Table des matières

Vous êtes-vous déjà demandé pourquoi vous pouviez envoyer les informations de votre carte de crédit sur Internet? Vous avez peut-être remarqué le `https://`sur les URL de votre navigateur, mais de quoi s'agit-il et comment protège-t-il **vos informations** ? Ou peut-être souhaitez-vous créer une application Python HTTPS, mais vous ne savez pas exactement ce que cela signifie. Comment pouvez-vous être sûr que votre application Web est sûre?

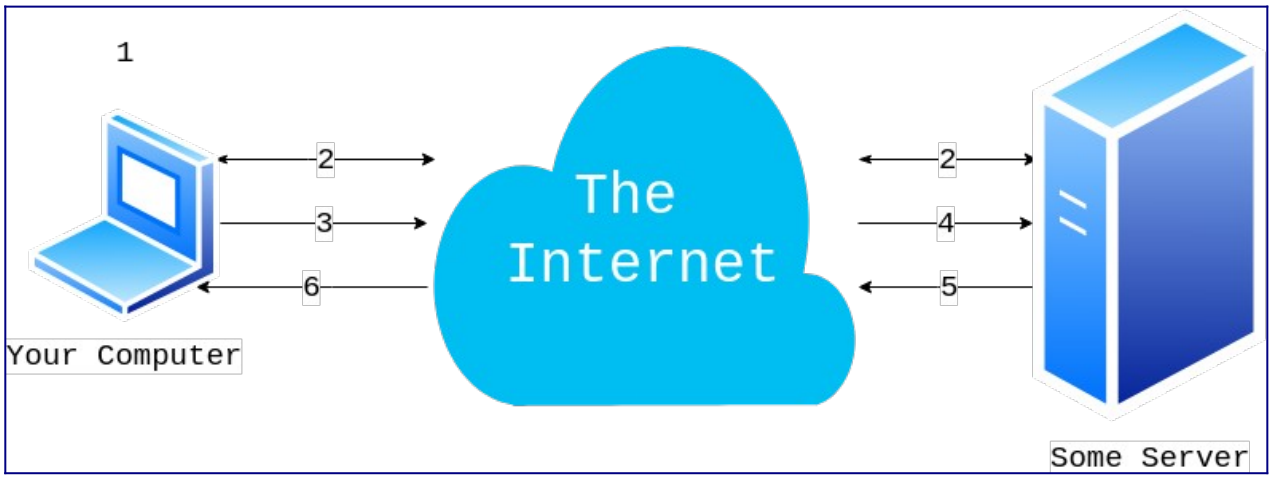
Cela peut vous surprendre de savoir que vous n'avez pas besoin d'être un expert en sécurité pour répondre à ces questions! Dans ce didacticiel, vous aurez une connaissance pratique des différents facteurs qui se combinent pour assurer la sécurité des communications sur Internet. Vous verrez des exemples concrets de la façon dont une application Python HTTPS sécurise les informations.

Dans ce didacticiel, vous apprendrez à:

- Surveiller et analyser le **trafic réseau**
- Appliquer la **cryptographie** pour protéger les données
- Décrire les concepts de base de **l'infrastructure à clé publique (PKI)**
- Créez votre propre **autorité de certification**
- Créer une **application Python HTTPS**
- Identifier les Python HTTPS courants **avertissements et erreurs**

Qu'est-ce que HTTP?

Avant de vous plonger dans HTTPS et son utilisation en Python, il est important de comprendre son parent, HTTP . Cet acronyme signifie **HyperText Transfer Protocol** , qui sous-tend la plupart des communications qui se produisent lorsque vous surfez sur vos sites Web préférés. Plus spécifiquement, HTTP est la façon dont un agent utilisateur, comme votre navigateur Web, communique avec un serveur Web, comme *realpython.com* . Voici un schéma simplifié des communications HTTP:



Ce diagramme montre une version simplifiée de la façon dont votre ordinateur communique avec un serveur. Voici la répartition de chaque étape:

1. Vous dites à votre navigateur d'accéder à `http://someserver.com/link`.
2. Votre appareil et le serveur établissent une TCP connexion .
3. Votre navigateur envoie une **requête HTTP** au serveur.
4. Le serveur reçoit la requête HTTP et l'analyse.
5. Le serveur répond avec une **réponse HTTP** .
6. Votre ordinateur reçoit, analyse et affiche la réponse.

Cette ventilation capture les bases de HTTP. Vous faites une demande à un serveur et le serveur renvoie une réponse. Bien que HTTP ne nécessite pas TCP, il nécessite un protocole fiable de niveau inférieur. En pratique, il s'agit presque toujours de TCP sur IP (bien que Google essaie de créer un remplaçant). Si vous avez besoin d'un rappel, consultez la programmation de socket en Python (Guide) .

En ce qui concerne les protocoles, HTTP est l'un des plus simples. Il a été conçu pour envoyer du contenu sur Internet, comme du HTML, des vidéos, des images, etc. Cela se fait avec une requête et une réponse HTTP. Les requêtes HTTP contiennent les éléments suivants:

- **La méthode** décrit l'action que le client souhaite effectuer. La méthode pour le contenu statique est généralement `GET`, bien qu'il y en ait d'autres disponibles, comme `POST`, `HEAD`, et `DELETE`.
- **Le chemin** indique au serveur la page Web que vous souhaitez demander. Par exemple, le chemin de cette page est `/python-https`.
- **La version** est l'une des nombreuses versions HTTP, comme 1.0, 1.1 ou 2.0. Le plus courant est probablement 1,1.

- **Les en-têtes** aident à décrire des informations supplémentaires sur le serveur.
- **Le corps** fournit au serveur les informations du client. Bien que ce champ ne soit pas obligatoire, il est typique pour certaines méthodes d'avoir un corps, comme un `POST`.

Ce sont les outils que votre navigateur utilise pour communiquer avec un serveur. Le serveur répond avec une réponse HTTP. La réponse HTTP contient les éléments suivants:

- **La version** identifie la version HTTP, qui sera généralement la même que la version de la requête.
- **Le code d'état** indique si une demande a été exécutée avec succès. Il existe de nombreux codes d'état .
- **Le message d'état** fournit un message lisible par l'homme qui aide à décrire le code d'état.
- **Les en-têtes** permettent au serveur de répondre avec des métadonnées supplémentaires sur la demande. Ce sont le même concept que les en-têtes de demande.
- **Le corps** porte le contenu. Techniquement, cela est facultatif, mais il contient généralement une ressource utile.

Ce sont les blocs de construction pour HTTP. Si vous souhaitez en savoir plus sur HTTP, vous pouvez consulter une page de présentation pour en savoir plus sur le protocole.

Qu'est-ce que HTTPS?

Maintenant que vous en savez un peu plus sur HTTP, qu'est-ce que HTTPS? La bonne nouvelle est que vous le savez déjà! HTTPS signifie **HyperText Transfer Protocol Secure** . Fondamentalement, HTTPS est le même protocole que HTTP mais avec l'implication supplémentaire que les communications sont sécurisées.

HTTPS ne réécrit aucun des principes de base HTTP sur lesquels il est construit. Au lieu de cela, HTTPS consiste en un HTTP régulier envoyé via une connexion cryptée. En règle générale, cette connexion chiffrée est fournie par TLS ou SSL, qui sont soient **des protocoles cryptographiques** qui chiffrent les informations avant qu'elles ne envoyées sur un réseau.

Remarque: TLS et SSL sont des protocoles extrêmement similaires, bien que SSL soit en voie de disparition, avec TLS pour le remplacer. Les différences entre ces protocoles sortent du cadre de ce didacticiel. Il suffit de savoir que TLS est la version la plus récente et la meilleure de SSL.

Alors, pourquoi créer cette séparation? Pourquoi ne pas simplement introduire la complexité dans le protocole HTTP lui-même? La réponse est la **portabilité** . La sécurisation des communications est un problème important et difficile, mais HTTP n'est que l'un des nombreux protocoles nécessitant une sécurité. Il en existe d'innombrables autres dans une grande variété d'applications:

- E-mail
- Instant Messaging
- VoIP (voix sur IP)

Il y a d'autres aussi! Si chacun de ces protocoles devait créer son propre mécanisme de sécurité, alors le monde serait beaucoup moins sûr et beaucoup plus déroutant. TLS, qui est souvent utilisé par les protocoles ci-dessus, fournit une méthode courante pour sécuriser les communications.

Remarque: cette séparation des protocoles est un thème courant dans les réseaux, à tel point qu'elle a un nom. Le modèle OSI représente les communications depuis le support physique jusqu'au HTML rendu sur cette page!

Presque toutes les informations que vous apprendrez dans ce didacticiel seront applicables à plus que de simples applications Python HTTPS. Vous apprendrez les bases des communications sécurisées ainsi que leur application spécifique au HTTPS.

Pourquoi HTTPS est-il important?

Les communications sécurisées sont essentielles pour fournir un environnement en ligne sûr. À mesure que le monde évolue en ligne, y compris les banques et les sites de soins de santé, il devient de plus en plus important pour les développeurs de créer des applications Python HTTPS. Encore une fois, HTTPS est juste HTTP sur TLS ou SSL. TLS est conçu pour assurer la confidentialité des écoutes indiscrètes. Il peut également fournir une authentification du client et du serveur.

Dans cette section, vous explorerez ces concepts en profondeur en procédant comme suit:

1. **Création d'** un serveur Python HTTPS
2. **Communiquer** avec votre serveur Python HTTPS
3. **Capturer** ces communications
4. **Analyser** ces messages

Commençons!

Création d'un exemple d'application

Supposons que vous soyez le chef d'un club Python sympa appelé les Secret Squirrels. Les écureuils, étant secrets, ont besoin d'un message secret pour assister à leurs réunions. En tant que responsable, vous choisissez le message secret, qui change à chaque réunion. Parfois, cependant, il vous est difficile de rencontrer tous les membres avant la réunion pour leur dire le message secret! Vous décidez de mettre en place un serveur secret où les membres peuvent simplement voir le message secret par eux-mêmes.

Remarque: l'exemple de code utilisé dans ce didacticiel n'est **pas** conçu pour la production. Il est conçu pour vous aider à apprendre les bases de HTTP et TLS. **Veuillez ne pas utiliser ce code pour la production.** De nombreux exemples ci-dessous ont des pratiques de sécurité terribles. Dans ce didacticiel, vous en apprendrez davantage sur TLS et sur une façon dont il peut vous aider à être plus sûr.

Vous avez suivi quelques tutoriels sur Real Python et décidez d'utiliser certaines dépendances que vous connaissez:

- **Flask** pour créer une application Web
- **uWSGI en** tant que serveur de production
- **demandes** d'exercice de votre serveur

Pour installer toutes ces dépendances, vous pouvez utiliser `pip`:

```
$ pip install flask uwsgi requests
```

Une fois vos dépendances installées, vous commencez à écrire votre application. Dans un fichier appelé `server.py`, vous créez une Flask application :

```
# server.py
from flask import Flask

SECRET_MESSAGE = "fluffy tail"
app = Flask(__name__)

@app.route("/")
```

```
def get_secret_message():
    return SECRET_MESSAGE
```

Cette application Flask affichera le message secret chaque fois que quelqu'un visite le `/`chemin de votre serveur. Avec cela à l'écart, vous déployez votre application sur votre serveur secret et l'exécutez:

```
$ uwsgi --http-socket 127.0.0.1:5683 --mount /=server:app
```

Cette commande démarre un serveur à l'aide de l'application Flask ci-dessus. Vous le démarrez sur un port étrange parce que vous ne voulez pas que les gens puissent le trouver, et vous vous félicitez d'être si sournois! Vous pouvez confirmer que cela fonctionne en visitant `http://localhost:5683` dans votre navigateur.

Puisque tout le monde dans Secret Squirrels connaît Python, vous décidez de les aider. Vous écrivez un script appelé `client.py`cela les aidera à obtenir le message secret:

```
# client.py
import os
import requests

def get_secret_message():
    url = os.environ["SECRET_URL"]
    response = requests.get(url)
    print(f"The secret message is: {response.text}")

if __name__ == "__main__":
    get_secret_message()
```

Ce code imprimera le message secret tant qu'ils auront le `SECRET_URL`jeu de variables d'environnement. Dans ce cas, le `SECRET_URL` est `127.0.0.1:5683`. Donc, votre plan est de donner à chaque membre du club l'URL secrète et de leur dire de la garder secrète et sûre .

Bien que cela puisse sembler correct, soyez assuré que ce n'est pas le cas! En fait, même si vous deviez mettre un nom d'utilisateur et un mot de passe sur ce site, ce ne serait toujours pas sûr. Mais même si votre équipe réussissait d'une manière ou d'une autre à protéger l'URL, votre message secret ne serait toujours pas sécurisé. Pour démontrer pourquoi vous devez en savoir un peu plus sur la surveillance du trafic réseau. Pour ce faire, vous utiliserez un outil appelé **Wireshark** .

Configuration de Wireshark

Wireshark est un outil largement utilisé pour l'analyse de réseau et de protocole. Cela signifie que cela peut vous aider à voir ce qui se passe sur les connexions réseau. L'installation et la configuration de Wireshark sont facultatives pour ce didacticiel, mais n'hésitez pas si vous souhaitez suivre. La page de téléchargement a plusieurs installateurs disponibles:

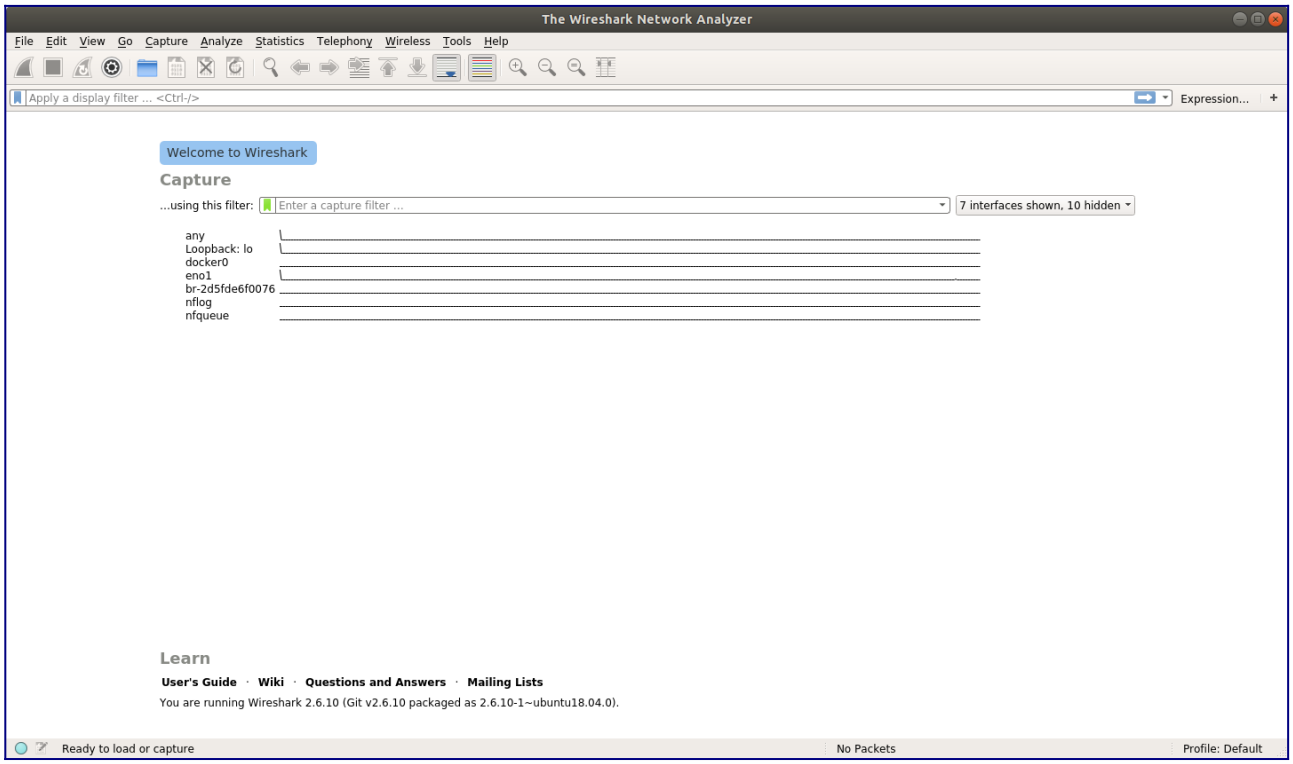
- macOS 10.12 et versions ultérieures
- Windows installer 64-bit
- Windows installer 32-bit

Si vous utilisez Windows ou Mac, vous devriez pouvoir télécharger le programme d'installation approprié et suivre les invites. En fin de compte, vous devriez avoir un Wireshark en cours d'exécution.

Si vous êtes dans un environnement Linux basé sur Debian, l'installation est un peu plus difficile, mais toujours possible. Vous pouvez installer Wireshark avec les commandes suivantes:

```
$ sudo add-apt-repository ppa:wireshark-dev/stable
$ sudo apt-get update
$ sudo apt-get install wireshark
$ sudo wireshark
```

Vous devriez être rencontré avec un écran qui ressemble à ceci:



Avec Wireshark en cours d'exécution, il est temps d'analyser le trafic!

Voir que vos données ne sont pas en sécurité

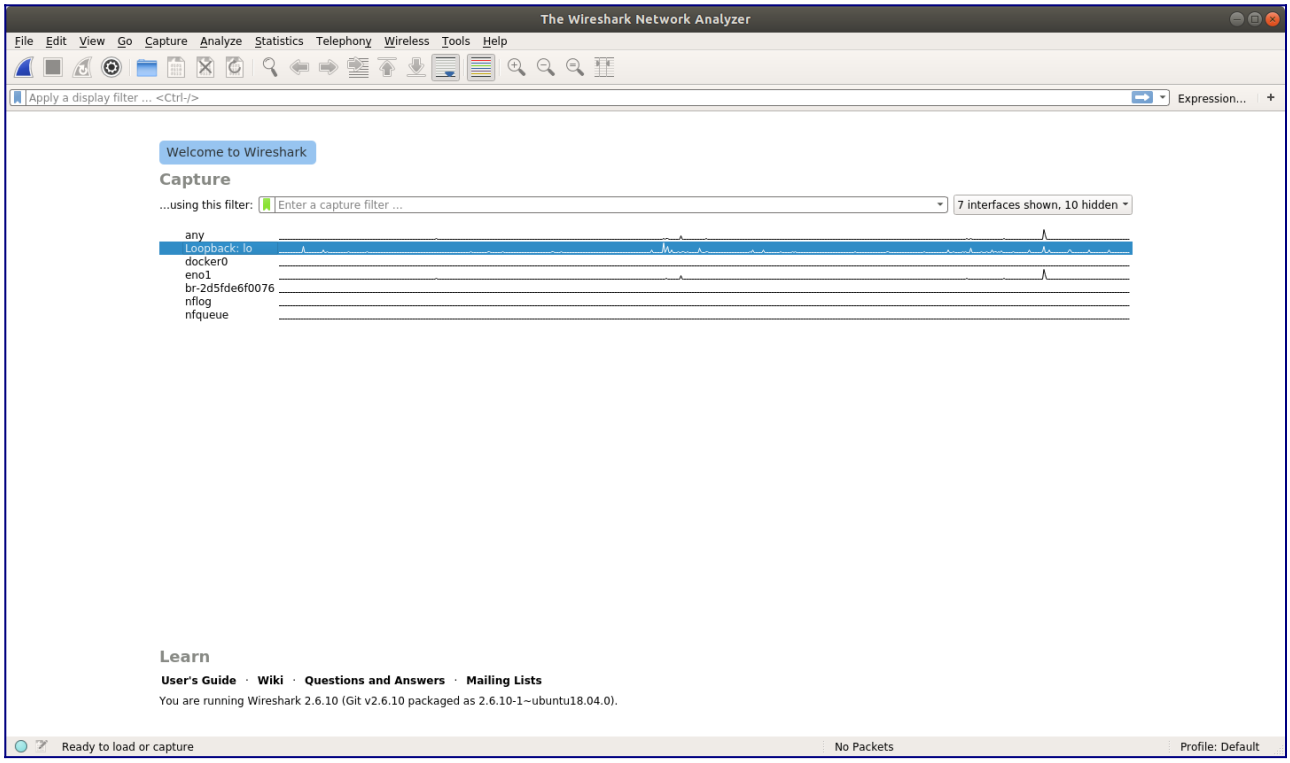
La façon dont votre client et votre serveur actuels fonctionnent n'est **pas sécurisée** . HTTP enverra tout en clair pour que tout le monde puisse le voir. Cela signifie que même si quelqu'un n'a pas votre

SECRET_URL, ils peuvent toujours voir tout ce que vous faites tant qu'ils peuvent surveiller le trafic sur *n'importe quel* appareil entre vous et le serveur.

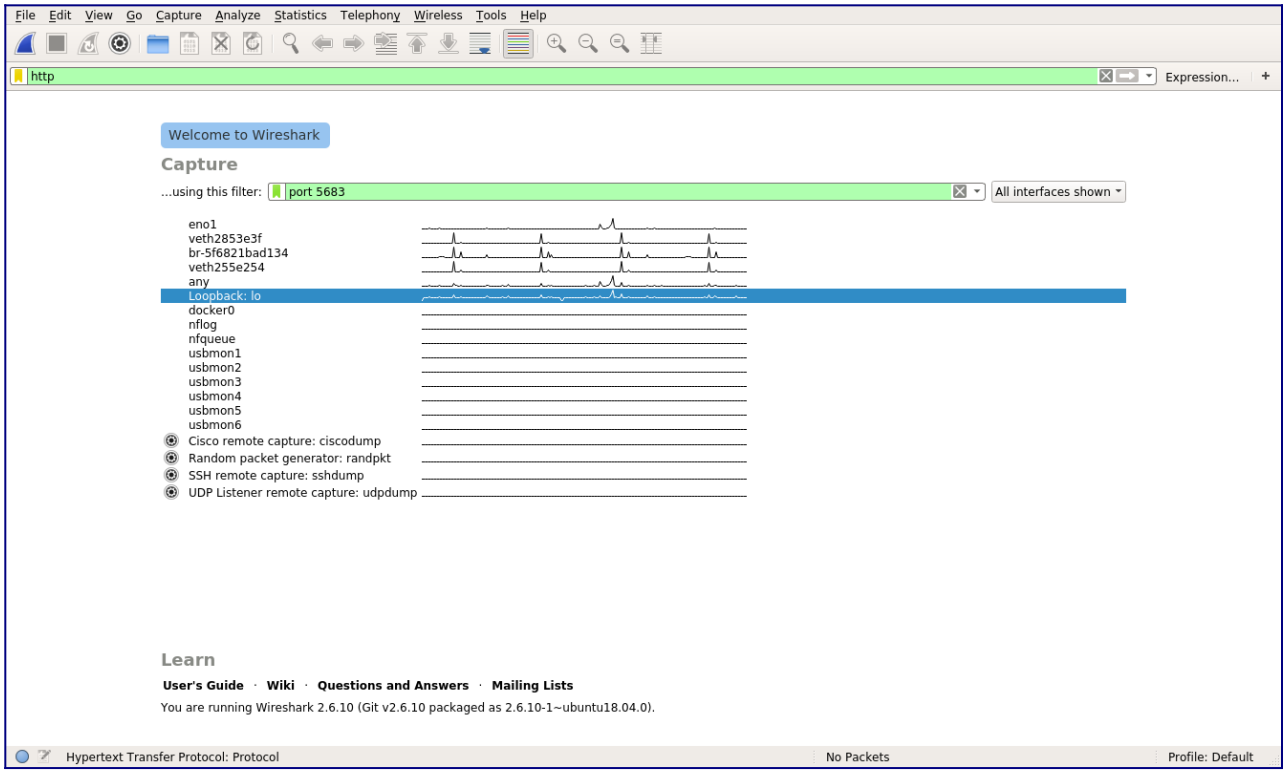
Cela devrait être relativement effrayant pour vous. Après tout, vous ne voulez pas que d'autres personnes se présentent à vos réunions Secret Squirrel! Vous pouvez prouver que cela se produit. Tout d'abord, démarrez votre serveur si vous ne l'avez pas encore en cours d'exécution:

```
$ uwsgi --http-socket 127.0.0.1:5683 --mount /=server:app
```

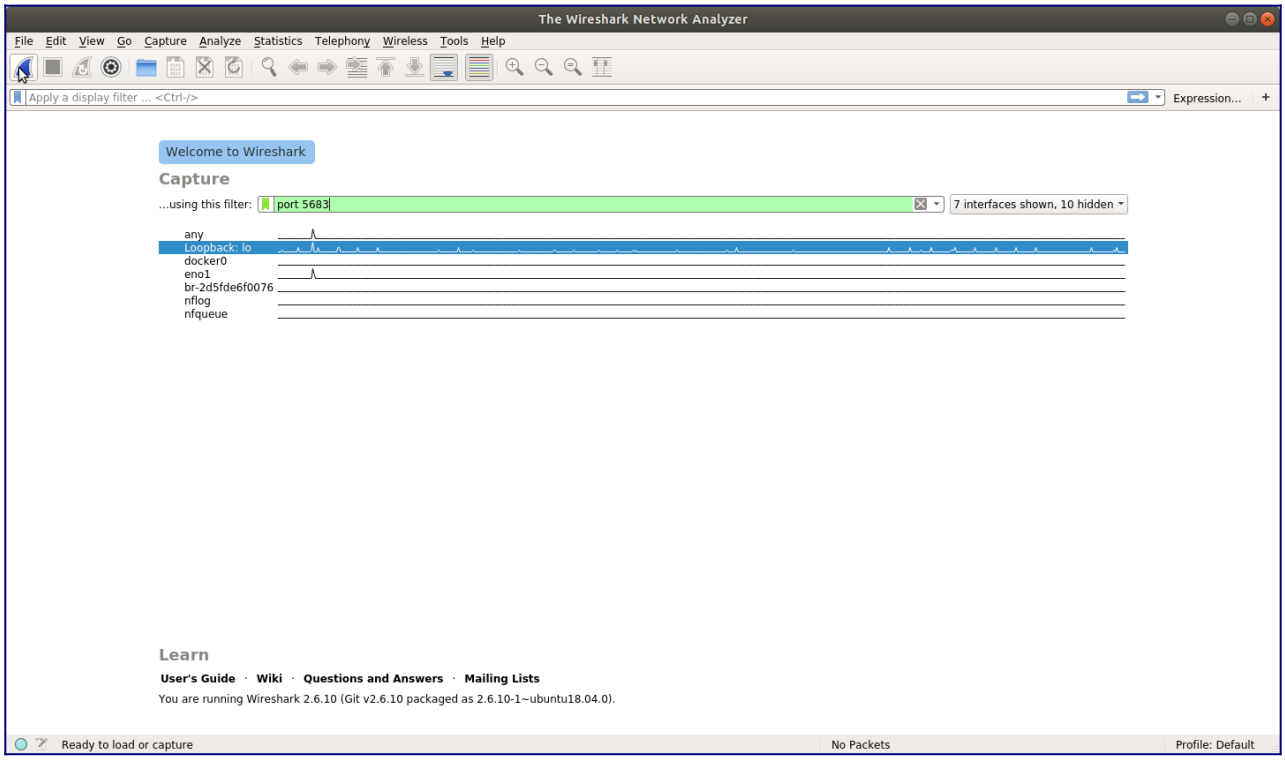
Cela démarrera votre application Flask sur le port 5683. Ensuite, vous lancerez une capture de paquets dans Wireshark. Cette capture de paquets vous aidera à voir tout le trafic entrant et sortant du serveur. Commencez par sélectionner l' *Loopback: lo* interface sur Wireshark:



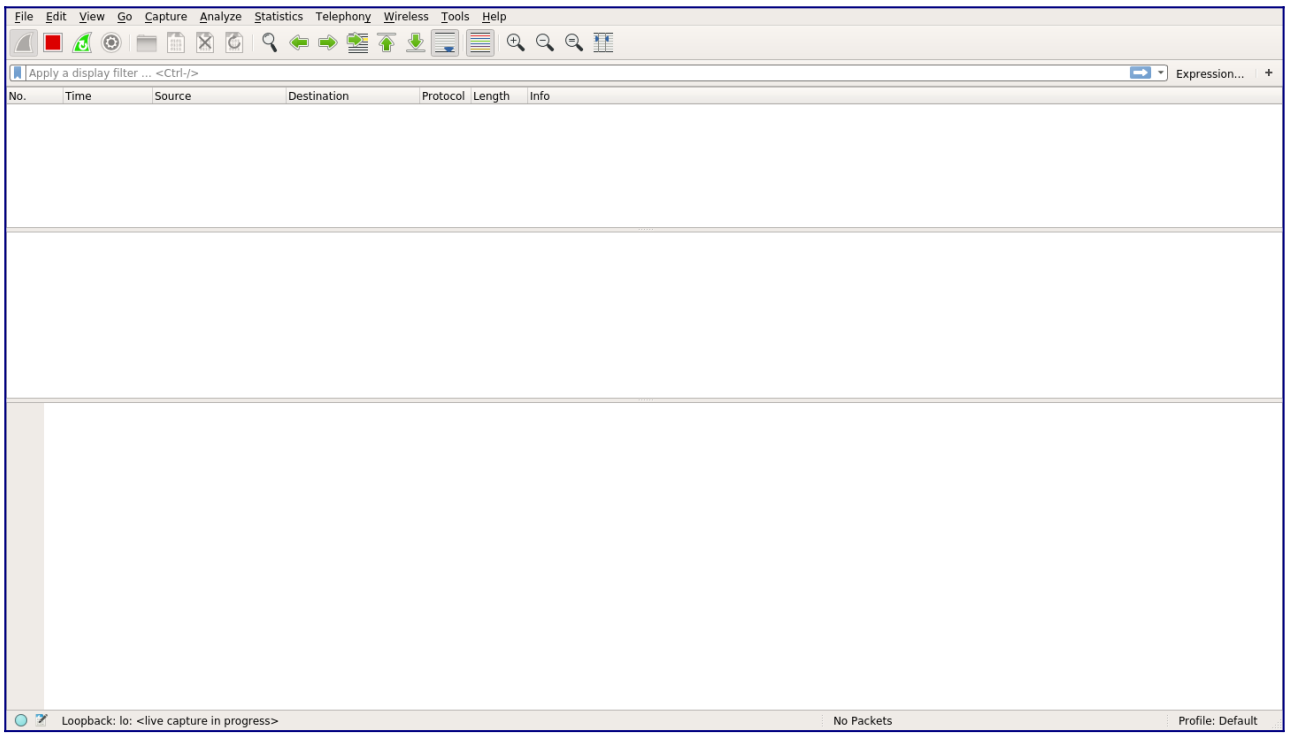
Vous pouvez voir que la partie *Loopback: lo* est en surbrillance. Cela demande à Wireshark de surveiller ce port pour le trafic. Vous pouvez faire mieux et spécifier le port et le protocole que vous souhaitez capturer. Vous pouvez taper `port 5683` dans le filtre de capture et `http` dans le filtre d'affichage:



La case verte indique que Wireshark est satisfait du filtre que vous avez saisi. Vous pouvez maintenant commencer la capture en cliquant sur l'aileron en haut à gauche:



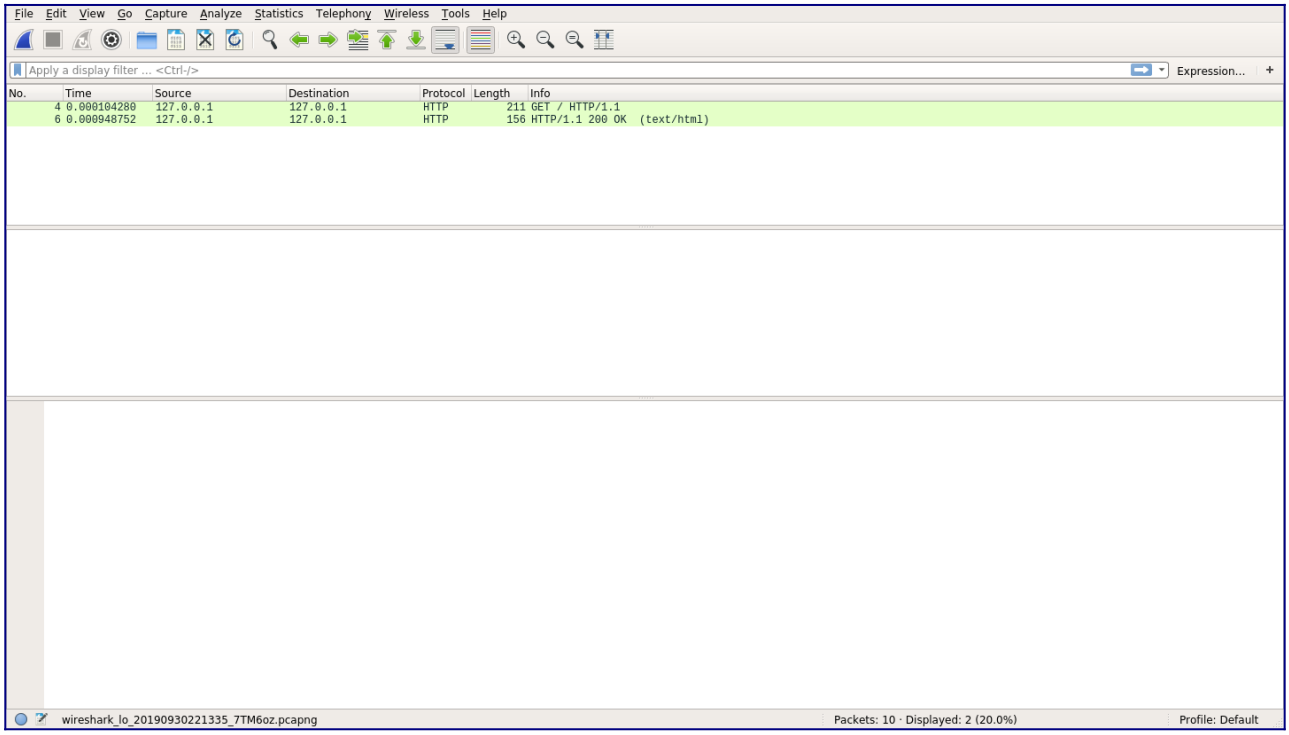
Cliquez sur ce bouton pour ouvrir une nouvelle fenêtre dans Wireshark:



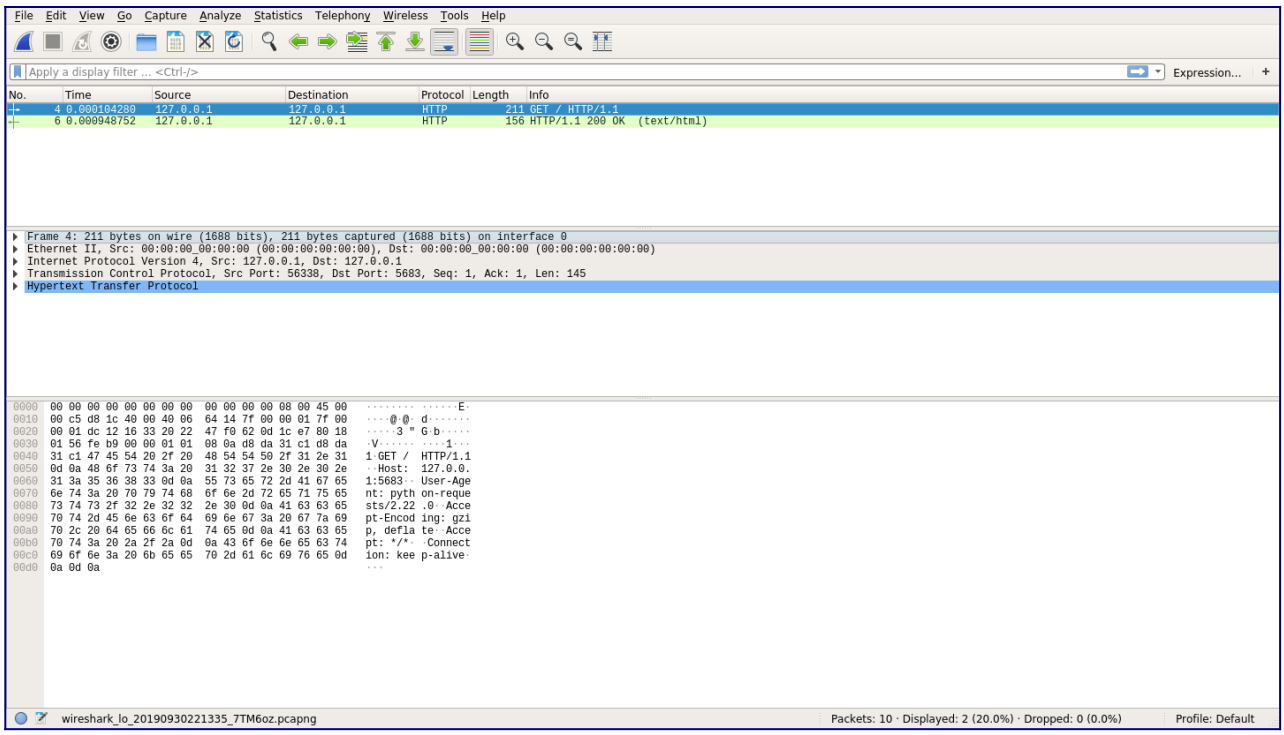
Cette nouvelle fenêtre est assez simple, mais le message en bas dit `<live capture in progress>`, ce qui indique que cela fonctionne. Ne vous inquiétez pas que rien ne s'affiche, car c'est normal. Pour que Wireshark signale quoi que ce soit, il doit y avoir une activité sur votre serveur. Pour obtenir des données, essayez d'exécuter votre client:

```
$ SECRET_URL="http://127.0.0.1:5683" python client.py
The secret message is: fluffy tail
```

Après avoir exécuté le `client.py`code ci-dessus, vous devriez maintenant voir quelques entrées dans Wireshark. Si tout s'est bien passé, vous verrez deux entrées qui ressemblent à ceci:



Ces deux entrées représentent les deux parties de la communication qui a eu lieu. Le premier est la demande du client à votre serveur. Lorsque vous cliquez sur la première entrée, vous verrez une pléthore d'informations:

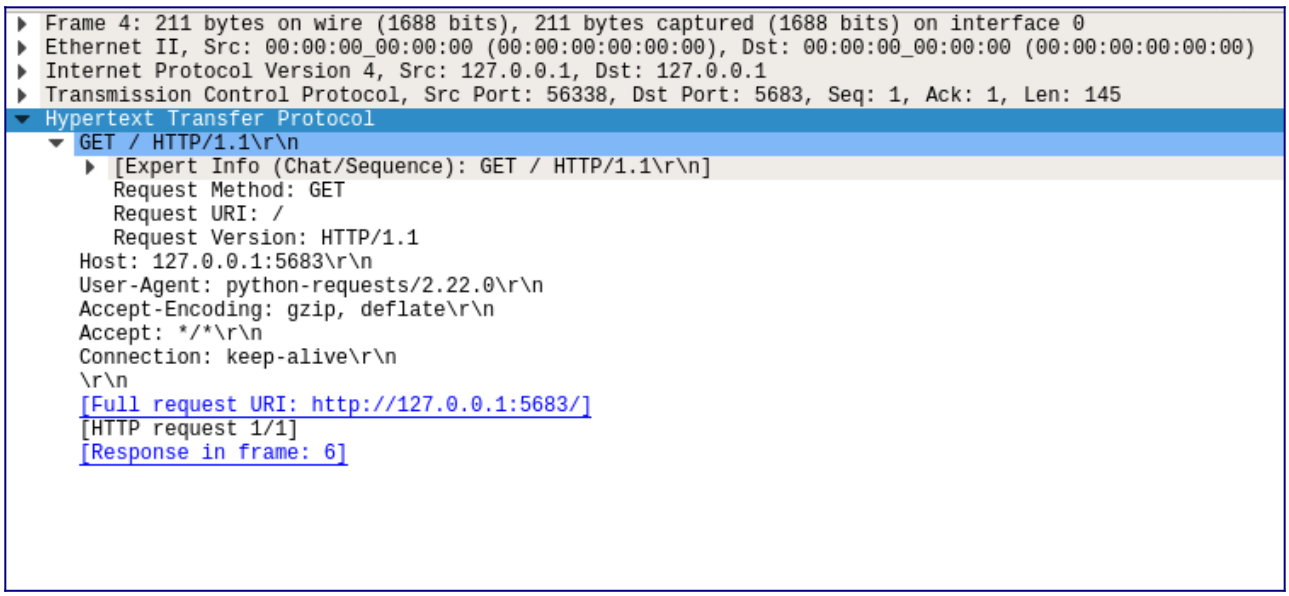


C'est beaucoup d'informations! En haut, vous avez toujours votre requête et réponse HTTP. Une fois que vous sélectionnez l'une de ces entrées, vous verrez les lignes du milieu et du bas se remplir d'informations.

La ligne du milieu vous fournit une ventilation des protocoles que Wireshark a pu identifier pour la demande sélectionnée. Cette ventilation vous permet d'explorer ce qui s'est réellement passé dans votre requête HTTP. Voici un bref résumé des informations que Wireshark décrit dans la rangée du milieu de haut en bas:

1. **Couche physique:** cette ligne décrit l'interface physique utilisée pour envoyer la demande. Dans votre cas, il s'agit probablement de l'ID d'interface 0 (lo) pour votre interface de bouclage.
2. **Informations Ethernet:** cette ligne vous montre le protocole Layer-2, qui comprend les adresses MAC source et de destination.
3. **IPv4:** cette ligne affiche les adresses IP source et de destination (127.0.0.1).
4. **TCP:** Cette ligne inclut la prise de contact TCP requise afin de créer un canal de données fiable.
5. **HTTP:** cette ligne affiche des informations sur la requête HTTP elle-même.

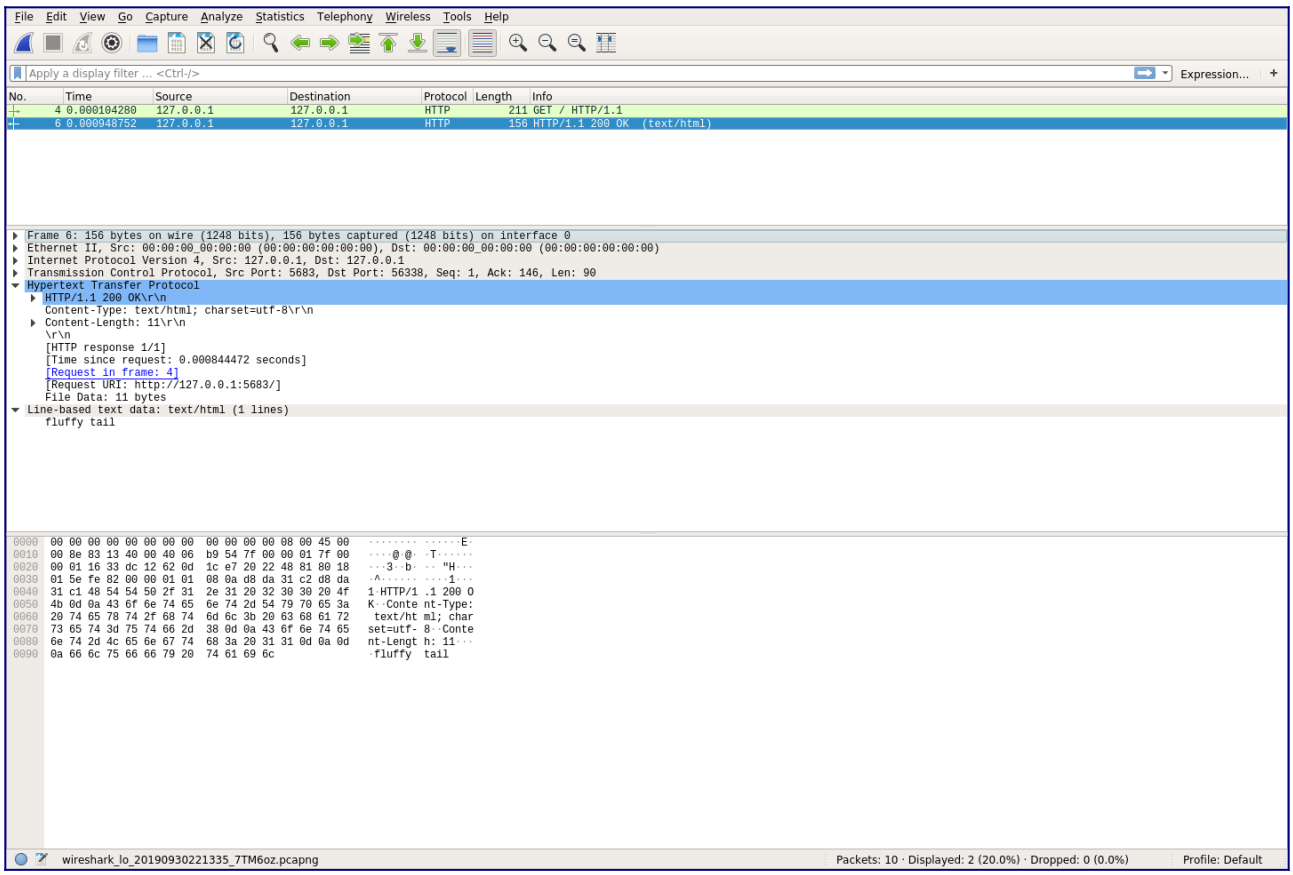
Lorsque vous développez la couche Hypertext Transfer Protocol, vous pouvez voir toutes les informations qui composent une requête HTTP:



Cette image vous montre la requête HTTP de votre script:

- **Méthode:** `GET`
- **Chemin:** `/`
- **Version:** `1.1`
- **En-têtes:** `Host: 127.0.0.1:5683`, `Connection: keep-alive`, et d'autres
- **Corps:** pas de corps

La dernière ligne que vous verrez est un vidage hexadécimal des données. Vous remarquerez peut-être dans ce vidage hexadécimal que vous pouvez réellement voir les parties de votre requête HTTP. C'est parce que votre requête HTTP a été envoyée à l'air libre. Mais qu'en est-il de la réponse? Si vous cliquez sur la réponse HTTP, vous verrez une vue similaire:



Encore une fois, vous avez les trois mêmes sections. Si vous regardez attentivement le vidage hexadécimal, vous verrez le message secret en texte brut! C'est un gros problème pour les écureuils secrets. Cela signifie que toute personne ayant un certain savoir-faire technique peut très facilement voir ce trafic si elle est intéressée. Alors, comment résolvez-vous ce problème? La réponse est la **cryptographie** .

Comment la cryptographie aide-t-elle?

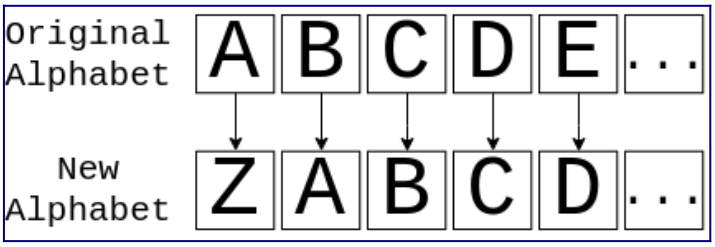
Dans cette section, vous découvrirez une façon de protéger vos données en créant vos propres **clés de cryptographie** et en les utilisant à la fois sur votre serveur et votre client. Bien que ce ne soit pas votre dernière étape, cela vous aidera à obtenir une base solide sur la façon de créer des applications Python HTTPS.

Comprendre les bases de la cryptographie

La cryptographie est un moyen de sécuriser les communications des écoutes indiscrètes ou des adversaires. Une autre façon de le dire est que vous prenez des informations normales, appelées **texte en clair** , et que vous les convertissez en texte brouillé, appelé **texte chiffré** .

La cryptographie peut être intimidante au début, mais les concepts fondamentaux sont assez accessibles. En fait, vous avez probablement déjà pratiqué la cryptographie auparavant. Si vous avez déjà eu un langage secret avec vos amis et que vous l'avez utilisé pour passer des notes en classe, alors vous avez pratiqué la cryptographie. (Si vous ne l'avez pas fait, ne vous inquiétez pas, vous êtes sur le point de le faire.)

D'une manière ou d'une autre, tu dois prendre la ficelle `"fluffy tail"`et le convertir en quelque chose d'inintelligible. Une façon de faire est de mapper certains personnages sur différents personnages. Un moyen efficace pour ce faire est de décaler les caractères d'un endroit dans l'alphabet. Cela ressemblerait à quelque chose comme ceci:



Cette image vous montre comment traduire de l'alphabet d'origine au nouvel alphabet et inversement. Donc, si tu avais le message `ABC`, alors vous enverriez réellement le message `ZAB`. Si vous appliquez ceci à `"fluffy tail"`, alors en supposant que les espaces restent les mêmes, vous obtenez `ekteex szhk`. Bien que ce ne soit pas parfait, cela ressemblera probablement à du charabia pour quiconque le verra.

Toutes nos félicitations! Vous avez créé ce que l'on appelle en cryptographie un **chiffrement** , qui décrit comment convertir du texte brut en texte chiffré et inversement. Votre chiffre, dans ce cas, est décrit en anglais. Ce type particulier de chiffrement est appelé un **chiffrement de substitution** . Fondamentalement, il s'agit du même type de chiffrement utilisé dans Enigma Machine , même s'il s'agit d'une version beaucoup plus simple.

Maintenant, si vous voulez faire passer un message aux écureuils secrets, vous devez d'abord leur dire combien de lettres doivent être décalées, puis leur donner le message codé. En Python, cela peut ressembler à ceci:

```
CIPHER = {"a": "z", "A": "Z", "b": "a"} # And so on

def encrypt(plaintext: str):
    return "".join(CIPHER.get(letter, letter) for letter in plaintext)
```

Ici, vous avez créé une fonction appelée `encrypt()`, qui prendra du texte en clair et le convertira en texte chiffré. Imaginez que vous ayez un dictionnaire `CIPHER`qui a tous les personnages cartographiés. De même, vous pouvez créer un `decrypt()`:

```
DECIPHER = {v: k for k, v in CIPHER.items()}

def decrypt(ciphertext: str):
    return "".join(DECIPHER.get(letter, letter) for letter in ciphertext)
```

Cette fonction est l'opposé de `encrypt()`. Il prendra du texte chiffré et le convertira en texte brut. Dans cette forme de chiffrement, vous disposez d'une clé spéciale que les utilisateurs doivent connaître pour crypter et décrypter les messages. Pour l'exemple ci-dessus, cette clé est `1`. Autrement dit, le chiffre indique que vous devez reculer chaque lettre d'un caractère. Il est très important de garder la clé secrète car toute personne possédant la clé peut facilement déchiffrer votre message.

Remarque: bien que vous puissiez l'utiliser pour votre cryptage, ce n'est toujours pas très sécurisé. Ce chiffre est rapide à casser en utilisant l' analyse de fréquence et est beaucoup trop primitif pour les écureuils secrets.

À l'ère moderne, la cryptographie est beaucoup plus avancée. Il s'appuie sur une théorie mathématique complexe et sur l'informatique pour être sécurisé. Bien que les calculs derrière ces chiffrements sortent du cadre de ce didacticiel, les concepts sous-jacents sont toujours les mêmes. Vous disposez d'un chiffrement qui décrit comment prendre du texte brut et le convertir en texte chiffré.

La seule vraie différence entre votre chiffrement de substitution et les chiffrements modernes est qu'il est mathématiquement prouvé que les chiffrements modernes sont irréalisables par une écoute indiscrète. Voyons maintenant comment utiliser vos nouveaux chiffrements.

Utilisation de la cryptographie dans les applications Python HTTPS

Heureusement pour vous, vous n'avez pas besoin d'être un expert en mathématiques ou en informatique pour utiliser la cryptographie. Python a également un `secrets`module qui peut vous aider à générer des données aléatoires cryptographiquement sécurisées . Dans ce didacticiel, vous découvrirez une bibliothèque Python qui porte bien son nom `cryptography`. Il est disponible sur PyPI, vous pouvez donc l'installer avec pip:

```
$ pip install cryptography
```

Cela installera `cryptography`dans votre environnement virtuel . Avec `cryptography` installé, vous pouvez désormais crypter et décrypter les choses d'une manière mathématiquement sécurisée en utilisant le `Fernet` méthode.

Rappelez-vous que votre clé secrète dans votre chiffrement était `1`. Dans le même ordre d'idées, vous devez créer une clé pour que Fernet fonctionne correctement:

```
>>> from cryptography.fernet import Fernet
>>> key = Fernet.generate_key()
>>> key
b'8jtTR9QcD-k3RO9Pcd5ePgmTu_itJQt9WKQPzqjrcoM='
```

Dans ce code, vous avez importé `Fernet`et a généré une clé. La clé est juste un tas d'octets, mais il est extrêmement important que vous gardiez cette clé secrète et sûre. Tout comme l'exemple de substitution ci-dessus, toute personne possédant cette clé peut facilement déchiffrer vos messages.

Remarque: dans la vraie vie, vous garderiez cette clé **très** sécurisée. Dans ces exemples, il est utile de voir la clé, mais c'est une mauvaise pratique, surtout si vous la publiez sur un site Web public! En d'autres termes, **n'utilisez pas la clé exacte que vous voyez ci-dessus** pour tout ce que vous souhaitez sécuriser.

Cette clé se comporte un peu comme la clé précédente. Il est nécessaire de faire la transition vers le texte chiffré et de revenir au texte brut. Il est maintenant temps pour la partie amusante! Vous pouvez crypter un message comme celui-ci:

```
>>> my_cipher = Fernet(key)
>>> ciphertext = my_cipher.encrypt(b"fluffy tail")
>>> ciphertext
b'gAAAAABdlW033LxsrnmA2P0WzaS-wk1UKXA1IdyDpmHcV6yrE7H_ApmSK8KpCW-
6jaODFaeTeDRKJMMsa_526koApx1suJ4_dQ=='
```

Dans ce code, vous avez créé un objet Fernet appelé `my_cipher`, que vous pouvez ensuite utiliser pour crypter votre message. Notez que votre message secret `"fluffy tail"` doit être un `bytes`objet afin de le crypter. Après le cryptage, vous pouvez voir que le `ciphertext` est un long flux d'octets.

Grâce à Fernet, ce texte chiffré ne peut pas être manipulé ou lu sans la clé! Ce type de cryptage nécessite que le serveur et le client aient accès à la clé. Lorsque les deux côtés nécessitent la même clé, cela s'appelle un cryptage symétrique . Dans la section suivante, vous verrez comment utiliser ce cryptage symétrique pour protéger vos données.

Vérifier que vos données sont en sécurité

Maintenant que vous comprenez certaines des bases de la cryptographie en Python, vous pouvez appliquer ces connaissances à votre serveur. Créez un nouveau fichier appelé `symmetric_server.py`:

```
# symmetric_server.py
import os
from flask import Flask
from cryptography.fernet import Fernet

SECRET_KEY = os.environb[b"SECRET_KEY"]
SECRET_MESSAGE = b"fluffy tail"
app = Flask(__name__)

my_cipher = Fernet(SECRET_KEY)

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

Ce code combine votre code serveur d'origine avec le `Fernet`objet que vous avez utilisé dans la section précédente. La clé est maintenant lue comme un `bytes` objet de l'environnement en utilisant `os.environb`. Avec le serveur à l'écart, vous pouvez maintenant vous concentrer sur le client. Collez ce qui suit dans `symmetric_client.py`:

```
# symmetric_client.py
import os
import requests
from cryptography.fernet import Fernet

SECRET_KEY = os.environb[b"SECRET_KEY"]
my_cipher = Fernet(SECRET_KEY)

def get_secret_message():
    response = requests.get("http://127.0.0.1:5683")

    decrypted_message = my_cipher.decrypt(response.content)
    print(f"The codeword is: {decrypted_message}")

if __name__ == "__main__":
    get_secret_message()
```

Encore une fois, il s'agit d'un code modifié pour combiner votre client précédent avec le `Fernet` mécanisme de cryptage. `get_secret_message()` fait ce qui suit:

1. **Faites** la demande à votre serveur.
2. **Prenez** les octets bruts de la réponse.
3. **Essayez** de déchiffrer les octets bruts.
4. **Imprimez** le message déchiffré.

Si vous exécutez à la fois le serveur et le client, vous verrez que vous avez réussi à chiffrer et déchiffrer votre message secret:

```
$ uwsgi --http-socket 127.0.0.1:5683 \
    --env SECRET_KEY="8jtTR9QcD-k3RO9Pcd5ePgmTu_itJQt9WKQPzqjrcoM=" \
    --mount /=symmetric_server:app
```

Dans cet appel, vous redémarrez le serveur sur le port 5683. Cette fois, tu passes dans un SECRET_KEYqui doit être au moins une chaîne codée en base64 de 32 longueurs. Une fois votre serveur redémarré, vous pouvez maintenant l'interroger:

```
$ SECRET_KEY="8jtTR9QcD-k3RO9Pcd5ePgmTu_itJQt9WKQPzqjrcoM=" python
symmetric_client.py
The secret message is: b'fluffy tail'
```

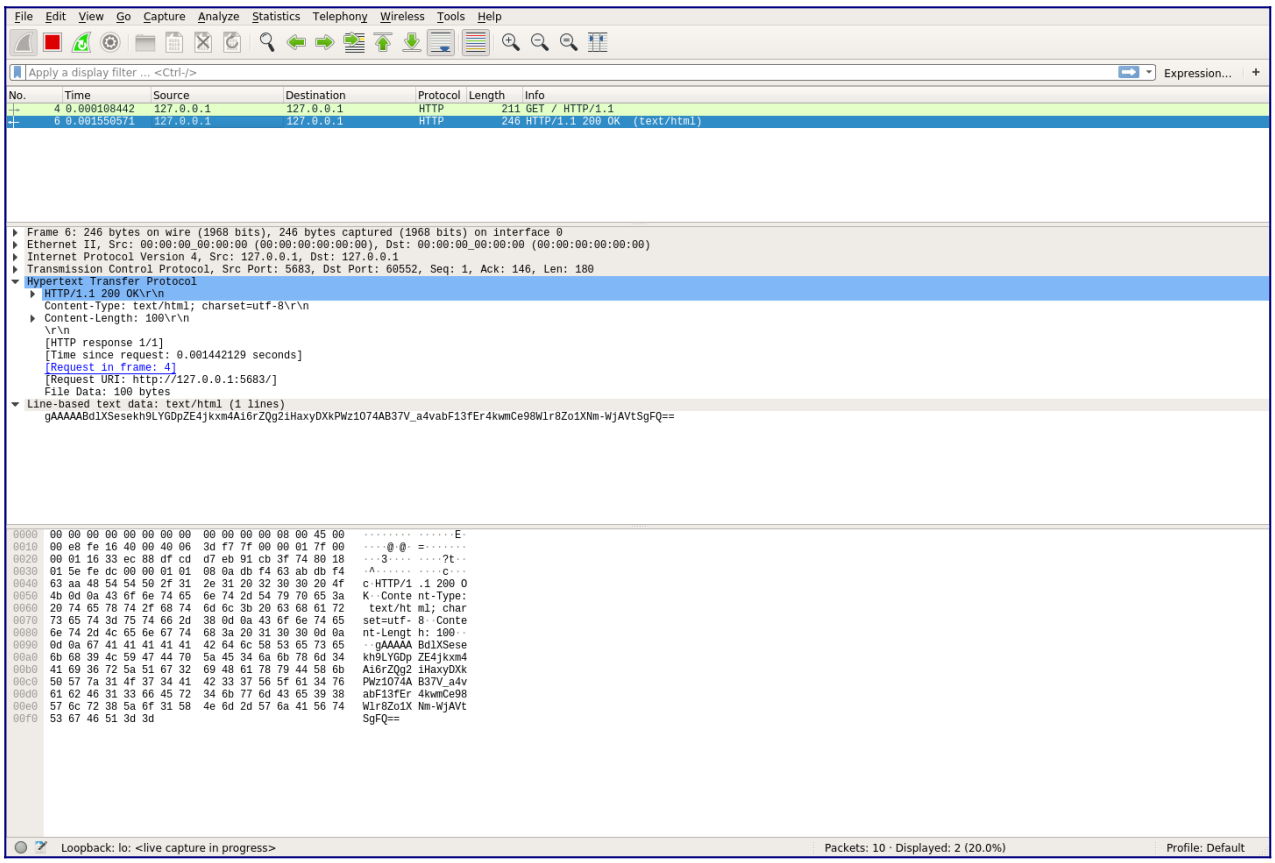
Woohoo! Vous avez pu crypter et décrypter votre message. Si vous essayez d'exécuter ceci avec un SECRET_KEY, alors vous obtiendrez une erreur:

```
$ SECRET_KEY="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=" python
symmetric_client.py
Traceback (most recent call last):
  File ".../cryptography/fernet.py", line 104, in _verify_signature
    h.verify(data[-32:])
  File ".../cryptography/hazmat/primitives/hmac.py", line 66, in verify
    ctx.verify(signature)
  File ".../cryptography/hazmat/backends/openssl/hmac.py", line 74, in verify
    raise InvalidSignature("Signature did not match digest.")
cryptography.exceptions.InvalidSignature: Signature did not match digest.

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "symmetric_client.py", line 16, in <module>
    get_secret_message()
  File "symmetric_client.py", line 11, in get_secret_message
    decrypted_message = my_cipher.decrypt(response.content)
  File ".../cryptography/fernet.py", line 75, in decrypt
    return self._decrypt_data(data, timestamp, ttl)
  File ".../cryptography/fernet.py", line 117, in _decrypt_data
    self._verify_signature(data)
  File ".../cryptography/fernet.py", line 106, in _verify_signature
    raise InvalidToken
cryptography.fernet.InvalidToken
```

Ainsi, vous savez que le cryptage et le décryptage fonctionnent. Mais est-ce **sécurisé** ? Eh bien, oui, ça l'est. Pour le prouver, vous pouvez revenir à Wireshark et démarrer une nouvelle capture avec les mêmes filtres qu'auparavant. Une fois la configuration de la capture terminée, exécutez à nouveau le code client:

```
$ SECRET_KEY="8jtTR9QcD-k3RO9Pcd5ePgmTu_itJQt9WKQPzqjrcoM=" python
symmetric_client.py
The secret message is: b'fluffy tail'
```

Vous avez effectué une autre requête et réponse HTTP réussies, et une fois de plus, vous voyez ces messages dans Wireshark. Étant donné que le message secret n'est transféré que dans la réponse, vous pouvez cliquer dessus pour consulter les données:



Dans la rangée du milieu de cette image, vous pouvez voir les données qui ont été réellement transférées:

```
gAAAAABdlXSesekh9LYGDpZE4jkxm4Ai6rZQg2iHaxyDXkPWz1O74AB37V_a4vabF13fEr4kwmCe98Wlr8Zo1XNm-WjAVtSgFQ==
```

Impressionnant! Cela signifie que les données ont été cryptées et que les espions indiscrets n'ont aucune idée de ce qu'est réellement le contenu du message. Non seulement cela, mais cela signifie également qu'ils pourraient passer un temps incroyablement long à essayer de casser ces données par force brute, et qu'ils ne réussiraient presque jamais.

Vos données sont en sécurité! Mais attendez une minute - vous n'aviez jamais rien à savoir sur une clé lorsque vous utilisiez des applications Python HTTPS auparavant. C'est parce que HTTPS n'utilise pas exclusivement le cryptage symétrique. Il s'avère que partager des secrets est un problème difficile.

Pour prouver ce concept, accédez à `http://127.0.0.1:5683`dans votre navigateur et vous verrez le texte de réponse chiffré. En effet, votre navigateur ne sait rien de votre clé de chiffrement secrète. Alors, comment les applications Python HTTPS fonctionnent-elles vraiment? C'est là que le **cryptage asymétrique** entre en jeu.

Comment les clés sont-elles partagées?

Dans la section précédente, vous avez vu comment vous pouvez utiliser le cryptage symétrique pour sécuriser vos données lorsqu'elles traversent Internet. Pourtant, même si le cryptage symétrique est sécurisé, ce n'est pas la seule technique de cryptage utilisée par les applications Python HTTPS pour protéger vos données. Le cryptage symétrique introduit des problèmes fondamentaux qui ne sont pas si faciles à résoudre.

Remarque: n'oubliez pas que le chiffrement symétrique nécessite que vous disposiez d'une **clé partagée** entre le client et le serveur. Malheureusement, la sécurité ne fonctionne aussi dur que votre lien le plus faible, et les liens faibles sont particulièrement catastrophiques dans le cryptage symétrique. Une fois qu'une personne a compromis la clé, **chaque clé est compromise** . Il est prudent de supposer que tout système de sécurité sera, à un moment donné, compromis.

Alors, comment vous **changez-** votre clé? Si vous n'avez qu'un serveur et un client, cela peut être une tâche rapide. Cependant, à mesure que vous obtenez de plus en plus de clients et de serveurs, il y a de plus en plus de coordination qui doit se produire afin de changer la clé et de protéger efficacement vos secrets.

De plus, vous devez à chaque fois choisir un nouveau secret. Dans l'exemple ci-dessus, vous avez vu une clé générée aléatoirement. Il peut être pratiquement impossible pour vous d'essayer d'amener les gens à se souvenir de cette clé. Au fur et à mesure que votre nombre de clients et de serveurs augmente, vous utiliserez probablement des clés plus faciles à mémoriser et à deviner.

Si vous pouvez changer votre clé, vous avez encore un problème à résoudre. Comment vous **partagez-** votre clé initiale? Dans l'exemple Secret Squirrels, vous avez résolu ce problème en ayant un accès physique à chacun des membres. Vous pouvez donner à chaque membre le secret en personne et lui dire de le garder secret, mais rappelez-vous que quelqu'un sera le maillon le plus faible.

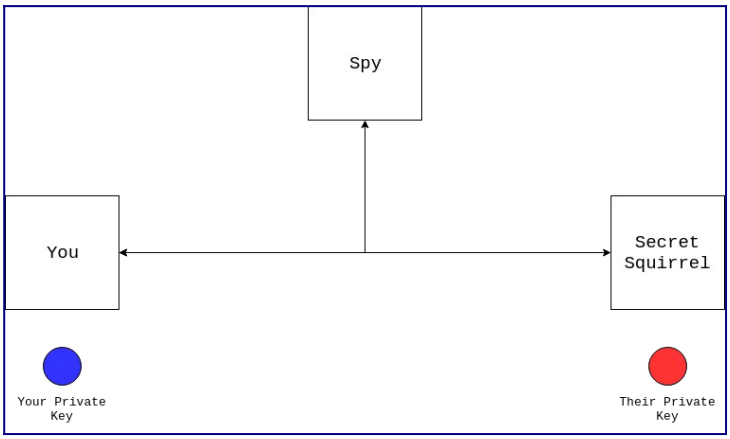
Maintenant, supposons que vous ajoutiez un membre aux écureuils secrets depuis un autre emplacement physique. Comment partagez-vous le secret avec ce membre? Les faites-vous prendre un avion pour vous à chaque fois que la clé change? Ce serait bien si vous pouviez mettre la clé secrète sur votre serveur et la partager automatiquement. Malheureusement, cela irait à l'encontre de l'objectif du cryptage, car n'importe qui pourrait obtenir la clé secrète!

Bien sûr, vous pouvez donner à tout le monde une clé principale initiale pour obtenir le message secret, mais maintenant vous avez juste deux fois plus de problèmes qu'avant. Si vous avez mal à la tête, ne vous inquiétez pas! Tu n'es pas le seul.

Ce dont vous avez besoin, c'est que deux parties qui n'ont jamais communiqué ont un secret partagé. Cela semble impossible, non? Heureusement, trois gars du nom de Ralph Merkle , Whitfield Diffie et Martin Hellman vous soutiennent. Ils ont aidé à démontrer que la cryptographie à clé publique, également connue sous le nom de cryptage asymétrique , était possible.

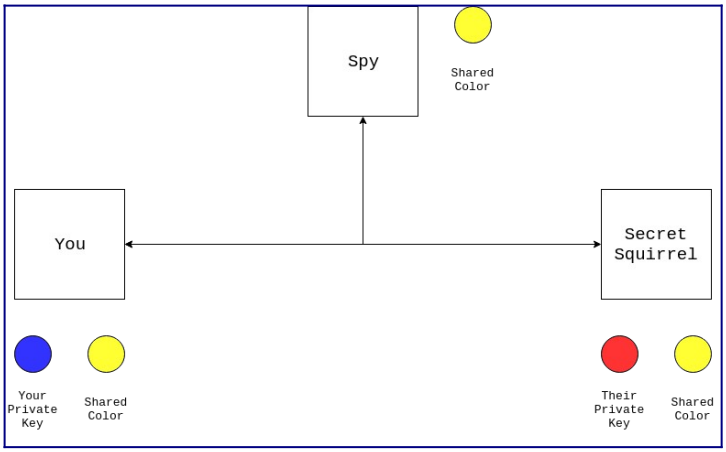
Remarque: alors que Whitfield Diffie et Martin Hellman sont largement connus pour être les premiers à découvrir le schéma, il a été révélé en 1997 que trois hommes travaillant pour le GCHQ nommés James H.Ellis , Clifford Cocks et Malcolm J. Williamson avaient déjà démontré cette capacité sept fois. des années plus tôt!

Le cryptage asymétrique permet à deux utilisateurs qui n'ont jamais communiqué auparavant de partager un secret commun. L'un des moyens les plus simples de comprendre les principes fondamentaux consiste à utiliser une analogie des couleurs. Imaginez que vous ayez le scénario suivant:

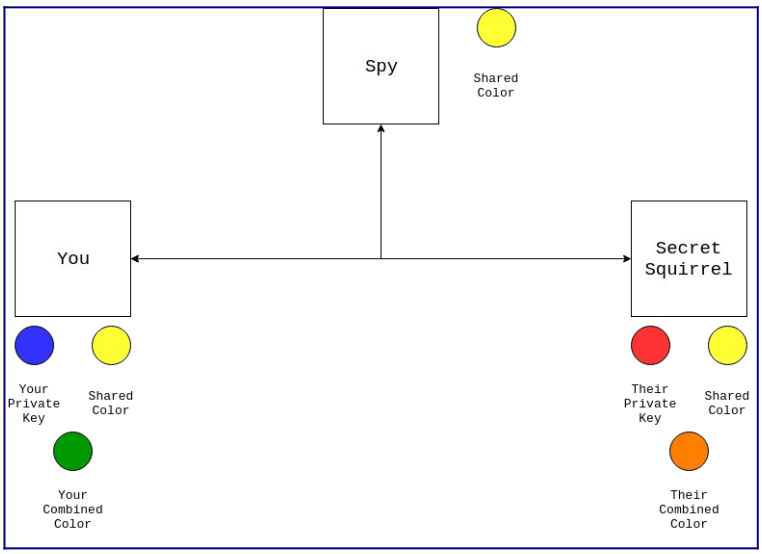


Dans ce diagramme, vous essayez de communiquer avec un écureuil secret que vous n'avez jamais rencontré auparavant, mais un espion peut voir tout ce que vous envoyez. Vous connaissez le cryptage symétrique et souhaitez l'utiliser, mais vous devez d'abord partager un secret. Heureusement, vous avez tous les deux une clé privée. Malheureusement, vous ne pouvez pas envoyer votre clé privée car l'espion la verra. Donc que fais-tu?

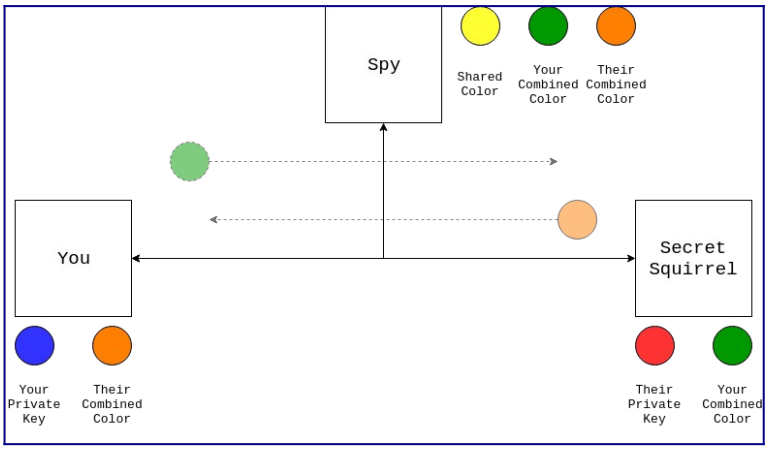
La première chose à faire est de vous mettre d'accord avec votre partenaire sur une couleur, comme le jaune:



Remarquez ici que l'espion peut voir la couleur partagée, tout comme vous et l'écureuil secret. La couleur partagée est effectivement **publique** . Maintenant, vous et l'écureuil secret combinez vos **privées** clés avec la couleur partagée:



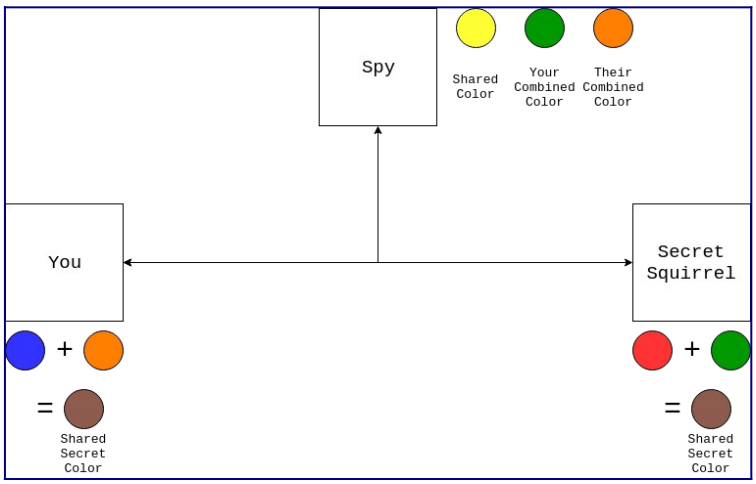
Vos couleurs se combinent pour faire du vert, tandis que les couleurs de l'écureuil secret se combinent pour faire de l'orange. Vous avez tous les deux terminé avec la couleur partagée, et vous devez maintenant partager vos couleurs combinées les unes avec les autres:



Vous avez maintenant votre clé privée et la couleur combinée de l'écureuil secret. De même, l'écureuil secret a sa clé privée et votre couleur combinée. C'était assez rapide pour vous et l'écureuil secret de combiner vos couleurs.

L'espion, cependant, n'a que ces couleurs combinées. Essayer de déterminer votre *exacte* couleur d'origine est très difficile, même compte tenu de la couleur initiale partagée. L'espion devrait aller au magasin et acheter plein de blues différents pour essayer. Même dans ce cas, il serait difficile de savoir s'ils regardaient la bonne nuance de vert après la combinaison! En bref, votre clé privée est toujours **privée** .

Mais qu'en est-il de vous et de l'écureuil secret? Vous n'avez toujours pas de secret combiné! C'est là que votre clé privée revient. Si vous combinez votre clé privée avec la couleur combinée que vous avez reçue de l'écureuil secret, vous vous retrouverez tous les deux avec la même couleur:



Maintenant, vous et l'écureuil secret avez la même couleur secrète partagée. Vous avez maintenant partagé avec succès un secret sécurisé avec un parfait inconnu. Ceci est étonnamment précis par rapport au fonctionnement de la cryptographie à clé publique. Un autre nom courant pour cette séquence d'événements est l' **échange de clés Diffie-Hellman** . L'échange de clés se compose des éléments suivants:

- **La clé privée** est votre couleur privée parmi les exemples.
- **La clé publique** est la couleur combinée que vous avez partagée.

La clé privée est quelque chose que vous gardez toujours privé, tandis que la clé publique peut être partagée avec n'importe qui. Ces concepts correspondent directement au monde réel des applications Python HTTPS. Maintenant que le serveur et le client ont un secret partagé, vous pouvez utiliser votre ancien cryptage symétrique copain pour crypter tous les autres messages!

Remarque: la cryptographie à clé publique repose également sur des calculs mathématiques pour effectuer le mélange des couleurs. La page Wikipédia pour l'échange de clés Diffie-Hellman a une bonne explication, mais une explication approfondie sort du cadre de ce didacticiel.

Lorsque vous communiquez sur un site Web sécurisé, comme celui-ci, votre navigateur et le serveur mettent en place une communication sécurisée en utilisant les mêmes principes:

1. Votre navigateur **demande des** informations au serveur.
2. Votre navigateur et le serveur **échangent** des clés publiques.
3. Votre navigateur et le serveur **génèrent** une clé privée partagée.
4. Votre navigateur et le serveur **chiffrent et déchiffrent les** messages à l'aide de cette clé partagée via un chiffrement symétrique.

Heureusement pour vous, vous n'avez besoin de mettre en œuvre aucun de ces détails. Il existe de nombreuses bibliothèques intégrées et tierces qui peuvent vous aider à sécuriser vos communications client et serveur.

À quoi ressemble HTTPS dans le monde réel?

Compte tenu de toutes ces informations sur le chiffrement, faisons un zoom arrière et parlons de la façon dont les applications Python HTTPS fonctionnent réellement dans le monde réel. Le chiffrement n'est que la moitié de l'histoire. Lors de la visite d'un site Web sécurisé, deux composants principaux sont nécessaires:

1. **Le chiffrement** convertit le texte brut en texte chiffré et inversement.
2. **L'authentification** vérifie qu'une personne ou une chose est bien ce qu'elle prétend être.

Vous avez beaucoup entendu parler du fonctionnement du cryptage, mais qu'en est-il de l'authentification? Pour comprendre l'authentification dans le monde réel, vous devez connaître **l'infrastructure à clé publique** . PKI introduit un autre concept important dans l'écosystème de sécurité, appelé **certificats** .

Les certificats sont comme des passeports pour Internet. Comme la plupart des choses dans le monde informatique, ce ne sont que des morceaux de données dans un fichier. De manière générale, les certificats contiennent les informations suivantes:

- **Délivré à:** identifie à qui appartient le certificat
- **Émis par:** identifie qui a émis le certificat
- **Période de validité:** identifie la période pendant laquelle le certificat est valide

Tout comme les passeports, les certificats ne sont vraiment utiles que s'ils sont générés et reconnus par une autorité. Il n'est pas pratique pour votre navigateur de connaître chaque certificat de chaque site que vous visitez sur Internet. Au lieu de cela, PKI repose sur un concept connu sous le nom **d'autorités de certification (CA)** .

Les autorités de certification sont responsables de la délivrance des certificats. Ils sont considérés comme des tiers de confiance (TTP) dans PKI. Essentiellement, ces entités agissent en tant qu'autorités valides pour un certificat. Supposons que vous souhaitiez visiter un autre pays et que vous ayez un passeport contenant toutes vos informations. Comment les agents d'immigration du pays étranger savent-ils que votre passeport contient des informations valides?

Si vous remplissez vous-même toutes les informations et que vous les signez, chaque agent d'immigration de chaque pays que vous souhaitez visiter devra vous connaître personnellement et être en mesure d'attester que les informations y sont effectivement correctes.

Une autre façon de gérer cela consiste à envoyer toutes vos informations à un **tiers de confiance (TTP)** . Le TTP ferait une enquête approfondie sur les informations que vous avez fournies, vérifierait vos réclamations, puis signerait votre passeport. Cela s'avère beaucoup plus pratique car les agents d'immigration n'ont besoin de connaître que les tiers de confiance.

Le scénario TTP est la manière dont les certificats sont traités dans la pratique. Le processus ressemble à ceci:

1. **Créer une demande de signature de certificat (CSR):** C'est comme remplir les informations de votre visa.
2. **Envoyer le CSR à un tiers de confiance (TTP):** C'est comme envoyer vos informations à un bureau de demande de visa.

3. **Vérifiez vos informations:** D'une manière ou d'une autre, le TTP doit vérifier les informations que vous avez fournies. À titre d'exemple, voyez comment Amazon valide la propriété .
4. **Générer une clé publique:** le TTP signe votre CSR. Cela équivaut à la signature de votre visa par le TTP.
5. **Émettez la clé publique vérifiée:** cela équivaut à recevoir votre visa par la poste.

Notez que le CSR est lié de manière cryptographique à votre clé privée. Ainsi, les trois informations (clé publique, clé privée et autorité de certification) sont liées d'une manière ou d'une autre. Cela crée ce que l'on appelle une chaîne de confiance , de sorte que vous disposez désormais d'un certificat valide qui peut être utilisé pour vérifier votre identité.

Le plus souvent, cela relève uniquement de la responsabilité des propriétaires de sites Web. Un propriétaire de site Web suivra toutes ces étapes. À la fin de ce processus, leur certificat dit ce qui suit:

> De temps A au temps B Je suis X selon Y

Cette phrase est tout ce qu'un certificat vous dit vraiment. Les variables peuvent être renseignées comme suit:

- **A** est la date et l'heure de début valides.
- **B** est la date et l'heure de fin valides.
- **X** est le nom du serveur.
- **Y** est le nom de l'autorité de certification.

Fondamentalement, c'est tout ce qu'un certificat décrit. En d'autres termes, avoir un certificat ne signifie pas nécessairement que vous êtes qui vous dites être, juste que vous avez Yde *convenir* que vous êtes qui vous dites vous êtes. C'est là qu'intervient la partie «de confiance» des tiers de confiance.

Les TTP doivent être partagés entre les clients et les serveurs pour que tout le monde soit satisfait de la poignée de main HTTPS. Votre navigateur est livré avec de nombreuses autorités de certification installées automatiquement. Pour les voir, procédez comme suit:

- **Chrome:** accédez à *Paramètres> Avancé> Confidentialité et sécurité> Gérer les certificats> Autorités* .
- **Firefox:** accédez à *Paramètres> Préférences> Confidentialité et sécurité> Afficher les certificats> Autorités* .

Cela couvre l'infrastructure requise pour créer des applications Python HTTPS dans le monde réel. Dans la section suivante, vous appliquerez ces concepts à votre propre code. Vous découvrirez les exemples les plus courants et deviendrez votre propre autorité de certification pour les écureuils secrets!

À quoi ressemble une application Python HTTPS?

Maintenant que vous avez une compréhension des éléments de base nécessaires pour créer une application Python HTTPS, il est temps de relier tous les éléments un par un à votre application d'avant. Cela garantira que votre communication entre le serveur et le client est sécurisée.

Il est possible de configurer l'ensemble de l'infrastructure PKI sur votre propre machine, et c'est exactement ce que vous allez faire dans cette section. Ce n'est pas aussi difficile que ça en a l'air, alors ne vous inquiétez pas! Devenir une véritable autorité de certification est beaucoup plus difficile que de suivre les étapes ci-dessous, mais ce que vous lirez est plus ou moins tout ce dont vous avez besoin pour exécuter votre propre autorité de certification.

Devenir une autorité de certification

Une autorité de certification n'est rien de plus qu'une paire de clés publique et privée très importante. Pour devenir une autorité de certification, il vous suffit de générer une paire de clés publique et privée.

Remarque: devenir une autorité de certification destinée à être utilisée par le public est un processus très ardu, bien que de nombreuses entreprises aient suivi ce processus. Cependant, vous ne ferez pas partie de ces entreprises à la fin de ce didacticiel!

Votre paire de clés publique et privée initiale sera un certificat auto-signé . Vous générez le secret initial, donc si vous allez réellement devenir une autorité de certification, il est extrêmement important que cette clé privée soit sûre. Si quelqu'un accède à la paire de clés publique et privée de l'autorité de certification, il peut générer un certificat entièrement valide et vous ne pouvez rien faire pour détecter le problème, sauf pour arrêter de faire confiance à votre autorité de certification.

Avec cet avertissement à l'écart, vous pouvez générer le certificat en un rien de temps. Pour commencer, vous devrez générer une clé privée. Collez ce qui suit dans un fichier appelé `pki_helpers.py`:

```
# pki_helpers.py

from cryptography.hazmat.backends import default_backend

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives.asymmetric import rsa


def generate_private_key(filename: str, passphrase: str):

    private_key = rsa.generate_private_key(

        public_exponent=65537, key_size=2048, backend=default_backend()

    )


    utf8_pass = passphrase.encode("utf-8")
```

```
    algorithm = serialization.BestAvailableEncryption(utf8_pass)


    with open(filename, "wb") as keyfile:
        keyfile.write(
            private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.TraditionalOpenSSL,
                encryption_algorithm=algorithm,
            )
        )


    return private_key
```

`generate_private_key()`génère une clé privée à l'aide de RSA . Voici une ventilation du code:

- **Les lignes 2 à 4** importent les bibliothèques nécessaires au fonctionnement de la fonction.
- **Les lignes 7 à 9** utilisent RSA pour générer une clé privée. Les nombres magiques `65537` et `2048`ne sont que deux valeurs possibles. Vous pouvez en savoir plus sur les raisons ou simplement croire que ces chiffres sont utiles.
- **Les lignes 11 à 12** définissent l'algorithme de cryptage à utiliser sur votre clé privée.
- **Les lignes 14 à 21** écrivent votre clé privée sur le disque au `filename`. Ce fichier est crypté à l'aide du mot de passe fourni.

La prochaine étape pour devenir votre propre autorité de certification consiste à générer une clé publique auto-signée. Vous pouvez contourner la demande de signature de certificat (CSR) et créer immédiatement une clé publique. Collez ce qui suit dans `pki_helpers.py`:

```
# pki_helpers.py

from datetime import datetime, timedelta

from cryptography import x509

from cryptography.x509.oid import NameOID

from cryptography.hazmat.primitives import hashes


def generate_public_key(private_key, filename, **kwargs):

    subject = x509.Name(
```

```
    [
        x509.NameAttribute(NameOID.COUNTRY_NAME, kwargs["country"]),
        x509.NameAttribute(
            NameOID.STATE_OR_PROVINCE_NAME, kwargs["state"]
        ),
        x509.NameAttribute(NameOID.LOCALITY_NAME, kwargs["locality"]),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, kwargs["org"]),
        x509.NameAttribute(NameOID.COMMON_NAME, kwargs["hostname"]),
    ]
)

# Because this is self signed, the issuer is always the subject
issuer = subject

# This certificate is valid from now until 30 days
valid_from = datetime.utcnow()
valid_to = valid_from + timedelta(days=30)

# Used to build the certificate
builder = (
    x509.CertificateBuilder()
    .subject_name(subject)
    .issuer_name(issuer)
    .public_key(private_key.public_key())
    .serial_number(x509.random_serial_number())
    .not_valid_before(valid_from)
    .not_valid_after(valid_to)
)
```

```
    # Sign the certificate with the private key

    public_key = builder.sign(

        private_key, hashes.SHA256(), default_backend()

    )



    with open(filename, "wb") as certfile:

        certfile.write(public_key.public_bytes(serialization.Encoding.PEM))



    return public_key
```

Ici vous avez une nouvelle fonction `generate_public_key()` qui générera une clé publique auto-signée. Voici comment ce code fonctionne:

- **Les lignes 2 à 5** sont des importations nécessaires pour que la fonction fonctionne.
- **Les lignes 8 à 18** rassemblent des informations sur le sujet du certificat.
- **La ligne 21** utilise le même émetteur et le même sujet puisqu'il s'agit d'un certificat auto-signé.
- **Les lignes 24 à 25** indiquent la période pendant laquelle cette clé publique est valide. Dans ce cas, c'est 30 jours.
- **Les lignes 28 à 36** ajoutent toutes les informations requises à un objet générateur de clé publique, qui doit ensuite être signé.
- **Les lignes 38 à 41** signent la clé publique avec la clé privée.
- **Les lignes 43 à 44** écrivent la clé publique dans `filename`.

En utilisant ces deux fonctions, vous pouvez générer votre paire de clés privée et publique assez rapidement en Python:

```
>>> from pki_helpers import generate_private_key, generate_public_key
>>> private_key = generate_private_key("ca-private-key.pem", "secret_password")
>>> private_key
<cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey object at
0x7ffbb292bf90>
>>> generate_public_key(
...   private_key,
...   filename="ca-public-key.pem",
...   country="US",
...   state="Maryland",
...   locality="Baltimore",
...   org="My CA Company",
...   hostname="my-ca.com",
... )
<Certificate(subject=<Name(C=US,ST=Maryland,L=Baltimore,O=My CA
Company,CN=logan-ca.com)>, ...)>
```

Après avoir importé vos fonctions d'assistance depuis `pki_helpers`, vous générez d'abord votre clé privée et l'enregistrez dans le fichier `ca-private-key.pem`. Vous passez ensuite cette clé

privée dans `generate_public_key()`pour générer votre clé publique. Dans votre répertoire, vous devriez maintenant avoir deux fichiers:

```
$ ls ca*
ca-private-key.pem ca-public-key.pem
```

Toutes nos félicitations! Vous avez maintenant la possibilité d'être une autorité de certification.

Faire confiance à votre serveur

La première étape pour que votre serveur devienne fiable consiste à générer une **demande de signature de certificat (CSR)** . Dans le monde réel, le CSR serait envoyé à une autorité de certification réelle comme Verisign ou Let's Encrypt . Dans cet exemple, vous utiliserez l'autorité de certification que vous venez de créer.

Collez le code de génération d'un CSR dans le `pki_helpers.py` fichier d'en haut:

```
# pki_helpers.py

def generate_csr(private_key, filename, **kwargs):

    subject = x509.Name(

        [

            x509.NameAttribute(NameOID.COUNTRY_NAME, kwargs["country"]),

            x509.NameAttribute(

                NameOID.STATE_OR_PROVINCE_NAME, kwargs["state"]

            ),

            x509.NameAttribute(NameOID.LOCALITY_NAME, kwargs["locality"]),

            x509.NameAttribute(NameOID.ORGANIZATION_NAME, kwargs["org"]),

            x509.NameAttribute(NameOID.COMMON_NAME, kwargs["hostname"]),

        ]

    )


    # Generate any alternative dns names

    alt_names = []

    for name in kwargs.get("alt_names", []):

        alt_names.append(x509.DNSName(name))

    san = x509.SubjectAlternativeName(alt_names)


    builder = (

        x509.CertificateSigningRequestBuilder()

        .subject_name(subject)

        .add_extension(san, critical=False)
```

```
    )


    csr = builder.sign(private_key, hashes.SHA256(), default_backend())


    with open(filename, "wb") as csrfile:

        csrfile.write(csr.public_bytes(serialization.Encoding.PEM))


    return csr
```

Pour la plupart, ce code est identique à la façon dont vous avez généré votre clé publique d'origine. Les principales différences sont décrites ci-dessous:

- **Les lignes 16 à 19** définissent des noms DNS alternatifs, qui seront valides pour votre certificat.
- **Les lignes 21 à 25** génèrent un objet constructeur différent, mais le même principe fondamental s'applique qu'auparavant. Vous créez tous les attributs requis pour votre CSR.
- **La ligne 27** signe votre CSR avec une clé privée.
- **Les lignes 29 à 30** écrivent votre CSR sur le disque au format PEM.

Vous remarquerez que, pour créer un CSR, vous aurez d'abord besoin d'une clé privée. Heureusement, vous pouvez utiliser le même `generate_private_key()`à partir du moment où vous avez créé la clé privée de votre autorité de certification. En utilisant la fonction ci-dessus et les méthodes précédentes définies, vous pouvez effectuer les opérations suivantes:

```
>>> from pki_helpers import generate_csr, generate_private_key
>>> server_private_key = generate_private_key(
...   "server-private-key.pem", "serverpassword"
... )
>>> server_private_key
<cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey object at
0x7f6adafa3050>
>>> generate_csr(
...   server_private_key,
...   filename="server-csr.pem",
...   country="US",
...   state="Maryland",
...   locality="Baltimore",
...   org="My Company",
...   alt_names=["localhost"],
...   hostname="my-site.com",
... )
<cryptography.hazmat.backends.openssl.x509._CertificateSigningRequest object at
0x7f6ad5372210>
```

Après avoir exécuté ces étapes dans une console, vous devriez vous retrouver avec deux nouveaux fichiers:

1. **`server-private-key.pem`:** la clé privée de votre serveur
2. **`server-csr.pem`:** le CSR de votre serveur

Vous pouvez afficher votre nouvelle CSR et votre clé privée à partir de la console:

```
$ ls server*.pem
server-csr.pem  server-private-key.pem
```

Avec ces deux documents en main, vous pouvez maintenant commencer le processus de signature de vos clés. En règle générale, de nombreuses vérifications se produiraient à cette étape. Dans le monde réel, l'AC s'assurerait que vous possédiez `my-site.com` et vous demander de le prouver de différentes manières.

Puisque vous êtes l'autorité de certification dans ce cas, vous pouvez renoncer à ce mal de tête et créer votre propre clé publique vérifiée. Pour ce faire, vous allez ajouter une autre fonction à votre `pki_helpers.py` déposer:

```
# pki_helpers.py
def sign_csr(csr, ca_public_key, ca_private_key, new_filename):
    valid_from = datetime.utcnow()
    valid_until = valid_from + timedelta(days=30)

    builder = (
        x509.CertificateBuilder()
        .subject_name(csr.subject)
        .issuer_name(ca_public_key.subject)
        .public_key(csr.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(valid_from)
        .not_valid_after(valid_until)
    )

    for extension in csr.extensions:
        builder = builder.add_extension(extension.value, extension.critical)

    public_key = builder.sign(
        private_key=ca_private_key,
        algorithm=hashes.SHA256(),
        backend=default_backend(),
    )

    with open(new_filename, "wb") as keyfile:
        keyfile.write(public_key.public_bytes(serialization.Encoding.PEM))
```

Ce code ressemble beaucoup à `generate_public_key()` du `generate_ca.py`déposer. En fait, ils sont presque identiques. Les principales différences sont les suivantes:

- **Les lignes 8 à 9** basent le nom du sujet sur le CSR, tandis que l'émetteur est basé sur l'autorité de certification.
- **La ligne 10** obtient cette fois la clé publique du CSR.
- **Les lignes 16 à 17** copient toutes les extensions définies sur le CSR.
- **La ligne 20** signe la clé publique avec la clé privée de l'autorité de certification.

L'étape suivante consiste à lancer la console Python et à utiliser `sign_csr()`. Vous devrez charger votre CSR et la clé privée et publique de votre autorité de certification. Commencez par charger votre CSR:

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> csr_file = open("server-csr.pem", "rb")
>>> csr = x509.load_pem_x509_csr(csr_file.read(), default_backend())
```

```
>>> csr
<cryptography.hazmat.backends.openssl.x509._CertificateSigningRequest object at
0x7f68ae289150>
```

Dans cette section de code, vous ouvrez votre `server-csr.pem` fichier et utilisation `x509.load_pem_x509_csr()` pour créer votre `csr`objet. Ensuite, vous devrez charger la clé publique de votre autorité de certification:

```
>>> ca_public_key_file = open("ca-public-key.pem", "rb")
>>> ca_public_key = x509.load_pem_x509_certificate(
...   ca_public_key_file.read(), default_backend()
... )
>>> ca_public_key
<Certificate(subject=<Name(C=US,ST=Maryland,L=Baltimore,O=My CA
Company,CN=logan-ca.com)>, ...)>
```

Encore une fois, vous avez créé un `ca_public_key` objet qui peut être utilisé par `sign_csr()`. Le `x509` le module avait la main `load_pem_x509_certificate()`aider. La dernière étape consiste à charger la clé privée de votre autorité de certification:

```
>>> from getpass import getpass
>>> from cryptography.hazmat.primitives import serialization
>>> ca_private_key_file = open("ca-private-key.pem", "rb")
>>> ca_private_key = serialization.load_pem_private_key(
...   ca_private_key_file.read(),
...   getpass().encode("utf-8"),
...   default_backend(),
... )
Password:
>>> private_key
<cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey object at
0x7f68a85ade50>
```

Ce code chargera votre clé privée. Rappelez-vous précédemment que votre clé privée a été chiffrée à l'aide du mot de passe que vous avez spécifié. Avec ces trois composants, vous pouvez désormais signer votre CSR et générer une clé publique vérifiée:

```
>>> from pki_helpers import sign_csr
>>> sign_csr(csr, ca_public_key, ca_private_key, "server-public-key.pem")
```

Après avoir exécuté cela, vous devriez avoir trois fichiers de clé de serveur dans votre répertoire:

```
$ ls server*.pem
server-csr.pem  server-private-key.pem  server-public-key.pem
```

Ouf! C'était pas mal de travail. La bonne nouvelle est que maintenant que vous disposez de votre paire de clés privée et publique, vous n'avez plus besoin de modifier le code serveur pour commencer à l'utiliser.

En utilisant votre original `server.py` fichier, exécutez la commande suivante pour démarrer votre toute nouvelle application Python HTTPS:

```
$ uwsgi \
    --master \
    --https localhost:5683,\
            logan-site.com-public-key.pem,\
            logan-site.com-private-key.pem \
    --mount /=server:app
```

Toutes nos félicitations! Vous avez maintenant un serveur compatible Python HTTPS fonctionnant avec votre propre paire de clés privée-publique, qui a été signée par votre propre autorité de certification!

Remarque: Il y a un autre aspect de l'équation d'authentification Python HTTPS, et c'est le **client** . Il est également possible de configurer la vérification de certificat pour un certificat client. Cela nécessite un peu plus de travail et n'est pas vraiment vu en dehors des entreprises. Cependant, l'authentification client peut être un outil très puissant.

Il ne vous reste plus qu'à interroger votre serveur. Tout d'abord, vous devrez apporter des modifications au `client.py` code:

```
# client.py
import os
import requests

def get_secret_message():
    response = requests.get("https://localhost:5683")
    print(f"The secret message is {response.text}")

if __name__ == "__main__":
    get_secret_message()
```

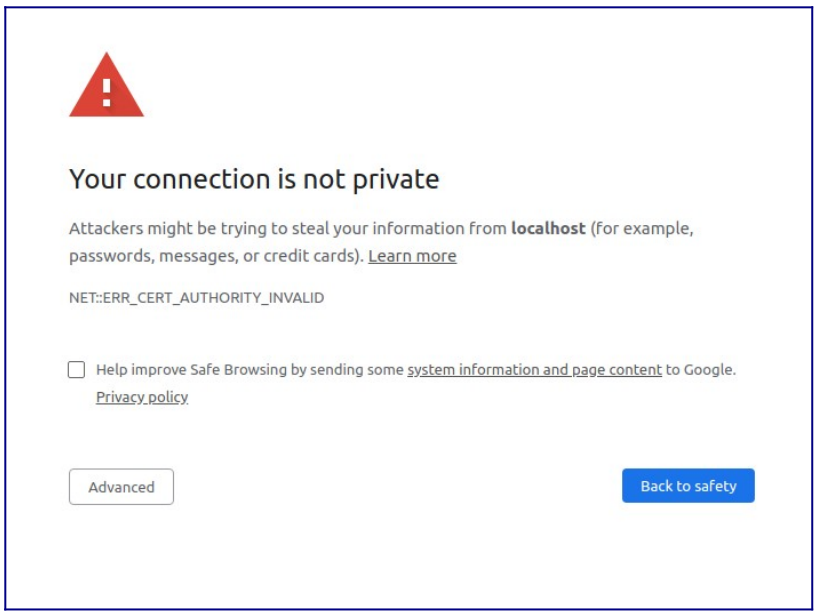
Le seul changement par rapport au code précédent est de `http` à `https`. Si vous essayez d'exécuter ce code, vous rencontrerez une erreur:

```
$ python client.py
...
requests.exceptions.SSLError: \
    HTTPSConnectionPool(host='localhost', port=5683): \
    Max retries exceeded with url: / (Caused by \
    SSLError(SSLCertVerificationError(1, \
    '[SSL: CERTIFICATE_VERIFY_FAILED] \
    certificate verify failed: unable to get local issuer \
    certificate (_ssl.c:1076)')))
```

C'est tout un message d'erreur désagréable! La partie importante ici est le message `certificate verify failed: unable to get local issuer`. Ces mots devraient vous être plus familiers maintenant. Essentiellement, il dit ce qui suit:

> localhost: 5683 m'a donné un certificat. J'ai vérifié l'émetteur du certificat qu'il m'a donné, et selon toutes les autorités de certification que je connais, cet émetteur n'en fait pas partie.

Si vous essayez de naviguer vers votre site Web avec votre navigateur, vous recevrez un message similaire:



Si vous voulez éviter ce message, vous devez le dire `requests`à propos de votre autorité de certification! Tout ce que vous avez à faire est de faire des demandes au `ca-public-key.pem` fichier que vous avez généré précédemment:

```
# client.py
def get_secret_message():
    response = requests.get("http://localhost:5683", verify="ca-public-key.pem")
    print(f"The secret message is {response.text}")
```

Après cela, vous devriez être en mesure d'exécuter ce qui suit avec succès:

```
$ python client.py
The secret message is fluffy tail
```

Joli! Vous avez créé un serveur Python HTTPS entièrement fonctionnel et l'avez interrogé avec succès. Vous et les écureuils secrets avez maintenant des messages que vous pouvez échanger avec bonheur et en toute sécurité!

Conclusion

Dans ce didacticiel, vous avez appris certains des fondements de base des **communications sécurisées** sur Internet aujourd'hui. Maintenant que vous comprenez ces éléments de base, vous deviendrez un développeur meilleur et plus sûr.

Tout au long de ce didacticiel, vous avez acquis une compréhension de plusieurs sujets:

- Cryptographie
- HTTPS et TLS
- Infrastructure à clé publique
- Certificats

Si cette information vous intéresse, alors vous avez de la chance! Vous avez à peine effleuré la surface de toutes les nuances impliquées dans chaque couche. Le monde de la sécurité évolue constamment et de nouvelles techniques et vulnérabilités sont constamment découvertes. Si vous avez encore des questions, n'hésitez pas à nous contacter dans la section commentaires ci-dessous ou sur Twitter .