

Compte rendu TP3 MI44 (Projet)

Club privé avec distribution de mot passe par protocole HTTPS

Valentin Mercy

- [Introduction](#)
- [Etape 1 : Vérification du serveur HTTP](#)
- [Etape 2 : Génération du certificat de l'autorité de certification](#)
- [Etape 3 : Génération du certificat du serveur](#)
- [Etape 4 : Connexion HTTPS](#)
- [Etape 5 : Améliorations](#)
- [Sources](#)
- [Conclusion](#)

Introduction

Le projet a pour but de sécuriser la consultation d'un mot de passe secret distribué par un club (par exemple un aéroclub) à ses membres. On imagine que ce mot de passe leur donne accès à une spécificité du club (dans l'exemple précédent, il pourrait s'agir du digicode permettant d'ouvrir le hangar où sont stockés les avions).

Les programmes fournis (en Python), dans l'état de départ, permettent d'effectuer cette tâche de manière non-sécurisée, comme nous allons l'observer dans l'étape 1. A la fin du projet, nous aurons donc modifié ces programmes de façon à établir une connexion **HTTPS** (HyperText Transfer Protocol Secure) = connexion **HTTP** combinée à une couche de sécurité comme **SSL** (Secure Sockets Layer) (utilisé ici) grâce à une autorité de certification qui signera un **certificat** transmis par le serveur.

Etape 1 : Vérification du serveur HTTP

Il s'agit ici de lancer le serveur Flask (pour afficher une page dans le navigateur dont le seul contenu est un texte, supposé secret, qui constitue le mot de passe à transmettre aux membres du club) et d'utiliser l'utilitaire Wireshark pour montrer que n'importe quelle personne mal intentionnée placée sur le même réseau (côté serveur ou côté client) peut écouter le trafic et découvrir notre mot de passe. Le mot de passe choisi ici est **valentinmercyMI44**

- Premièrement, je renseigne mon mot de passe dans le programme **serveur.py**:

```
MESSAGE_SECRET="valentinmercyMI44"
```

- Ensuite, je lance le serveur :

```
python serveur.py
```

- En ouvrant le navigateur à l'adresse **localhost:8081**, on obtient bien le mot de passe :

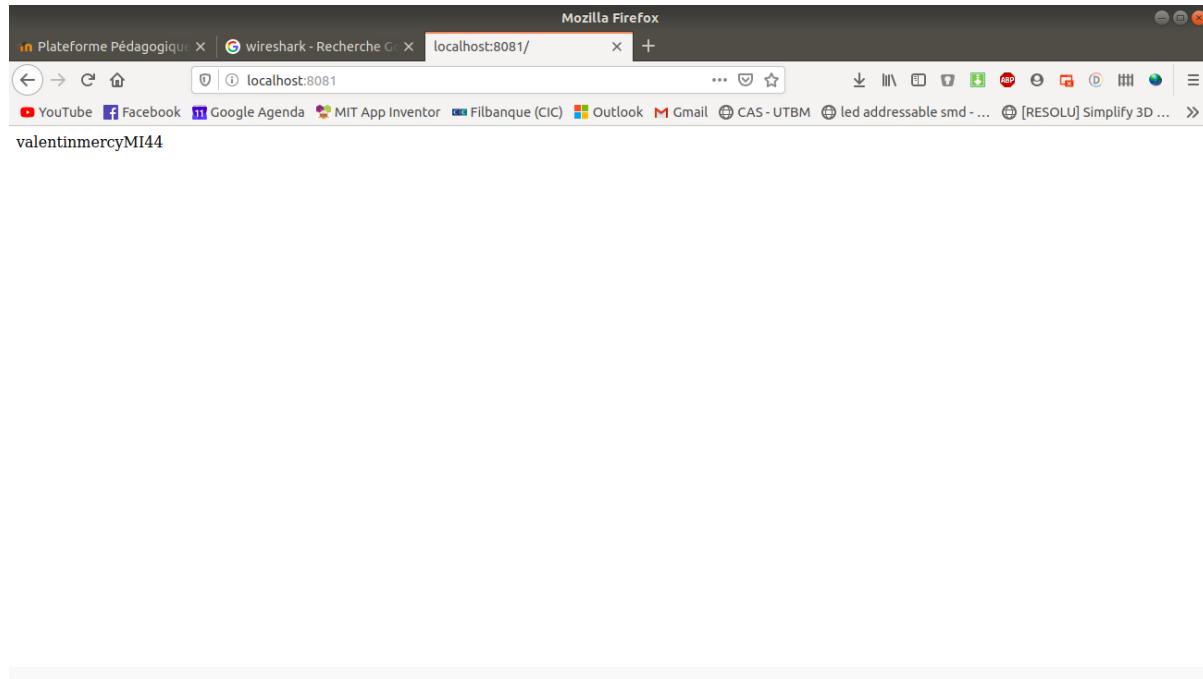


Figure 1: Affichage du mot de passe sur le navigateur en HTTP

* On ouvre alors Wireshark (analyseur de paquets). En lui indiquant le port à filtrer par la commande ci-dessous, on obtient la liste des transmissions suivante : `tcp.port==8081`

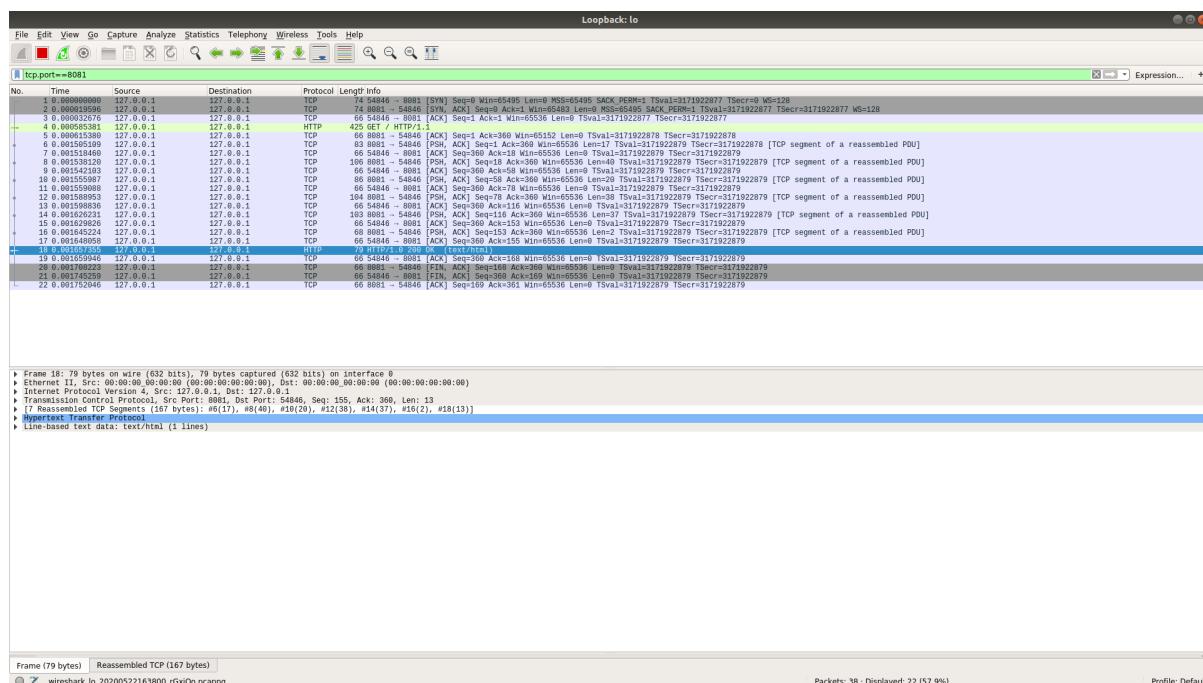


Figure 2: Liste des transmissions disponibles dans Wireshark

* Il nous suffit d'ouvrir la transmission repérée par le protocole HTTP pour afficher notre mot de passe en clair :

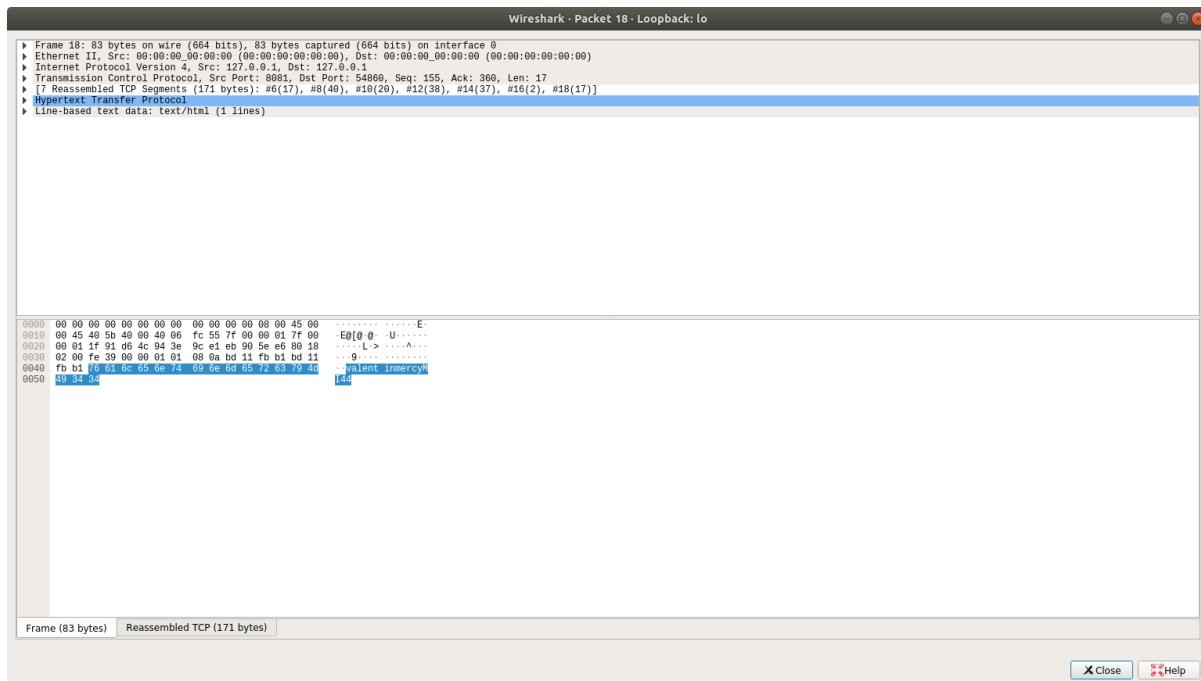


Figure 3: Affichage du mot de passe en clair par Wireshark

Etape 2 : Génération du certificat de l'autorité de certification

Cette étape a pour but de générer le certificat de l'autorité de certification.

- Pour cela, j'utilise le programme **ca_certification.py** en rajoutant premièrement la ligne suivante afin de générer sa clé privée (j'ai choisi le mot de passe **valentinmercy** pour générer le certificat) :

```
private_key = generate_private_key("ca-cle-privee.pem", "valentinmercy")
```

La fonction utilisée **generate_private_key(basename,password)** est importée de **PKI_utile.py**:

```
from PKI_utile import generate_private_key, generate_public_key
```

- Cette fonction, utilisée avec les paramètres donnés plus haut, génère alors le fichier crypté **serveur-cle-privee.pem** contenant la clé privée. Cette clé privée est aussi renournée par la fonction, pour la suite de l'exécution. Le fichier ainsi créé est bien crypté avec RSA comme en témoigne son en-tête:

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,DD18E0AF8078AF58448E3A0F70324E95
```

- Enfin, je rajoute la ligne

```
public_key = generate_public_key(private_key, "ca-cle-
publique.pem", country="FR", state="AL", locality="Mulhouse", org="UTBM", hostna
```

```
me="monca.com")
```

Celle-ci génère un fichier similaire au fichier précédent, à la différence qu'il n'est pas crypté. Cela signifie qu'on peut le lire en utilisant, par exemple, un programme python. Ce programme est donné dans le TP, mais je l'ai renommé en **imprime_pem.py** et modifié de façon à pouvoir passer en argument (sur la ligne de commande) :

```
cert_file_base_name = sys.argv[1]
cert_file_name = os.path.realpath(__file__).split("/")
cert_file_name = "/".join(cert_file_name[:-1]) + "/" + cert_file_base_name
```

Ainsi, avec la ligne de commande suivante :

```
python imprime_pem.py ca-cle-publique.pem
```

On obtient le résultat suivant :

```
Python 2.7.17 (default, Apr 15 2020, 17:20:14) [GCC 7.5.0] 64bit on linux2
/home/valentin/Projet_MI44/ca-cle-publique.pem
Certificate (ca-cle-publique.pem) data:

{'issuer': (((('countryName', u'FR'),),
              (('stateOrProvinceName', u'AL'),),
              (('localityName', u'Mulhouse'),),
              (('organizationName', u'UTBM'),),
              (('commonName', u'monca.com'),)),
   'notAfter': 'Jul 21 15:58:16 2020 GMT',
   'notBefore': u'May 22 15:58:16 2020 GMT',
   'serialNumber': u'0D04CF7D626AD3CD7FE96CBA311CE7A17F121BD2',
   'subject': (((('countryName', u'FR'),),
                 (('stateOrProvinceName', u'AL'),),
                 (('localityName', u'Mulhouse'),),
                 (('organizationName', u'UTBM'),),
                 (('commonName', u'monca.com'),)),
    'version': 3L}
```

Done.

Par contre, lorsqu'on essaie de faire de même avec la clé privée :

```
python imprime_pem.py ca-cle-privee.pem
```

On obtient le résultat suivant :

```
Python 2.7.17 (default, Apr 15 2020, 17:20:14) [GCC 7.5.0] 64bit on linux2
/home/valentin/Projet_MI44/ca-cle-privee.pem
Error decoding certificate: ('Error decoding PEM-encoded file', )
Done.
```

Ce qui est normal, puisque cette clé ne doit pas être accessible autrement qu'avec le mot de passe.

Etape 3 : Génération du certificat du serveur

Cette étape consiste à générer, de façon analogue à l'étape précédente, le certificat du serveur et à le faire signer par l'autorité de certification.

Pour cela, on utilise le programme ***csr_certificat.py*** qu'on complète ainsi :

```
private_key = generate_private_key("serveur-cle-
privee.pem", "valentinmercy")
csr = generate_csr(private_key, "serveur-
csr.pem", country="FR", state="AL", locality="Mulhouse", org="UTBM", hostname="m
oncsr.com")
```

- La première ligne se charge de générer une clé privée pour le serveur (avec le mot de passe **valentinmercy**).
- La seconde se charge d'envoyer la requête CSR (Certificate Sign Request) à l'autorité de certification, grâce à la fonction ***generate_csr(private_key, basename, **kwargs)***. ****kwargs** en Python permet de passer à la fonction un nombre illimité d'arguments. Ici, nous nous en servons pour passer :
 - le pays : `country="FR"`
 - la région : `state="AL"` pour "Alsace"
 - la ville : `locality="Mulhouse"`
 - l'organisation : `org="UTBM"`
 - le nom d'hôte : `hostname="moncsr.com"`

On obtient maintenant 2 nouveaux fichiers **.pem** :

* ***serveur-cle-privee.pem*** : clé privée du serveur, cryptée

* ***serveur_csr.pem*** : clé publique temporaire du serveur (CSR) à faire signer par l'autorité de certification

- On exécute ensuite le programme ***certificatserveur.py*** qui se charge de faire signer la clé publique du serveur par l'autorité de certification, à condition qu'on lui fournisse le mot de passe de cette dernière (celui créé à l'étape 1). Attention : Ce programme est à lancer avec Python >= 3 : utiliser

```
python3 certificatserveur.py
```

On obtient alors le fichier **serveur-cle-publique.pem** qui est la clé publique du serveur signée par l'autorité de certification.

- On utilise à nouveau le programme **imprime_pem.py** pour afficher cette fois le certificat final (**serveur-cle-publique.pem**) :

```
python imprime_pem.py serveur-cle-publique.pem
```

On obtient alors le résultat suivant :

```
Python 2.7.17 (default, Apr 15 2020, 17:20:14) [GCC 7.5.0] 64bit on linux2
/home/valentin/Projet_MI44/serveur-cle-publique.pem
Certificate (serveur-cle-publique.pem) data:

{'issuer': (((('countryName', u'FR'),),
              (('stateOrProvinceName', u'AL'),),
              (('localityName', u'Mulhouse'),),
              (('organizationName', u'UTBM'),),
              (('commonName', u'monca.com'),)),
             'notAfter': 'Jul 21 16:13:25 2020 GMT',
             'notBefore': u'May 22 16:13:25 2020 GMT',
             'serialNumber': u'4217BE4D92C5EBF12984326AD95F3FBAE61A36E6',
             'subject': (((('countryName', u'FR'),),
                           (('stateOrProvinceName', u'AL'),),
                           (('localityName', u'Mulhouse'),),
                           (('organizationName', u'UTBM'),),
                           (('commonName', u'moncsr.com'),)),
                        'subjectAltName': (),
                        'version': 3L}
```

Done.

Etape 4 : Connexion HTTPS

Cette étape consiste à apporter une très légère modification au programme **serveur.py** de façon à établir la connexion HTTPS finale. Cette modification est la suivante :

- Avant modification :

```
app.run(debug=True, host="0.0.0.0", port=8081)
```

- Après modification :

```
app.run(debug=True, host="0.0.0.0", port=8081, ssl_context=("serveur-cle-publique.pem", "serveur-cle-privee.pem"))
```

L'argument **ssl_context** permet de rajouter le tuple (`server_public_key, server_private_key`) qui donne accès aux certificats précédemment générés pour le serveur, nécessaires pour établir une connexion HTTPS. On en profite pour changer le message secret (mot de passe du club) pour être sûr qu'on a bien apporté une modification par rapport à la connexion HTTP précédente :

```
MESSAGE_SECRET="JeSuisLeMotDePasseDuClub"
```

Au lancement du programme **serveur.py** par la ligne de commande

```
python serveur.py
```

le terminal nous demande maintenant le mot de passe du serveur (**valentinmercy**) :

```
valentin@Valentin-Ubuntu:~/Projet_MI44$ python serveur.py
 * Serving Flask app "serveur" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
 deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on https://0.0.0.0:8081/ (Press CTRL+C to quit)
 * Restarting with inotify reloader
 * Debugger is active!
 * Debugger PIN: 343-531-283
Enter PEM pass phrase:
 * Detected change in '/home/valentin/Projet_MI44/serveur.py', reloading
 * Restarting with inotify reloader
 * Debugger is active!
 * Debugger PIN: 343-531-283
Enter PEM pass phrase:
```

Une fois ce mot de passe saisi, on peut ré-ouvrir notre navigateur et constater qu'il refuse à priori de nous donner accès à la page. Le message d'erreur obtenu est le suivant :

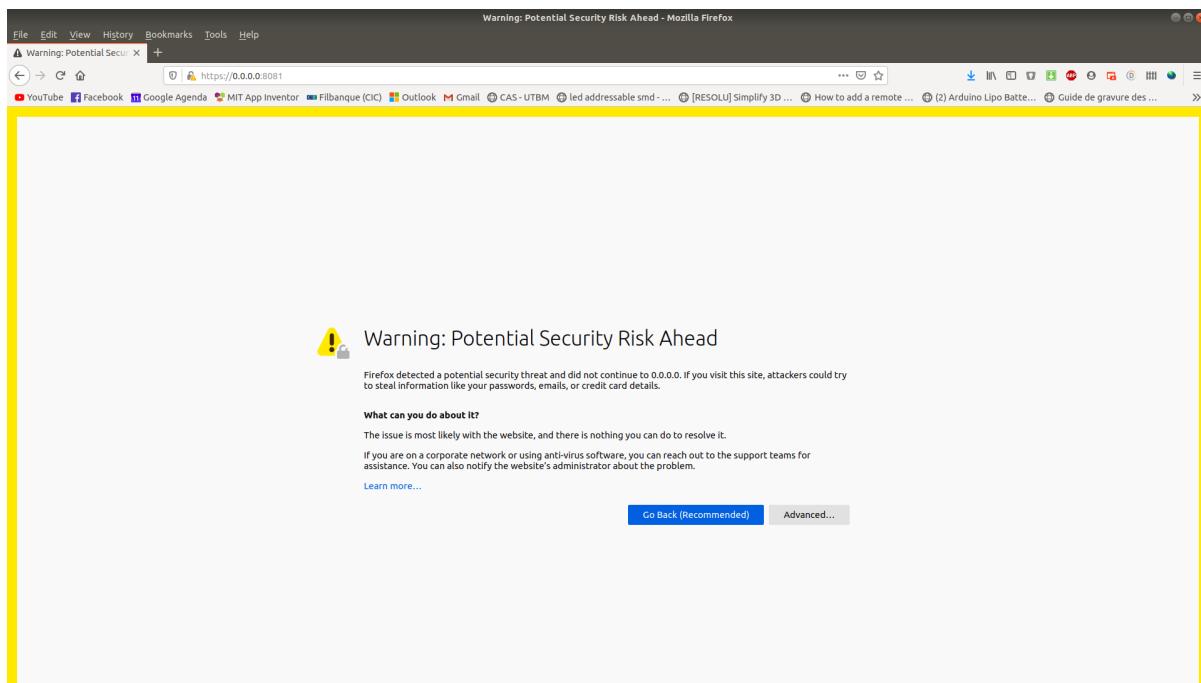


Figure 4 : Message d'erreur obtenu en essayant d'ouvrir la page avec connexion HTTPS

Il suffit alors de cliquer sur **Advanced** puis **Accept the Risk and Continue** pour accepter le certificat et afficher la page avec le mot de passe du club :



Figure 5 : Affichage du mot de passe du club avec la connexion HTTPS

Etape 5 : Améliorations

Pour renforcer la sécurité établie jusque-là, on peut penser à plusieurs pistes :

- Changer l'algorithme de chiffrement ou y ajouter un échange de clés par Diffie-Hellman (éventuellement avec les courbes elliptiques)
- Donner accès au mot de passe après que le membre du club se soit authentifié à l'aide, par exemple, d'un login et d'un mot de passe

C'est cette dernière piste que j'ai décidé d'approfondir. J'ai donc implémenté une connexion par login et mot de passe avec hachage du mot de passe concaténé à un sel, en utilisant les dernières recommandations PBKDF2, c'est-à-dire un HMAC à 80000 itérations. Le mot de passe et le sel seront stockés en base de données MySQL. En cas d'attaque de notre base de données, le HMAC 80000 ralentira considérablement les opérations de hachage pour tenter de découvrir le mot de passe d'un membre.

Pour cette sous-partie du projet, je me suis appuyé sur l'exemple de l'aéroclub donné dans mon introduction. On imagine donc que l'aéroclub souhaite donner à ses membres (désormais nommés pilotes) l'accès à un code renouvelé mensuellement, lequel leur permet d'ouvrir le hangar où sont stockés les avions.

On traitera le problème avec un code à 10 chiffres généré aléatoirement à chaque lancement de **serveur.py**.

Pour pouvoir exécuter mon code, il est nécessaire d'installer **flask_wtf**, **wtforms** ainsi que **flask_sqlalchemy** qui permettent notamment de créer des formulaires avec Flask et communiquer avec des bases de données SQL :

```
pip3 install flask_wtf
pip3 install wtforms
pip3 install flask_sqlalchemy
```

L'objectif de ce projet n'étant pas de s'attarder sur la mise en forme d'une page html, j'ai utilisé des templates html et css trouvées [ici](#).

Pour la base de données, j'ai utilisé une base SQLite, dont la structure et les données sont stockées dans le même fichier **site.db**.

Voici les comptes de pilote actuellement dans la base de données :

Identifiant	Mot de passe
Alice	ADSL
Bob	imbob
Charlie	jesuischarlie
Kevin	kevdu68
Mallory	malleauriz

Tableau I: Comptes existants dans la base de données

Les hash et les sels sont générés automatiquement lorsqu'un nouveau compte est ajouté. Ils sont stockés en base de donnée :

Table: pilot				
	id	username	password_hash	password_salt
1	1	Kevin	36d04deda504811dd80c6d35705644d02914341dcbafe6d16b1ca4f49bd318b1	Filter VoGkuuMuHwiFjeaECKx5zAWIA
2	2	Alice	564d365543078afbf12be456d387b7d2408fbdc58b216cfb04b6ded39928820a	FnWIHjf3SUuNIAsZbab53wgL
3	3	Bob	5516147378d4f91b2203266b9bdf444d06516589a986a18d3947f02bf0698a79	UPFLVvxOdHBPrviaVvfDYYnR7
4	4	Mallory	dcc5e2201c32a8214b24afeecd6203dbe9a8e11a192328bdb6fb4f17d8efc2706	ZXioKb9kqXfQt6fBOlzTV3E14
5	5	Charlie	776acf22cf7a0d3156ece4366cff16604a516688de8720c420751d615cc0105	xTqYifcPWkrQStj62isNuVJLB

Figure 6 : Table des pilotes dans la BdD

L'interface que j'ai créée permet également de créer des comptes de pilote. Cela vous permettra de mieux tester ma solution en vous créant vos propres comptes. Toutefois, on imagine bien sûr qu'en réalité cette fonction serait réservée à un administrateur de l'aéroclub, ou à minima soumise à son approbation. Sinon, tout le monde pourrait se créer un compte et accéder au code du hangar... Pour simplifier le projet, je n'ai pas traité cette approbation. Il s'agirait simplement de créer un compte spécial qui donne le privilège de créer d'autres comptes.

Pour tester le projet, il vous suffit donc de démarrer le serveur en Python 3 (impératif !):

```
python3 serveur.py
```

Vous devez maintenant saisir le mot de passe du serveur : **valentinmercy**

Ensuite, connectez-vous avec l'un des comptes existants. Vous devriez alors voir apparaître le code du hangar. Vous pouvez maintenant vérifier que le code du hangar est le même avec différents comptes en vous déconnectant/reconnectant plusieurs fois.

Vous pouvez également vérifier que le code du hangar affiché est bien le même que celui sorti sur le terminal par le serveur.

Bien entendu, si vous essayez d'accéder à l'url https://0.0.0.0:8081/get_code sans vous être connecté au préalable, vous obtiendrez le message d'erreur suivant :

Vous n'êtes pas connecté !

[Retour](#)

Figure 8 : Erreur lors d'un accès non-connecté

Voici les résultats de mes tests :

```
valentin@Valentin-Ubuntu:~/Projet_MI44$ python3 serveur.py
/home/valentin/.local/lib/python3.6/site-
packages/flask_sqlalchemy/__init__.py:834: FSADeprecationWarning:
SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be
disabled by default in the future. Set it to True or False to suppress
this warning.
  'SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and '
Mot de passe du hangar généré aléatoirement : 1535160774
  * Serving Flask app "serveur" (lazy loading)
  * Environment: production
WARNING: This is a development server. Do not use it in a production
deployment.
  Use a production WSGI server instead.
```

```
* Debug mode: on
* Running on https://0.0.0.0:8081/ (Press CTRL+C to quit)
* Restarting with stat
/home/valentin/.local/lib/python3.6/site-
packages/flask_sqlalchemy/__init__.py:834: FSADeprecationWarning:
SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be
disabled by default in the future. Set it to True or False to suppress
this warning.
  'SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and '
Mot de passe du hangar genere aleatoirement : 9211497873
```

Menu Principal - Aéroclub

[Créer un compte](#)

[Me connecter](#)

Figure 8: Menu principal

Connexion en tant que pilote

Identifiant

Mot de passe

[Me connecter](#)

Vous voulez créer un compte pilote ? [Créer un compte](#)

Figure 9: Page de connexion

Connexion en tant que pilote

Identifiant

Charlie

Mot de passe

Me connecter

Vous voulez créer un compte pilote ? [Créer un compte](#)

Figure 10 : Page de connexion remplie avec les identifiants de Charlie

Connexion réussie !

Bonjour Charlie ! Voici le code du hangar aux avions : **9211497873**

[Me déconnecter](#)

Figure 11 : Affichage du code du hangar

Sources

[Wikipedia : HTTPS](#)

[Wikipedia : SSL](#)

[YouTube : Corey Schafer - Tutoriel Python Flask Codes sources pour formulaires Flask](#)

Conclusion

Finalement, ce TP m'aura permis de comprendre le fonctionnement et l'implémentation d'une connexion sécurisée HTTPS, qui est devenue ces dernières années la norme pour établir toute connexion internet nécessitant un minimum de sécurité (sites de banques par exemple).

De plus, il m'aura permis d'apprendre à utiliser Flask pour Python, dont j'avais déjà entendu parler mais dont je n'avais pas eu l'utilité jusque-là. Cet outil peut être utile pour implémenter rapidement des applications légères telles que celle développée pour ce TP.