi que la fonction **D** renvoie 0 pour des valeurs dans $[\![-255, 255]\!]$. Par ailleurs, les valeurs de S_a appartenant à $\{-1, 0, 1\}$, il faut que les valeurs des coefficients $u_{i,j}$ soit ≥ 255 ou ≤ -255. Lors de ma résolution, j'ai pris le premier nombre qui me donnait des valeurs intéressantes, sans pour autant considérer que je pouvais prendre une valeur plus intelligente. Vous allez le constater dans la suite de ce rapport. En effet, en ayant testé dans l'interpréteur Python, je trouve ceci :

```
>>> C
matrix([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
>>> U
matrix([[   0., 1023.,    0., ...,    0.,    0.,    0.],
        [   0.,    0.,    0., ...,    0.,    0.,    0.],
        [   0.,    0.,    0., ...,    0.,    0.,    0.],
        [   0.,    0.,    0., ...,    0.,    0.,    0.]])
>>> guess__S_a
matrix([[ 1.,  0., -1.,  1.],
        [ 1.,  1., -1.,  1.],
        [ 0.,  0.,  0.,  0.],
        ...,
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
>>> np.mod(C - np.dot(U, guess__S_a),q)
matrix([[1025., 1025., 1023., 1025.],
        [   0.,    0.,    0.,    0.],
        [   0.,    0.,    0.,    0.],
        [   0.,    0.,    0.,    0.]])
>>> decode(np.mod(C - np.dot(U, guess__S_a),q),q)
matrix([[-2., -2.,  2., -2.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
```

J'ai remarqué que si je prenais une valeur de $u_{i,j}$ égal à 1023, pour la deuxième ligne de $\_\_guess\_S_a$ dans mon exemple, les valeurs retournées dans $\mathbf{Z}/2048\mathbf{Z}$ correspondent à 1025 ou 1023 suivant les valeurs des coefficients de la matrice S_a. En effet :

$$\begin{aligned} 0 - 1023 * 1 \mod 2048 &\iff -1023 \mod 2048 \\ &\iff 1025 \mod 2048 \end{aligned}$$

et

$$0 - 1023 * (-1) \mod 2048 \iff 1023 \mod 2048$$

On ne traite pas le cas trivial lorsque le coefficient $S_{a_{i,j}} = 0$. Ainsi, grâce à cette valeur de $u_{i,j}$, on peut leak les valeurs des coefficients de S_a. En effet, lorsque les 1023, 1025, ou 0 sont passés dans la fonction **D**, (**recenter** renvoie respectivement 1023, $1025-2048 = -1023$, 0), on obtient les valeurs 2, -2 et 0. Ainsi, par identification avec la matrice S_a de départ, on remarque les transformations suivantes :
- les 1 deviennent des -2.
- les -1 deviennent des 2.
- les 0 restent des 0.

Pour que le processus prenne un minimum de temps, on va donc leak S_a ligne par ligne. $U_{4,280}$ n'étant pas carré, on ne peut alimentés de matrices identités tel que :

$$U = \begin{pmatrix} u_{1,1} & \cdots & u_{1,280} \\ \vdots & \ddots & \vdots \\ u_{280,1} & \cdots & u_{280,280} \end{pmatrix} \tag{1}$$

$$= \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \text{ puis} \tag{2}$$

$$U = \begin{pmatrix} 0 & \cdots & 0 \\ 0 & 1 \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \text{ ...} \tag{3}$$

En revanche, en utilisant le principe de l'identité, on peut créer une matrice $U_{280,4}$ comme suit :

$$\begin{pmatrix} u_{1,1} & \cdots & u_{1,280} \\ \vdots & \ddots & \vdots \\ u_{4,1} & \cdots & u_{4,280} \end{pmatrix} = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \text{ puis} \tag{4}$$

$$U = \begin{pmatrix} 0 & 1 \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots\cdots & 0 \end{pmatrix} \tag{5}$$

La ligne (1) sera notre première matrice $U_{280,4}$, la ligne (2) la deuxième etc. De cette manière, on leak d'abord la ligne 1 de S_a, puis la ligne 2, jusqu'à la ligne 280. Grâce

à l'envoie de k_b, on peut envoyer des matrices dont la première ligne contient les valeurs $\{-2, 0, 2\}$ et lorsque le check renvoie "Success .....", cela signifie que les valeurs de la ligne de k_b correspondent à $-2S_a$. De cette manière, on remplit petit à petit notre matrice S_a.

4. Calculer E_a

Maintenant que nous avons la matrice S_a, on doit calculer la matrice E_a, pour pouvoir renvoyer ces deux matrices et obtenir le flag. D'apres cette ligne :

```
self.B      = np.mod(self.A * self.__S_a + self.__E_a, self.q)
```

en isolant E_a, il vient :

$$B = A * S_a + E_a \mod q \iff E_a = B - A * S_a \mod q$$

A l'aide de Python, on réalise alors ce calcul et on renvoie les deux matrices S_a et E_a pour obtenir le flag.

5. Script de résolution

```python
from pwn import *
from zlib import decompress, compress
from base64 import b64decode as b64d, b64encode as b64e
import numpy as np
import itertools
import time

r = remote("challenges1.france-cybersecurity-challenge.fr", 2001)
#r = process(['python',  './test.py'])
r.recvline()

n = 280
n_bar = 4
m_bar = 4
q = 2**11

A = np.reshape(np.frombuffer(decompress(b64d(r.recvline()[4:-1])), dtype = np.int64), (n, n))
B = np.reshape(np.frombuffer(decompress(b64d(r.recvline()[4:-1])), dtype = np.int64), (n, n_bar))
print(r.recv())

def decode(matrice,q):
    def recenter(x):
        if x>q//2:
            return x-q
        else:
            return x

    def mult_and_round(x):
        return round((x / (q/4)))

    out = np.vectorize(recenter)(matrice)
```

```
    out = np.vectorize(mult_and_round)(out)
    return out

# Initialisation de nos matrices tests U, C, et key_b.
C = np.asmatrix(np.zeros(shape=(m_bar,n_bar)),dtype=np.int64)
U = np.asmatrix(np.zeros(shape=(m_bar,n)),dtype=np.int64)
key_b = np.asmatrix(np.zeros(shape=(m_bar,n_bar)),dtype=np.int64)
guess__S_a = np.asmatrix(np.zeros(shape=(n,n_bar)),dtype=np.int64)

i = 0
try:
    while i!=280:

        try:
            U[0,i] = 1023
            U[0,i-1] = 0
        except IndexError:
            pass
        r.sendline("1")
        r.recv()
        #print(f"__S_a : {guess__S_a}")


        for comb in list(itertools.product([2, -2, 0], repeat=4)):
            #print("IN THE GU3SSING L00P")
            key_b[0] = list(comb)
            #print(f"Actual combination : {comb}")
            r.sendline(b64e(compress(U.tobytes())).decode())
            #print("Je viens de send U")
            r.recv()
            r.sendline(b64e(compress(C.tobytes())).decode())
            r.recv()
            r.sendline(b64e(compress(key_b.tobytes())).decode())
            response = r.recv()
            #print(response)
            if b"Success" in response:
                #print("=== SUCCESS ===")
                #print(response)
                print(f"[+] {i} line of __S_a contains {comb}")
                guess__S_a[i] = list(comb)
                i+=1
                #time.sleep(0.2)
                break
            else:
                #print("ELSE CONDITION")
                r.sendline("1")
                r.recv()
except IndexError:
    pass

# transformer la matrice avec ses valeurs initiales
with np.nditer(guess__S_a, op_flags=['readwrite']) as it:
    for x in it:
        if x == -2.0:
            x[...] = 1
```

```
        elif x == 2.0:
            x[...] = -1
print(f"__S_a : {guess__S_a}")

__E_a = np.mod(B-np.dot(A,guess__S_a),q)
with np.nditer(__E_a, op_flags=['readwrite']) as it:
    for x in it:
        if x == 2047.0:
            x[...] = -1

print(f"__E_a : {__E_a}")
r.sendline("2")
print(r.recv())
r.sendline(b64e(compress(guess__S_a.tobytes())).decode())
print(r.recv())
r.sendline(b64e(compress(__E_a.tobytes())).decode())
print(r.recv())
```

6. Challenge Pippin

Le challenge Pippin était basé sur un code source similaire, dont on ne disposais cette fois ci, pas. On test alors notre script qui fonctionnait sur Merry, et on remarque qu'on ne peut effectuer que 3000 requêtes pour leak S_a, et avec le script actuel, arrivé à la moitié de la matrice, on ne pourrait plus rien effectuer de requêtes. On regarde donc s'il y a un pattern dans les réponses, et on remarque cela :

```
[+] 0  (x, 0, 0, y)
[+] 1  (0, 0, y, x)
[+] x  (0, y, 0, x)
[+] 3  (y, 0, 0, x)
[+] 4  (y, x, 0, 0)
[+] 5  (y, 0, 0, x)
[+] 6  (x, 0, 0, y)
[+] 7  (0, y, x, 0)
[+] 8  (0, x, 0, y)
[+] 9  (y, 0, 0, x)
[+]10  (0, 0, x, y)
```

Chaque ligne contient exactement deux fois le chiffre 0, ce qui nous permet de réduire drastiquement le nombre de requêtes. En effet, pour le challenge précédent, on testait toutes les combinaisons de $\{-2, 0, 2\}$, tandis que maintenant, on peut simplement dans notre combinaison bruteforce deux valeurs, car il y a deux 0 que l'on connait. On remplace donc cette ligne du script précédent :

```
for comb in list(itertools.product([2, -2, 0], repeat=4)):
```

par la ligne suivante :

```
for comb in set(list(itertools.permutations([2, -2, 0, 0],r=4))):
```

C'est tout pour ce Write-Up, j'espére avoir été clair et si certains points semblent encore flou, n'hésitez pas à me les faire remonter.