

Objectif :

Nous supposons un club privé qui distribue un mot de passe d'entrée aux adhérents une fois par mois. Ils utilisent ce mot de passe pour accéder au club. Les adhérents obtiennent le mot de passe en se connectant sur un lien URL secret. Le lien leur est communiqué lors de leur dernière rencontre physique. Actuellement, il s'agit simplement d'une connexion http, ce qui permet à toute personne observant le trafic de lire le mot de passe du club. L'objectif du projet est de remplacer la connexion http par une connexion https.

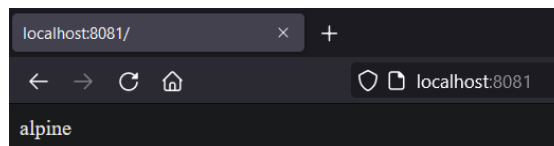
1. Vérification du serveur http

Examinons le fichier `run_server.py` et changeons son mot de passe :

```
# définir le message secret
SECRET_MESSAGE = "alpine" # A modifier
app = Flask(__name__)
```

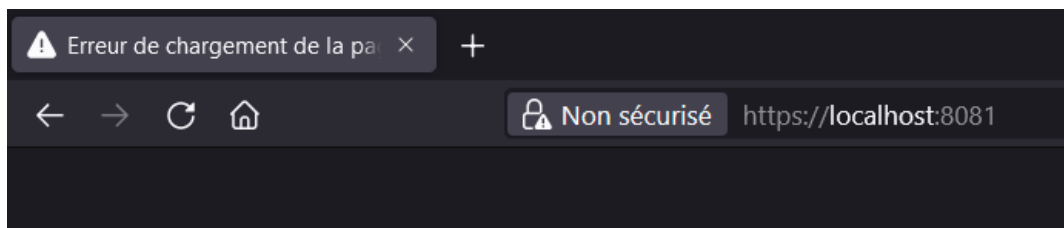
Message secret envoyé en plaintext sur la page web

On peut ici changer le message secret. La page web affiche donc :



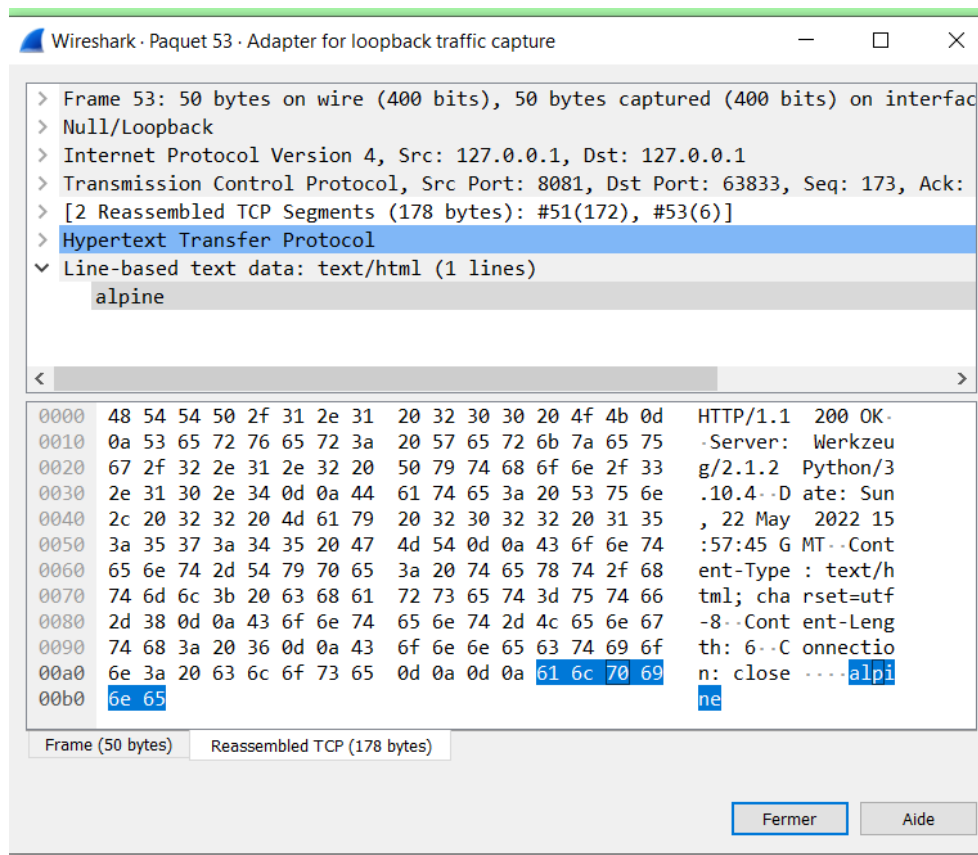
Affichage du secret sur la page web

De plus, si j'essaie d'utiliser le https, mon navigateur me retourne une erreur :



Erreur retournée suite à l'utilisation pour protocole https

Désormais, on va écouter l'hôte local grâce à Wireshark.



Analyse du paquet 53 de type text/html

On peut lire clairement le mot de passe en clair (plaintext). Si quelqu'un effectue une attaque « man in the middle », il peut lire tranquillement le mot de passe. Nous devons résoudre ce problème très embarrassant.

2. Génération du certificat de l'autorité de certification et du serveur

L'autorité de certification (ca) crée un certificat autosigné permettant de sécuriser la connexion et de rendre celle-ci légitime. Pour se faire, modifions d'abord le fichier localisé dans ca/core.py.

```
def __init__(self, config: Configuration, password: str, private_key_filename: str, public_key_filename: str):
    self._config = config
    self._private_key = generate_private_key(private_key_filename, password)
    self._public_key = generate_public_key(self._private_key, public_key_filename, config) #Configuration : c
```

Définition de la classe CertificateAuthority

Ici, on a complété les fonctions generate_private_key() et generate_public_key() avec leurs arguments correspondant selon leur définition dans tools/core.py. Lors de l'instanciation de la classe, les arguments de celles-ci seront utilisés pour nos deux fonctions.

Pour generate_private_key() qui génère la clé privée, on lui passe le nom de fichier (avec le chemin) de la clé privée et son mot de passe qui lui sera associé.

Pour generate_public_key() qui génère la clé publique, on lui passe la clé privée par référencement (celle créée juste au-dessus) puis le nom de fichier (avec chemin) de la clé publique, et enfin la config du certificat (UTBM, FR...)

Maintenant que la classe est complétée, on va l'instancier correctement dans build.py.

```
# Création de l'autorité de certification
certificate_authority = CertificateAuthority(CA_CONFIGURATION, CA_PASSWORD, CA_PRIVATE_KEY_FILENAME, CA_PUBLIC_KEY_FILENAME)
```

Instanciation de la classe CertificateAuthority

Ici, instancie CertificateAuthority et on lui donne comme argument la configuration du certificat ("FR", "Territoire de Belfort", "Sevenans", "UTBM_CA", "localhost"), le mot de passe de la clé privée, le nom de fichier de la clé privée, et enfin le nom de fichier de la clé publique. Voilà, notre certificat provenant de l'autorité de certification est créé.

Désormais, nous allons voir comment cela fonctionne pour le certificat du serveur.

Modifions d'abord le fichier localisé dans server/core.py

```
def __init__(self, config: Configuration, password: str, private_key_filename: str, csr_filename: str):
    self._config = config
    self._private_key = generate_private_key(private_key_filename, password)
    self._csr = generate_csr(self._private_key, csr_filename, config)
```

Définition de la classe Server

Ici, on va appeler les fonctions generate_private_key() et generate_csr() contenue dans le fichier /tools/core.py. Lors de l'instanciation de la classe, les arguments de celles-ci seront utilisés pour nos deux fonctions.

Pour generate_private_key(), on lui passe en argument le nom de la clé privée et son mot de passe associé. Pour generate_csr(), on lui passe la clé privée générée juste au-dessus, le nom du fichier csr (certificat non signé csr = *certificat signing request*) et la configuration du certificat (utbm, fr...).

Maintenant que la classe est complétée, instancions-la depuis build.py

```
# Création du server
server = Server(SERVER_CONFIGURATION, SERVER_PASSWORD, SERVER_PRIVATE_KEY_FILENAME, SERVER_CSR_FILENAME)
```

Instanciation de la classe Server

On donne en argument pour instancier la classe la configuration du certificat du serveur ("FR", "Territoire de Belfort", "Sevenans", "UTBM_SER", "localhost"), le mot de passe de la clé privée du serveur, le nom du fichier de la clé privée du serveur (avec le chemin) et le nom du certificate signing request (certificat non signé)

Préparons désormais ca/core.py pour signer notre certificat temporaire afin qu'il devienne notre certificat définitif (clé publique) pour le serveur.

```
def sign(self, csr, certificate_filename: str):
    sign_csr(csr, self._public_key, self._private_key, certificate_filename)
```

Définition de la fonction sign

Ici, on fait appel à la méthode sign_csr définie dans tools/core.py. On a besoin du certificat non signé en premier argument, de la clé publique et privée générée plus haut dans la classe CertificateAuthority, et du nom du certificat, ce sera la clé publique du serveur.

On appelle ensuite cette fonction dans le build.py :

```
# Signature du certificat par l'autorité de certification
signed_certificate = certificate_authority.sign(server._csr, SERVER_PUBLIC_KEY_FILENAME)
```

Appel de la méthode sign de la classe instanciée : certificate_authority

On lui donne donc comme argument la requête de certificat du server instancié plus haut, puis le nom du fichier de la clé publique du serveur qui sera créé.

Désormais, on va afficher certaines infos de nos certificats (clés publics : celui de l'autorité de certification et celui de notre serveur)

Pour se faire, appelons depuis build.py la méthode print_perms contenu dans le fichier print_pems.py


```
#impression des certificats à compléter regardez #print_pems
for files in [
    CA_PUBLIC_KEY_FILENAME,
    SERVER_PUBLIC_KEY_FILENAME]:
    print("\nVoici le contenu du fichier "+files)
    ppems.print_perms(files)
```

On affiche ici les fichiers des clés publiques

Ici, j'ai fais une boucle for pour afficher les deux en même temps.

On doit juste passer le nom de fichier (avec chemin) pour la méthode print_perms.

Voici le contenu des fichiers retourné.



```
Windows PowerShell
PS C:\Users\Brice\Documents\UTBM\P22\RS40\TP2> py '.\build.py'

Voici le contenu du fichier resources/ca-public-key.pem
[<Certificate(PEM string with SHA-1 digest 'e60d4ad2d2ced3b1733cc6f48ba86e0d9ff89136')>]

Voici le contenu du fichier resources/server-public-key.pem
[<Certificate(PEM string with SHA-1 digest '05da2c236a0c8fa1f9193e2860666219480a27fc')>]

finished ...
PS C:\Users\Brice\Documents\UTBM\P22\RS40\TP2>
```

Résultats de l'affichage des fichiers pems

Le contenu retourné par parsefile (de la bibliothèque pem) nous affiche le SHA-1 des fichier clé publique du ca et du serveur.

Pour ouvrir les clé privée, on a besoin des mots de passe suivants :

- server-private-key.pem : **caPassword**
- server-public-key.pem : **serverPassword**

3. Connexion https

Ici, nous devons spécifier à flask le contexte SSL (ssl_context). En effet, nous devons lui indiquer la clé publique et privée du serveur.

```
RESOURCES_DIR = "resources/"
SERVER_PUBLIC_KEY_FILENAME = RESOURCES_DIR + "server-public-key.pem"
SERVER_PRIVATE_KEY_FILENAME = RESOURCES_DIR + "server-private-key.pem"

@app.route("/")
def get_secret_message():
    return SECRET_MESSAGE

if __name__ == "__main__":
    # HTTPS version
    context = (SERVER_PUBLIC_KEY_FILENAME, SERVER_PRIVATE_KEY_FILENAME)
    app.run(debug=True, host="0.0.0.0", port=8081, ssl_context=context)
```

Implémentation de l'HTTPS : On précise les certificats à Flask

On crée donc la variable context (tableau) contenant la clé public et privée du serveur, on ajoute ensuite context aux argument de la méthode run de app (flask).

Désormais, lorsqu'on exécute run_server.py, il faut préciser le mot de passe de la clé privée du serveur (ici : **serverPassword**)

```
PS C:\Users\Brice\Documents\UTBM\P22\RS40\TP2> py '.\run_server.py'
* Serving Flask app 'run_server' (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
Enter PEM pass phrase:
```

Demande de mot de passe lors du lancement du serveur web

Après nous avoir demandé deux fois le mot de passe, le serveur web se lance.

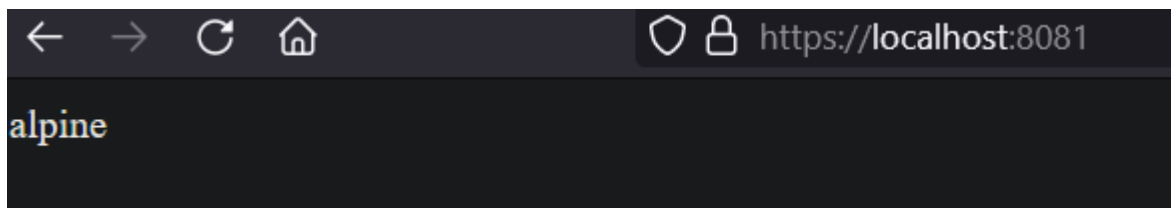
Logiquement, en entrant <https://localhost:8081/> dans mon navigateur (Firefox), cela devrait fonctionner. Cependant, Firefox ne reconnaît pas le délivreur (ca) du certificat, il va falloir l'ajouter manuellement.



Message d'erreur retourné par le navigateur

En ajoutant le certificat ca-public-key.pem à Firefox dans les préférences (ou le certificat server-public-key.pem car il est lui-même signé par le ca), l'erreur disparaît et on obtient bien une connexion sécurisée. (penser à recharger la page).

J'aurai très bien pu ignorer l'exception en cliquant sur « Accepter le risque et poursuivre » mais je voulais bien faire les choses.



Connexion https réussie

Le cadenas est bien visible dans la barre de recherche, il n'est pas barré. La connexion https semble réussie. Retournons désormais dans Wireshark pour voir si on utilise TLS pour chiffrer nos paquets.

tcp.port==8081						
No.	Time	Source	Destination	Protocol	Length	Info
382	22.061570	127.0.0.1	127.0.0.1	TCP	52	64055 → 8081 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SAC
383	22.061599	127.0.0.1	127.0.0.1	TCP	52	8081 → 64055 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MS
384	22.061611	127.0.0.1	127.0.0.1	TCP	44	64055 → 8081 [ACK] Seq=1 Ack=1 Win=65495 Len=0
387	22.063485	127.0.0.1	127.0.0.1	TLSv1.3	683	Client Hello
388	22.063507	127.0.0.1	127.0.0.1	TCP	44	8081 → 64055 [ACK] Seq=1 Ack=640 Win=64856 Len=0
393	22.063729	127.0.0.1	127.0.0.1	TLSv1.3	285	Server Hello, Change Cipher Spec, Application Data, Ap
394	22.063739	127.0.0.1	127.0.0.1	TCP	44	64055 → 8081 [ACK] Seq=640 Ack=242 Win=65254 Len=0
395	22.064154	127.0.0.1	127.0.0.1	TLSv1.3	124	Change Cipher Spec, Application Data
396	22.064167	127.0.0.1	127.0.0.1	TCP	44	8081 → 64055 [ACK] Seq=242 Ack=720 Win=64776 Len=0
398	22.064252	127.0.0.1	127.0.0.1	TLSv1.3	299	Application Data
400	22.064263	127.0.0.1	127.0.0.1	TCP	44	64055 → 8081 [ACK] Seq=720 Ack=497 Win=64999 Len=0
403	22.064363	127.0.0.1	127.0.0.1	TLSv1.3	533	Application Data
404	22.064372	127.0.0.1	127.0.0.1	TCP	44	8081 → 64055 [ACK] Seq=497 Ack=1209 Win=64287 Len=0
409	22.065551	127.0.0.1	127.0.0.1	TLSv1.3	238	Application Data
410	22.065564	127.0.0.1	127.0.0.1	TCP	44	64055 → 8081 [ACK] Seq=1209 Ack=691 Win=64805 Len=0
411	22.065595	127.0.0.1	127.0.0.1	TLSv1.3	72	Application Data
412	22.065602	127.0.0.1	127.0.0.1	TCP	44	64055 → 8081 [ACK] Seq=1209 Ack=719 Win=64777 Len=0
416	22.065676	127.0.0.1	127.0.0.1	TCP	44	8081 → 64055 [FIN, ACK] Seq=719 Ack=1209 Win=64287 Len
418	22.065685	127.0.0.1	127.0.0.1	TCP	44	64055 → 8081 [ACK] Seq=1209 Ack=720 Win=64777 Len=0
419	22.065757	127.0.0.1	127.0.0.1	TLSv1.3	68	Application Data
420	22.065767	127.0.0.1	127.0.0.1	TCP	44	8081 → 64055 [RST, ACK] Seq=720 Ack=1233 Win=0 Len=0

Paquets capturés par Wireshark lors de l'actualisation de la page https

Ça y est, on ne voit plus de html / text. Notre connexion est bien sécurisée. On peut même voir que le protocole utilisé pour chiffrer les données est TLSv1.3, ce qui signifie qu'il utilise l'algorithme HKDF pour le chiffrement.

4. Améliorations

J'ai décidé de créer une petite page html d'authentification. Pas de base de données ici, juste un nom d'utilisateur et mot de passe définis dans une boucle if de `get_secret_message()`.

En utilisant `render_template`, nous devons stocker notre page (`login.html`) dans un dossier nommé « templates » à la racine de notre TP selon la documentation de flask.

J'ai d'abord utilisé cette méthode permettant cette authentification, mais ici sans réel chiffrement du mot de passe :

```
@app.route("/", methods=["GET", "POST"])
def get_secret_message():
    error = None
    if request.method == "POST":
        if request.form["username"] != "RS40" or request.form["password"] != "test1234":
            error = "Nom d'utilisateur ou mot de passe incorrect. Reessayez"
        else:
            return SECRET_MESSAGE
    return render_template("login.html", error=error)
```

Méthode `get_secret_message()` avec l'authentification

On peut voir ici que le nom d'utilisateur est « **RS40** » et que le mot de passe est « **test1234** ». J'aurai pu créer une base de données avec plusieurs noms d'utilisateurs.

J'ai décidé d'améliorer cette méthode avec l'utilisation de la fonction chiffrement `pbkdf2` de tel sorte que mon mot de passe soit stocké sous forme de hash, il n'est donc plus en clair.

`pbkdf2` ajoute le sel à notre mot de passe, puis utilise une fonction à sens unique (ici `sha256`), et répète ces opérations un nombre `n` de fois (ici 10).

```
username = input("Veuillez entrer le nom d'utilisateur\n")
sel = os.urandom(8) # Création du sel (8 caractères)
hashpassword = hashlib.pbkdf2_hmac('sha256', bytes(input("Veuillez entrer le mot de passe pour l'utilisateur "+username+"\n"))
                                , sel, 10)

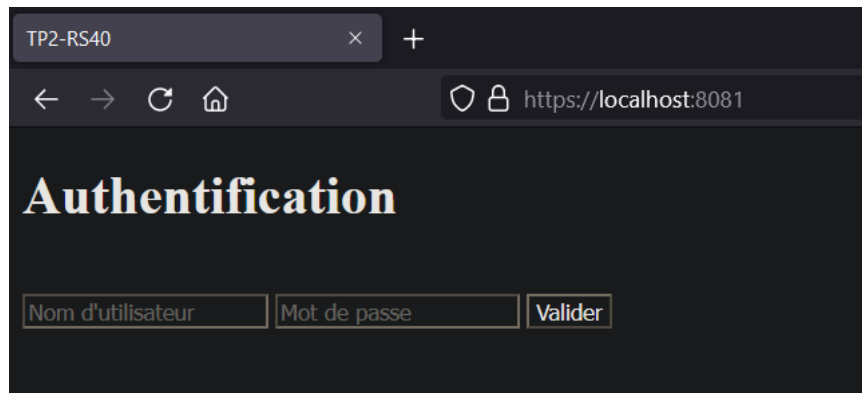
@app.route("/", methods=["GET", "POST"])
def get_secret_message():
    error = None
    if request.method == "POST":
        hashEnteredPassword = hashlib.pbkdf2_hmac('sha256', bytes(request.form["password"], encoding="utf-8"), sel, 10) # Hash
        #request.form["password"] récupérer le mot de passe entré dans le formulaire
        if request.form["username"] != username or hashEnteredPassword != hashpassword:
            error = "Nom d'utilisateur ou mot de passe incorrect. Reessayez"
        else:
            return SECRET_MESSAGE
    return render_template("login.html", error=error) # on retourne la page web avec l'erreur si mot de passe incorrect
```

Amélioration de l'authentification

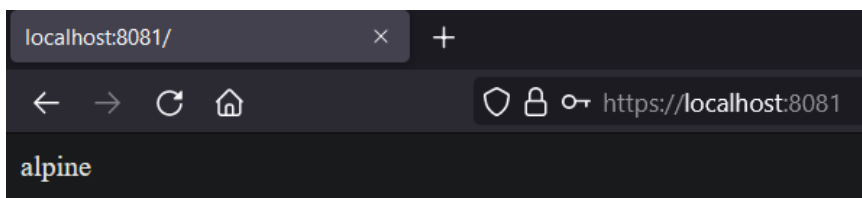
Pour générer le sel, j'appelle `urandom` de la librairie `os`, celui-ci est donc généré de manière aléatoire. J'ai ici choisi 8 caractères.

Pour `pbkdf2`, j'ai fait appelle à la fonction `pbkdf2_hmac` de la bibliothèque `hashlib`. Elle prend en argument le type de fonction de hash, le mot de passe en octet (utilisation de `bytes` avec `utf-8`), le sel, et le nombre d'itération.

Enfin, je laisse l'utilisateur entrer le nom d'utilisateur et le mot de passe dans la console python pour le test. Ce mot de passe sera stocké sous forme de hash avec pbkfd2. Lors de l'authentification sur la page web, le mot de passe sera directement stocké de manière hashé, puis comparé avec le hash calculé précédemment. Je ne stocke jamais en variable locale le mot de passe non hashé.



Page d'authentification



Affichage du secret après authentification

Encore une fois, si j'utilise Wireshark, je ne verrai pas le nom d'utilisateur et le mot de passe que j'ai entré pour accéder au secret.

Remarque : J'aurai pu utiliser des mots de passe forts (majuscule, minuscule, chiffres, caractères spéciaux) pour les mots de passe des clés privées et généré une première fois de manière aléatoire afin qu'ils soient très dur à attaquer en bruteforce. Cependant, par aspect pratique pour le correcteur, j'ai utilisé des mots de passes faciles pour le projet.

Une autre amélioration aurait été possible avec la création de l'utilisateur et du mot de passe directement dans la page web, impliquant une base de données, et une confirmation du mot de passe. (à entrer deux fois)