



---

UNIVERSITÉ PARIS 8 - VINCENNES À SAINT-DENIS

**Licence informatique & vidéoludisme**

## **Projet Final introduction à la sécurité**

**Maria Messaoud-Nacer**

Date de rendus : le 05/11/2023

Groupe : L3-A





# Rapport Projet CQ

## Résumé

Le présent document représente le rapport du projet Encrypted chat. Ce dernier est une implémentation d'un simple chat sécurisé en réseau.

Dans le premier chapitre, nous découperons ce projet en plusieurs parties que nous détaillons avec ,a certains points ,quelques exemples et illustrations en image avec des parties de code jugées plus ou moins pertinentes .

Dans le chapitre suivant une petite conclusion et un petit bilan du projet, et au final, la bibliographie des sources qui ont aidés à la réalisation de ce projet.

# Table des Matières

Résumé

Table des Matières

Introduction

Les fichiers nécessaires

Chiffrement / Déchiffrement

1.Echange de clés Diffie-Hellman

D'abord , la génération de clés privé (a et b) , fonction assez classique :

Calcul des clés Publique : (assez classique aussi)

Calcul de la clés partagé ( à laquelle j'ai choisis d'appliqué un hash 256 bits)

Fonctions d'échange de clés publique

2.Code de chiffrement/ déchiffrement des message

3.Réseau

4.Résultat

Conclusion et Perspective

# Introduction

Un projet réalisé et écrit entièrement en Python. .

Encrypted\_chat :

- La mission de ce programme consiste à permettre un chat sécurisé entre deux utilisateurs.
- La problématique est donc de mettre en œuvre les bonnes pratiques de sécurité afin que les messages ne puisse être lu que par les deux parties concernées.
- Les difficultés rencontrées ont été notamment l'échange de clés et toutes les bonnes connexion , les bonne pratique d'encodage et décodage des messages entre le serveur et les clients.
- Ce projet est actuellement incomplet et non optimisé, il était initialement prévu avec une interface graphique (Tkinter)

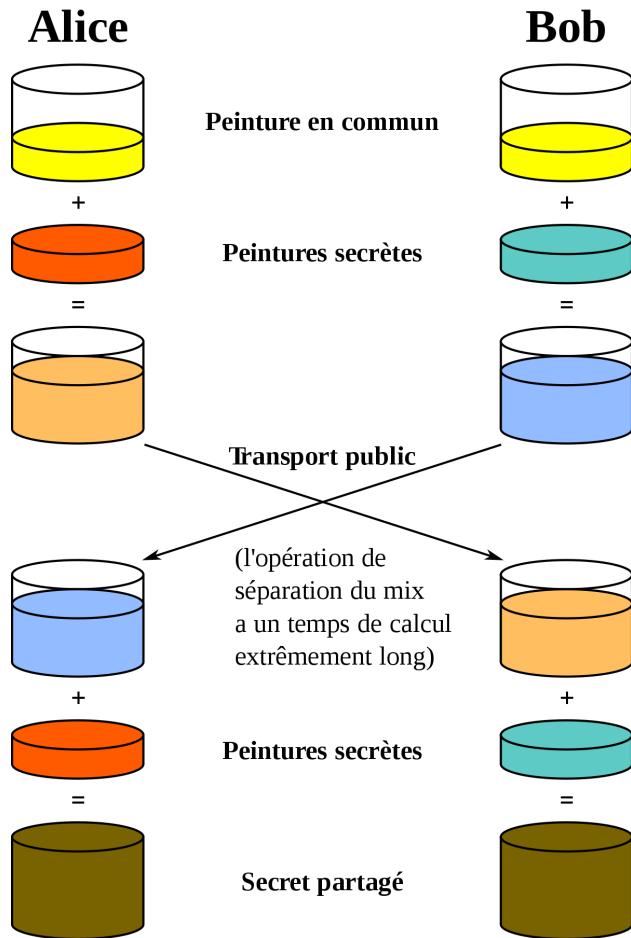
# **Les fichiers nécessaires**

1. client.py Qui gère :
  - a. Envoie et réception de message
  - b. Chiffrement et déchiffrement
  - c. Calcul des clés publique et de la clé partagée
2. server.py Qui gère :
  - a. La connexion entre les client (réception de client x et envoie à y (resp.))
  - b. L'échange de clés Diffie-Hellman
3. DHke.py qui contient le cœur de la clé publique (p et g)

# **Chiffrement / Déchiffrement**

## **1.Echange de clés Diffie-Hellman**

Le protocole d'échange de clés Diffie-Hellman est un protocole pour qui permet à deux parties de s'échanger des informations de manière sécurisée sur un canal non sécurisé (parfait donc pour le projet). C'est ce qu'on appelle un algorithme d'accord de clés, les deux parties se mettant d'accord sur une clé partagée.



- Alice et Bob se mettent d'accord sur 2 nombres : **p** (un très grand nombre premier), et **g** (un autre nombre, appelé générateur). **p** et **g** **sont transmis en clair** sur le réseau.
- Alice et Bob choisissent chacun de leur côté un très grand nombre aléatoire, **qu'ils gardent secret**. Soit **a** le nombre choisi par Alice, et **b** le nombre choisi par Bob.
- Alice calcule et transmet le résultat à Bob de :

$$P_1 = g^a \mod p$$

- Bob calcule et transmet le résultat à Alice de :

$$P_2 = g^b \mod p$$

- Alice calcule:

$$K_1 = P_2^a \mod p = (g^b \mod p)^a \mod p$$

- et Bob calcule:

$$K_2 = P_1^b \mod p = (g^a \mod p)^b \mod p$$

Les lois de l'arithmétique prouvent que les deux valeur **K1 et K2 sont égales**. Alice et Bob sont donc parvenus à se mettre d'accord sur une **clé privée commune**:

$$Key = g^{ab} \mod p$$

Eventuel problème :

- Attaque "Man-in-the-middle" ( mais dans mon cas ça ne m'a pas l'air important)

Dans mon implémentation de cet algorithme , j'ai choisis d'utiliser le p (prime) et le g (generator) des paramètres prédéfinis du groupe **MODP 14** (Modular Exponentiel 2048 bits), un choix que j'ai fait après quelques recherches sur le sujet et après avoir croisé cette pratique sur divers ressources et projets. (2048 bits car j'estime que c'est largement suffisant pour ce qui est demandé )

#### 2048-bit MODP Group

This group is assigned id 14.

This prime is:  $2^{2048} - 2^{1984} - 1 + 2^{64} * \{ [2^{1918} \pi] + 124476 \}$

Its hexadecimal value is:

FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1  
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD  
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245  
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED  
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D  
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F  
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D  
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B  
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9  
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510  
15728E5A 8AACAA68 FFFFFFFF FFFFFFFF

The generator is: 2.

**D'abord , la génération de clés privé (a et b) , fonction assez classique :**

```
def genRandom(bits):  
    bytes = bits // 8 + 1#nb d'octets  
    rand = int(hexlify(os.urandom(bytes)), 16)  
    #rand = random.randint(2, 10)  
    return rand
```

```
sk = genRandom(256)
```

## Calcul des clés Publique : (assez classique aussi)

```
def genPubKey(pk, p, g):
    print("generation de cle publique : g^(a/b) mod p \n")
    return pow(g, pk, p)# g^(private_key) mod p
```

## Calcul de la clés partagé ( à laquelle j'ai choisis d'appliqué un hash 256 bits)

```
#fonction que j'ai trouvé sur plusieurs ressources et que j'ai adapté à mes besoins
def genShared(fk, p, k):
    sharedSecret = pow(fk, k, p)
    _sharedSecretBytes = str(sharedSecret).encode('utf-8')
    s = hashlib.sha256()
    s.update(bytes(_sharedSecretBytes))
    key = s.digest()
    return key
```

## Fonctions d'échange de clés publique

```
def exchange_keys(client1, client2):
    client1.send("PK".encode('utf-8')) #demander Pa = g^a mod p
    client1_public_key = client1.recv(9000).decode('utf-8') #il les récupère dans des variables
    client1_public_key = str(client1_public_key.split()[1])
    #print(f"client1_public_key :{client1_public_key}")
    client2.send("PK".encode('utf-8')) #demander Pb = g^b mod p
    client2_public_key = client2.recv(9000).decode('utf-8')#il les récupère dans des variables
    client2_public_key = str(client2_public_key.split()[1])
    #print(f"client2_public_key :{client2_public_key}")

    #Puis les renvoie comme il faut
    client1.send(f"FK {client2_public_key}".encode('utf-8'))
    client2.send(f"FK {client1_public_key}".encode('utf-8'))
```

Fonction que j'ai pu implémenter facilement , la difficulté était plutôt de l'utiliser correctement , 9000 est un choix arbitraire , juste pour être sûre que ça tienne.

Amélioration que je voulais faire : vérifié si les deux clients ont la même clés partagé pour savoir si un canal est sécurisé ou pas , mais je n'ai pas réussi à trouver comment ...

## 2.Code de chiffrement/ déchiffrement des message

```

import nacl.secret
import nacl.utils

#le code tout autour
#Chiffrement :
box = nacl.secret.SecretBox(Key)
decrypted = box.decrypt(message)
#Dechiffrement
nonce = nacl.utils.random(nacl.secret.SecretBox.NONCE_SIZE)
encrypted = box.encrypt(message.encode(),nonce)
#le code tout autour

```

Pour ça j'ai donc utilisé ma bibliothèque PyNaCl (dite "Python Salt") , je voulais faire une version aussi en AES-256 mais manque de temps , j'ai finis par me servir des méthodes "encrypt" et "decruypt" de PyNaCl (qui utilise l'algorithme Xsalsa20 stream cipher, que j'ai trouvé satisfaisant pour le projet ) en m'aidant de la documentation Python.

Eventuels problèmes :

- Attaque par brut force (très compliquée )
- Attaque par analyse différentielle ou linéaire (très complexes)
- Des attaques attaques matérielles (inutiles de s'en soucier pour ce projet)

### 3.Réseau

Implémentation assez classique avec `socket` et `threading` de python .

Le défit était de réussir à séparer les différentes étapes :

```

def receive():
    session = True
    while True:
        client, address = server.accept()
        print("Nouvelle connection de",str(address))
        client.send("PSEUDO ".encode('utf-8'))
        pseudo = client.recv(9000).decode('utf-8')
        pseudos.append(pseudo)
        clients.append(client)

        print(f"Le pseudo du client est : {pseudo} !")
        broadcast(f"{pseudo} a rejoин la conversation ".encode("utf-8"),client)
        client.send("Vous etes connecté au serveur ! ".encode("utf-8"))
        if len(clients) == 2 and session : #---> determiner une session
            print("session\n")
            exchange_keys(clients[0], clients[1])
            session = False

```

```
thread = threading.Thread(target = handle, args=(client,))
thread.start()
```

```
# Separer les differentes phases
def receive():
    global box
    while True:
        try:
            message = client.recv(9000)
            if message == b"PSEUDO ": # Phase 1
                client.send(str(pseudo).encode('utf-8'))
            elif message == b"PK": # Phase 2
                Pk = pow(g, sk, p)
                public_key = f"Pk {Pk}"
                #print(f"Pk : {Pk}")
                client.send(str(public_key).encode('utf-8'))
            elif message.startswith(b"FK"): # Phase 3
                Fk = int(message.split()[1])
                #print(f"Le serveur a envoyé la valeur de P de l'autre client :")
                #print(Fk)
                # print("sk : ", sk)
                # print(sys.getsizeof(sk))
                Key = genShared(Fk, p, sk)
                #print(f"La clé partagée est : {Key}")
                #print(sys.getsizeof(Key))
                box = nacl.secret.SecretBox(Key)
            else:# Phase 4
                if box is not None:
                    decrypted = box.decrypt(message)
                    print(decrypted.decode())
                else:
                    print(message.decode())
        except Exception as e:
            print(f"Erreur dans le gestionnaire : {e}")
            client.close()
            break
```

## 4.Résultat

Petit test d'exécution avec des prints de ce qu'il se passe , on peut voir nettement que l'échange des clés est bien effectué et que la clé partagé est la même, on peut voir aussi que les messages sont bien chiffrés sur le serveur.

## Conclusion et Perspective

A travers ce projet j'ai beaucoup appris sur les processus de chiffrement et la manipulation de clés dans un réseau , j'ai également pu me familiariser avec les module PyNaCl et les différents groupes conventionnels de Diffie-Hellman qui m'en sont d'une grande utilité

▼ Parmi les liens qui m'ont été utiles :

[https://github.com/vrikodar/Terminal\\_chat](https://github.com/vrikodar/Terminal_chat) (Python)

<https://github.com/Gursimratsingh/Secure-Chat-with-Implementation-of-AES-with-Diffie-Hellman-Key-Exchange> (Js)

<https://github.com/spec-sec/SecureChat> (Python)

<https://github.com/onyeka/SimpleChat> (Python)

<https://gist.github.com/noqqe/cd9f8dc6477c7929f8b3>

Ainsi que les documentations python , des vidéos YouTube et des sites tel que [ce site](#) qui m'a aidé à comprendre le protocole Diffie-hellman .