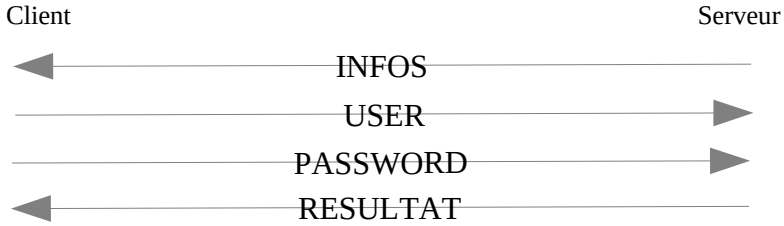
TP N° 03 - Module M2207

Le but de ce TP est de se familiariser avec la programmation d'un serveur d'authentification en Python.

1. Authentification simple :

a) Ecrire un client (`AuthClient.py`) qui suive le protocole AUTH suivant :



Le programme affichera la chaîne INFOS envoyée par le serveur et demandera à l'utilisateur les valeurs de USER et PASS qu'il enverra ensuite au serveur. On affichera enfin le RESULTAT indiqué par le serveur.
S'inspirer du code client utilisé au TP n° 02 (`echoClient.py`), pour écrire le client (`authClient.py`).
On pourra utiliser l'image docker fournie par l'enseignant sur le « registry » de l'IUT pour les tests :

```
docker pull registry.iutbeziers.fr/cb_auth:1.0
docker run --rm -t --name cb registry.iutbeziers.fr/cb_auth:1.0
...# A partir de là, lancer les clients dans une autre console vers l'adresse de l'interface docker0+1 (e.g. 172.17.0.2)
...# when finished
docker stop cb
```

NB : Seul l'utilisateur `toto` et mot de passe `titi` donne le résultat OK pour ce serveur.

b) Afin de ne pas afficher les caractères du mot de passe dans le terminal, on propose d'utiliser le module `getpass`. Tester votre solution dans le fichier `authClient1b.py`.

c) Vérifier que le mot de passe circule toujours « en clair » sur le réseau en capturant les trames avec wireshark.

d) Créer ensuite votre propre serveur (`authServeur.py`) qui ne valide que l'utilisateur `tata` avec le mot de passe `tirlipimpon`. Prendre comme base, le fichier serveur du TP n° 02 (`echoServeur-socketserver.py`).

e) Comment modifier votre serveur (`authServeurFichier.py`) pour qu'il ne valide maintenant que les utilisateurs d'un fichier texte (`users.txt`) respectant le format : 1 utilisateur par ligne, chaque ligne contenant `user:pass`.
On pourra utiliser la fonction `lineExists()` du TD n° 03.

f) Comment utiliser maintenant un format de fichier où le mot de passe n'est plus « en clair » mais remplacé par son haché SHA1 voir TD n° 01 ? Créer le script `authServeurFichier1f.py`.

2. Serveur d'authentification sécurisé (`secureAuthClient.py` et `secureAuthServeur.py`) :

Afin de sécuriser un peu les échanges entre les clients et le serveur, on propose d'utiliser un chiffrement par OU EXCLUSIF à l'aide d'une clé secrète partagée : CIPHER = SKEY ^ MSG et DECIPHER = SKEY ^ CIPHER (Voir annexes ci-dessous).

a) Créer le module `secureData.py` contenant la classe `xorCipher` ainsi que les méthodes **`cipher`**`(msgStr)` et **`decipher`**`(cipherHexStr)` en utilisant judicieusement le code de l'annexe.

b) Adapter les programmes pour utiliser cette classe.

c) Vérifier que le mot de passe ne circule **plus** « en clair » sur le réseau en capturant les trames avec wireshark.

ANNEXE

Voici un exemple de chiffrement par XOR : CIPHER = KEY ^ MSG et DECIPHER = KEY ^ CIPHER

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
def printValues(txtA,a,txtB,b,txtC,c):
  print(f'{txtA}: {hex(a):>18}')
  print(f'{txtB}: {hex(b):>18}')
  print(f'{txtC}: {hex(c):>18}')

k=0xaeb0e392ed25a496
msg='abcd'
#Transforme msg en entier (big-endian)
m=int.from_bytes(bytes(msg,'utf-8'),byteorder='big')
c=k^m
printValues('M',m,'K',k,'C',c)
#Transforme c en hexadecimal
hexStr=hex(c)

c=int(hexStr,16)
d=k^c
printValues('C',c,'K',k,'D',d)
#Transforme d en séquence de 16 bytes (big-endian)
#et supprime les \x00 en partant de la gauche
y=d.to_bytes(16,byteorder='big').lstrip(b'\x00')
msg=str(y,'utf-8')
print('Msg:',msg)
```

```
M:          0x61626364
K: 0xaeb0e392ed25a496
C: 0xaeb0e3928c47c7f2
C: 0xaeb0e3928c47c7f2
K: 0xaeb0e392ed25a496
D:          0x61626364
Msg: abcd
```

NB : Les messages MSG doivent être plus petits que la taille en octets de la clé secrète KEY !