TP unsecure chat

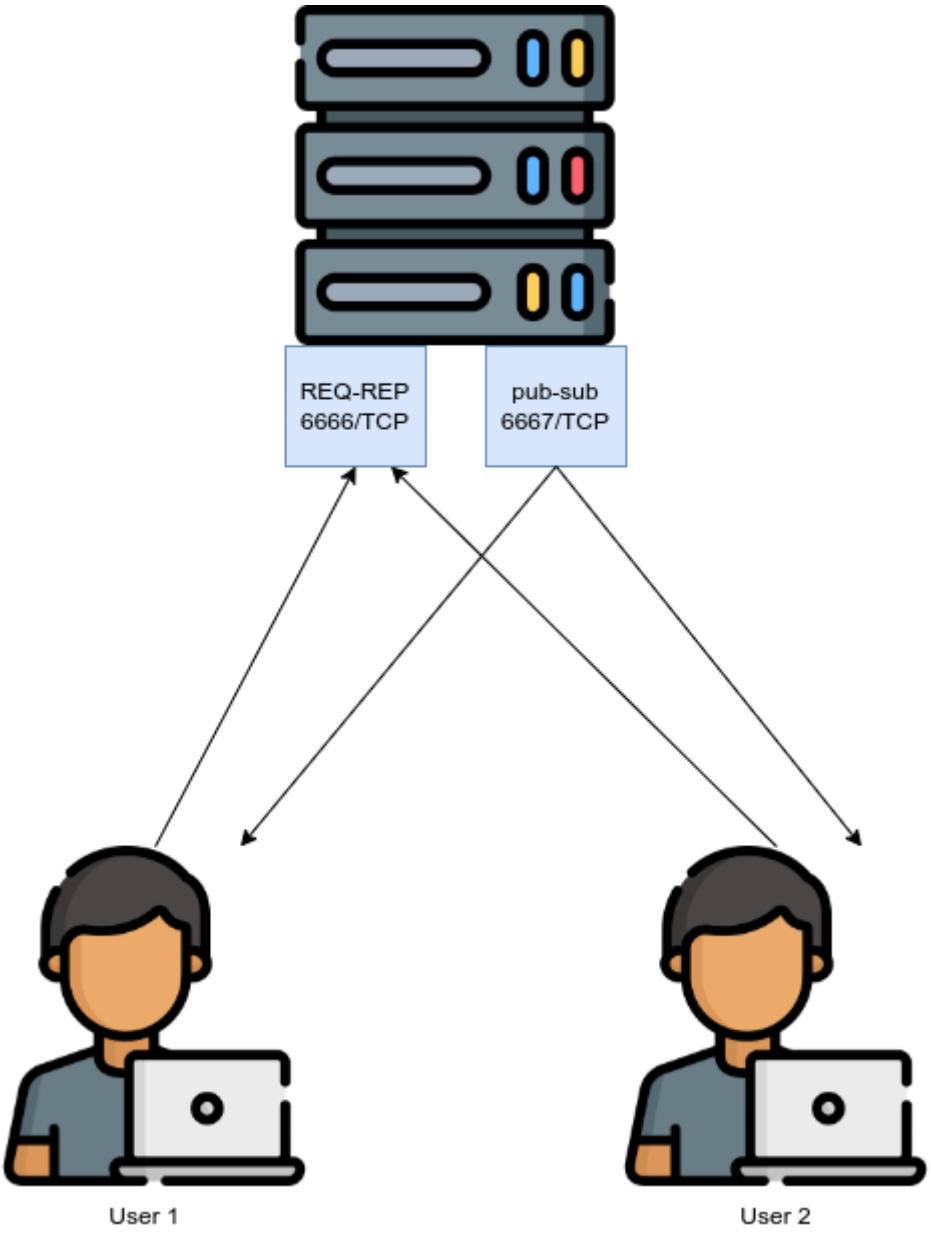
Introduction

Ce TP consiste à sécuriser un chat et à se prémunir contre certaines attaques en utilisant des outils/méthodes adaptés vus en cours.

Les fichiers du TP s'organise comme ceci :

- source
    - source/base_client.py : contient le code pour faire client par composition
    - source/base_server.py : contient le code pour faire server par composition
    - source/simple_client.py : un client chat
    - source/simple_server.py : un serveur chat
- ANSWERS.md : contient vos réponses aux questions de TP
- TP.pdf : donnes les consignes

La vue globale des communications est le suivante :



Le principe est simple et repose sur les socket ZeroMQ :

- Le port 6666/TCP est de type request-response (many to one) et permet d'envoyer les messages de type *join*, *leave*, **list** et *message*. La response faite par le serveur permet de savoir si la commande s'est bien passée ou si des informations sont à retourner.
- Le port 6667/tcp est de type Publish-Submit ce qui correspond à du broadcast (one to many). Ce canal ne sert qu'à broadcaster les messages envoyés par les utilisateurs.

En ignorant la méthode de serialisation, les messages sont des dictionnaires composés par défaut comme ceci :

- join : {"type":"join", "nick":"user nickname"}. Retourne {"response":()"ok"| "ko")}
- leave : {"type":"leave", "nick":"user nickname"}. Retourne {"response":()"ok"| "ko")}
- message : {"type":"message", "nick":"user nickname", "message":"something to say"}. Retourne {"response":()"ok"| "ko")}
- list : {"type":"list"}. Retourne {"response":["user1", "user2"]}

Conseils :

- Soyez curieux
- Lisez la documentation

- Lisez le code fournie
- Utilisez ChatGPT. A bon escient.

Coding rules

Le code devra respecter un certain nombre de règles (arbitraires) pour ne pas perdre de point :

- Le code doit être commenter
- Les noms des fonctions et de variables s'écrivent en en snake case (nom_de_fonction)
- Les noms de constantes s'écrivent en snake case majuscule (NOM_DE_CONSTANTES)
- Les noms de classe s'écrivent en camel case (MaClass)
- Les noms de fichiers reprennent le nom de la classe en snake case minuscule (ma_class)
- Pas de valeurs littérales dans le code, utilisez toujours des constantes pour les expliciter
- Une classe, un fichier
- Une fonction ne doit pas dépasser 20 lignes
- Pas plus de deux imbrications de bloc
- Utilisez des variable intermédiaires lors d'enchaînement de fonction. Exemple à ne pas faire : x = y(z(w(g,h,j,k)))
- Utilisez les log (self._log) plutôt que les print()
- Utilisez Github pour y mettre votre projet

Globalement : Plus je comprends ce que vous avez fait, plus j'ai envie de mettre des points … et inversement.

Installer votre environnement

Pour pouvoir faire le TP vous devez vous trouvez dans un linux (VM, natif, wsl, …).

Voici les commandes permettant de créer votre virtual env dans le répertoire du TP :

- apt update
- apt upgrade -y
- apt install python3-pip python3-venv
- python3 -m venv env
- source env/bin/activate
- pip3 install -r requirements

Prise en main

Exécuter le serveur (`python3 simple_server.py`) ainsi que deux clients (`python3 simple_client.py`) dans 3 terminaux. Vous devez avoir deux onglets de navigateurs d'ouvert, un pour chaque client. Amusez-vous avec, puis …

- Que pensez-vous de la confidentialité des données vis à vis du serveur ?
- Écrivez un script (simple_bigbrother.py, max 40 lignes) qui vous permettra d'écouter toutes les conversations.
- Écrivez un script (simple_dos.py, max 40 lignes) qui vous permettra de créer un déni de service pour les utilisateurs.

- Pouvez vous expliquer en quoi la sérialisation *pickle* est certainement le plus mauvais choix ?
- A partir du code ci dessous, réaliser une exécution arbitraire de code côté serveur (simple_exploite.py) (Remote Code Execution, *RCE*) :

```
import pickle

class MaliciousCode(object):
    def __reduce__(self):
        """
        __reduce__ : This special method is used by pickle to determine how an object should be deserialized. Here, the method returns a tuple containing the function os.system and the argument 'echo "Malicious command executed"'. This means that on deserialization, pickle will execute os.system('echo "Malicious command executed"'), which executes an arbitrary command in the system.
        """
        return (os.system, ('echo "H4cker takes the control !"',))

packet = pickle.dumps(MaliciousCode())

# ooops
pickle.loads(packet)
```

- [BONUS] A partir de l'exemple précédent, faire une *RCE* permettant d'exécuter une reverse shell. Si vous y arrivez, vous êtes le boss ! (reverse_shell.py)
- Quels types de sérialisation pourrait-on utiliser pour éviter cela ? (hors CVE)

Authenticated Encryption

Nous allons ici nous intéresser au chiffrement/intégrité du champ message contenues dans les paquets de type message. Nous utiliserons *msgpack* comme format de sérialisation. Pour le chiffrement et l'intégrité, nous utiliserons l'implémentation de Fernet présente dans la bibliothèque cryptography. Pour le nombre d'itération de la dérivation de clef, on prendra 100 pour l'exercice.

- Pourquoi le chiffrement seul est-il insuffisant ?
- Quelle fonction(s?) en python permet de générer un salt avec une qualité cryptographique ?
- Faudra t-il transmettre le salt comme champ en clair supplémentaire du paquet message ?
- Créer une classe AEServer.py qui va hérité de SimpleServer et ajouter le code nécessaire à son exécution en vous inspirant de simple_server.py. Cette classe doit utiliser msgpack au lieu de pickle, quelles variables faut il amender dans le constructeur ?
- Créer une classe AEClient.py qui va hérité de SimpleClient et ajouter le code nécessaire à son exécution en vous inspirant de simple_client.py. Votre classe doit ressembler à ceci :

```
import logging
import base64
import os
from typing import Tuple

import msgpack
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
```

```
from cryptography.hazmat.primitives import hashes
from cryptography.fernet import Fernet

from pywebio.output import put_text
from names_generator import generate_name

from simple_client import SimpleClient


class AEClient(SimpleClient):
    def __init__(self, host: str, send_port: int, broadcast_port: int, nick:
str, password:str):
        raise NotImplemented()

    def derive_key_from_password(self, password: str, salt: bytes) -> bytes:
        raise NotImplemented()

    def encrypt_message(self, password: str, message: str) -> Tuple[bytes,
bytes]:
        raise NotImplemented()

    def decrypt_message(self, password: str, encrypted_message: bytes, salt:
bytes, nick:str) -> str:
        # var nick is not used right now (further feature)
        raise NotImplemented()

    def send(self, frame:dict)->dict:
        raise NotImplemented()

    def message(self, message:str):
       raise NotImplemented()

    def on_recv(self, packet: bytes):
        # callback of broadcast message
        raise NotImplemented()


if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    client = AEClient("localhost", 6666, 6667, generate_name(),
"Best_Secr3t_ever_!")
    client.run()
```

- Que constatez-vous côté serveur ?
- Que peux faire le serveur si il est malveillant sur les messages ? Réaliser un script (rogue_server.py, 40 lignes max) illustrant cela.

Authenticated Encryption with Associated Data

- Que faudrait-il faire en théorie pour éviter l'action du rogue server ? Pourquoi Fernet n'est pas adapté dans ce cadre ?
- Dans la pratique, quelle solution **simple** et sous optimal peut-on mettre en place afin de conserver Fernet ? Utiliser la variable « nick » présente en paramètre de decrypt_message().

- Créer une classe AEADClient qui va hérité de AEClient et surcharger les fonctions encrypt et decrypt afin d'y intégrer votre solution.
- Est-ce que le rogue serveur fonctionne t-il encore ?

Annexes

Je vous propose ici quelques conseils et réflexions pour réussir votre TP et éveiller votre curiosité.

Règle N°1

En python tout est objet. Un nombre, une string, une fonction, une instance de classe, une classe, un module. Tout.

Héritage

L'héritage est une des facettes de la programmation orientée objet qui consiste à transmettre à une classe B tout ou partie des propriétés d'une classe A. Le python supporte l'héritage simple et multiple (nous omettrons ce dernier).

L'héritage simple en python peut se résumer par cet exemple :

```
class A:
    def __init__(self):
        self._nick = "Toto"
        self._name = self.__class__.__name__

    def f(self):
        print(f"{self._nick} is {self._name} IRL")
        return 2*self.g()

    def g(self):
        return 4

class B(A):
    def __init__(self):
        super().__init__()
        self._nick = "Titi"

    def g(self):
        return 5

    def h(self):
        return super().g() + 2


a = A()
# print Toto is A IRL
# return 8
print(f"A.f() -> {a.f()}")

b = B()
# print Titi is B IRL
# return 10
print(f"B.f() -> {b.f()}")
```

```
# return 6
print(f"B.h() -> {b.h()}")
```

Quelques commentaires sur le code :

- `self.__class__.__name__` est très pratique si vous avez besoin d'instancier un objet dans la classe mère devant porter le nom réel de la classe, y compris en cas d'héritage. C'est typiquement le cas des objets de logging.
- `super()` permet de faire référence directement à la classe mère. Cela sert essentiellement au constructeur (`__init__()`) pour appeler le constructeur de la classe mère, chose qui n'est pas implicite en python. Les déclarations/modifications de variables membres doivent être faites après cet appel. `super()` permet aussi d'appeler des méthodes de la classe mère qui auraient été surchargées (exemple ici : `g()`).

L'héritage dans la vraie vie

La conception orientée objet est relativement ancienne (année 80-90) et a longtemps été considérée comme le paradigme roi. On en retrouve une implémentation particulièrement complète en C++. Il est indéniable cette approche a révolutionnée le mode du développement logiciel. Seulement, quand on a un bon marteau, on veut que tout nos problèmes ressemblent à des clous. Et parfois ce sont parfois vos doigts qui sont en dessous ...

Un des aspects les plus puissants ce paradigme est l'héritage, permettant de transmettre tout ou partie de propriété à un objet. Cela peut être fait d'une façon simple (une classe hérite d'une autre classe) ou multiple (une classe hérite de plusieurs classes). Sur le papier, cela peut sembler séduisant. Mais l'héritage multiple c'est avéré être particulièrement complexe à mettre en place et à maintenir au point que certains langages (java par exemple) n'autorise pas l'héritage multiple. Certains langages ont même tout simplement abandonné cette possibilité, comme le langage Go.

Plusieurs raisons expliquent les limitations pratiques et financières de cette approche :

- Une modification de la classe mère implique une modification de toutes les classes filles. Pour un simple paramètre, vous pouvez casser une API entière, avec ses tests et sa documentation. Personne ne peux se permettre cela !
- Si une fonction attend une instance de classe d'une classe abstraite, il faut forcement dériver cette classe pour appeler la fonction. C'est un enfer pour le code.
- Corollaire du points précédent : cela complexifie énormément les tests. Cela décourage les développeurs d'écrire des tests, il y a des bugs, l'image du logiciel est écornée, celle de l'entreprise aussi. Bref, effet papillon assuré !

Mais python est un excellent langage, qui permet d'éviter ces travers. Tout d'abord, le duck typing est présent en python : « ça a des plumes, ça vol et ça fait couin-couin ? Alors c'est un canard ». Dit autrement, si vous passer une instance de classe à une fonction et que les méthodes quelle doit appelées sont présentes, alors tout va bien. Cela confirme aussi la règle 1. Prenons un exemple :

```
class A:
    def __init__(self):
        pass

    def f(self):
        return 4
```

```
class B:
    def __init__(self):
        pass

    def f(self):
        return 5


def double(x):
    return x.f() * 2

a = A()
print(f"double(a)->{double(a)}")

b = B()
print(f"double(b)->{double(b)}")
```

`double(x)` est particulièrement indifférent au fait que x doit de type A ou B et on apprécie cette qualité !

Vous l'avez compris, l'héritage pose de gros problèmes. L'alternative à cela - qui peut reposer sur le duck typing - est d'utiliser la composition pour remplacer l'héritage. On passe d'une relation « est » à une relation « contient », ce qui diminue le couplage. On peut ainsi facilement tester le code.

```
import time
class A:
    def __init__(self):
        pass

    def f(self):
        return time.time()

class B:
    def __init__(self, _dependancy=A):
        self._c = _dependancy()

    def get_time(self):
        return int(self._c.f())

# normal code usage
print(f"B().get_time()->{B().get_time()}")

class AMock:
    def f(self):
        return 100

# mimic a test
print(f"B(AMock).get_time()->{B(AMock).get_time()} == 100 ?")
```

Le python vous permet de faire de la programmation objet, l'impérative et du fonctionnelle : utilisez ce qui est le plus approprié !

Une clef à partir d'un mot de passe

Le cryptographie est tel qu'il y a un schisme entre l'utilisateur et le système cryptographique.

D'un côté, pour l'utilisateur, un secret partagé est un mot de passe, avec des chiffres, des lettres ou encore des « caractères spéciaux », et une longueur variable. C'est la forme de secret la plus simple à mémoriser pour un être humain (cf 3ième loi des principes de Kerchkoffs).

De l'autre côté, on a des systèmes cryptographiques, basés sur des mathématiques. Ces derniers sont inflexibles au possible. Par exemple, un système de chiffrement nécessitera une clef de longueur fixe, en binaire.

Il faut donc pour cela trouver une passerelle entre ces deux mondes - oh combien différents - pour leur permettre de cohabiter. L'approche la plus naïve est certainement de hacher le mot de passe avec une fonction de hachage tel que le SHA256. Cela permettrai facilement de faire du « key stretching ».

Toutefois, cela pose un certain nombre de problèmes :

- Il existe des rainbow tables permettant de cracker les hashes et de retrouver le mot de passe.
- Pour un même mot de passe, on obtient la même clef. Hors, ce comportement c'est pas souhaitable : on veut utiliser des « salts ». Ce champ aléatoire publique (le connaître ne pose pas de problème de sécurité) permet de faire varier la clef pour un même mot de passe, rendant le crackage bien plus complexe.
- Les attaques par « bruteforce » se réalisent très bien en parallèle sur des GPUs. C'est même le principe intrinsèque du minage de bitcoin !

Pour éviter cela, on utilise généralement des processus itératifs, dont l'état N dépend directement de l'état N-1 : il n'est plus possible de paralléliser. C'est pas exemple le cas du PBKDF2 (PKCS #5 v2). Le nombre d'itération va augmenter le temps pour obtenir la clef à partir du mot de passe. Si une dérivation prend 500 ms, elle sera presque imperceptible alors que pour une attaquant, cela signifie qu'il ne pourra tester que 2 mots de passe par seconde.

Vous pouvez retrouver un exemple d'usage de PBKDF2 en python ici.

Fernet

A l'origine, il s'agit d'une « recette » développée par le Python Cryptographic Authority. Elle inclut toutes les best practices : limitation dans le temps, IV, intégrité/authenticité, « One use, One key », base64 safe. Les algorithmes utilisés sont actuellement considérés comme sûres. Les spécifications sont disponibles ici. Bien que plutôt confidentiel, cela devient un standard, remplaçant avantageusement le JWT/JWS.

Cette recette dispose toutefois de certaines limitations :

- Repose sur l'AES 128 CBC. Il n'est pas possible d'utiliser de l'AES 256 ou le mode d'opération de bloc CTR.
- Ne supporte pas l'AEAD (Authenticated Encryption with Associated Data), uniquement l'AE (Authenticated Encryption). Ainsi, si l'on souhaite vérifié l'intégrité de donnée en clair, il faut les recopier dans la partie géré par Fernet, ce qui fait une redondance de l'information.

Vous pouvez retrouver des exemples directement dans le documentation de la bibliothèque cryptography.