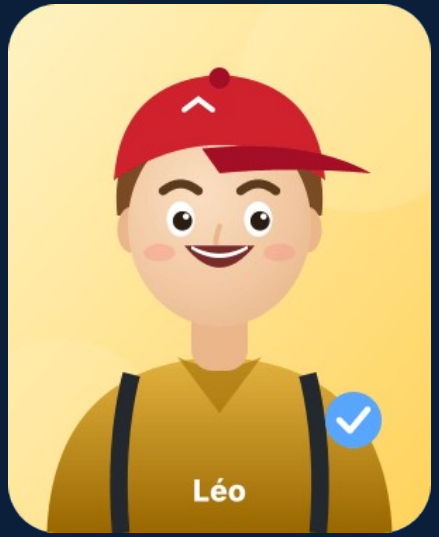
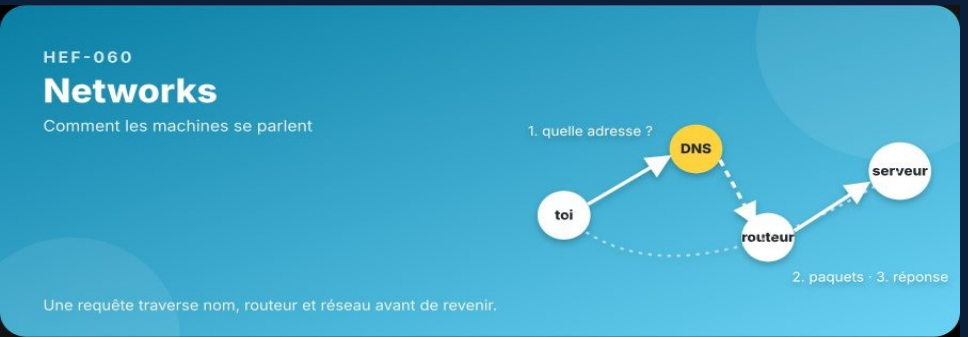
HEF-060 · MISSION 08 / 10

TLS et HTTPS : la confiance

Sans TLS, tout ce que tu envoies voyage en clair. Avec TLS, personne d'autre ne peut ni lire ni modifier.



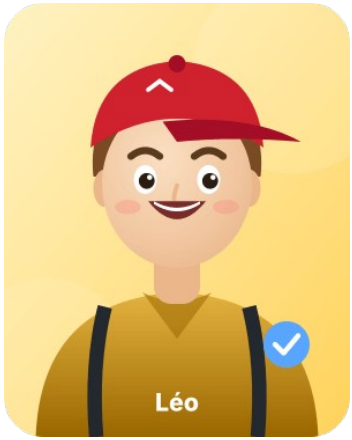


Badge : Gardien du cadenas

85 min · intermédiaire

L'histoire

Dans un café, Kodi se connecte au Wi-Fi public et ouvre le site de HashBox. Sam, à côté, lui montre une capture de ce qui circule : avec l'ancien site en HTTP, on lit les mots de passe en clair. Avec HTTPS, on ne voit qu'un flot illisible.



> *« Le petit cadenas du navigateur, dit Sam, représente trois promesses : confidentialité, intégrité, authenticité. Voyons comment elles sont tenues. »*

Tu comprends les idées derrière HTTPS, sans devenir cryptographe.

Ta mission

À la fin, je serai capable de…

1. énoncer les trois garanties de TLS (confidentialité, intégrité, authentification)
2. distinguer chiffrement symétrique et asymétrique
3. expliquer le rôle des certificats et des autorités de certification
4. décrire les grandes étapes de la poignée de main TLS
5. utiliser hachage et HMAC pour vérifier l'intégrité en Python

Défi d'échauffement

Défi d'échauffement

Tu envoies une carte postale : le facteur, les voisins et n'importe qui peuvent la lire (**confidentialité**), et même la modifier (**intégrité**). Tu ne peux pas non plus être sûr·e que la carte vient bien de ta grand-mère (**authenticité**). Comment ferais-tu pour garantir chacun de ces trois points ? Imagine ce que tu mettrais dans une enveloppe, un sceau, une signature.

Le concept

Trois garanties

Garantie	Question	Moyen
Confidentialité	Quelqu'un d'autre peut-il lire ?	chiffrement
Intégrité	Quelqu'un a-t-il modifié le message ?	empreinte, code d'authentification (MAC)
Authenticité	Suis-je bien en train de parler au bon serveur ?	certificat, signature

TLS (*Transport Layer Security*) offre les trois au-dessus de TCP. **HTTPS** = HTTP dans TLS.

Chiffrement symétrique

Une **même clé** chiffre et déchiffre. **Rapide**, mais comment partager la clé sans qu'on l'intercepte ?

Le concept

Chiffrement asymétrique

Une **paire de clés** : une **clé publique** (partageable) et une **clé privée** (secrète).

- Ce qui est chiffré avec la clé publique ne se déchiffre qu'avec la privée.
- Ce qui est **signé** avec la clé privée se vérifie avec la publique.

Plus **lent**, mais résout le partage de clé. TLS utilise l'asymétrique **pour s'entendre**, puis le symétrique **pour échanger les données** (le meilleur des deux).

L'échange de clés

Le concept

L'échange de clés (suite)

Les deux parties peuvent construire un **secret partagé** sans jamais le transmettre, grâce à l'échange de clés de **Diffie-Hellman**. L'idée avec des couleurs de peinture : chacun mélange une couleur secrète avec une couleur publique, échange le mélange, puis remélange sa couleur secrète : les deux obtiennent la **même couleur finale**, qu'un espion ne peut pas reconstituer.

Les certificats

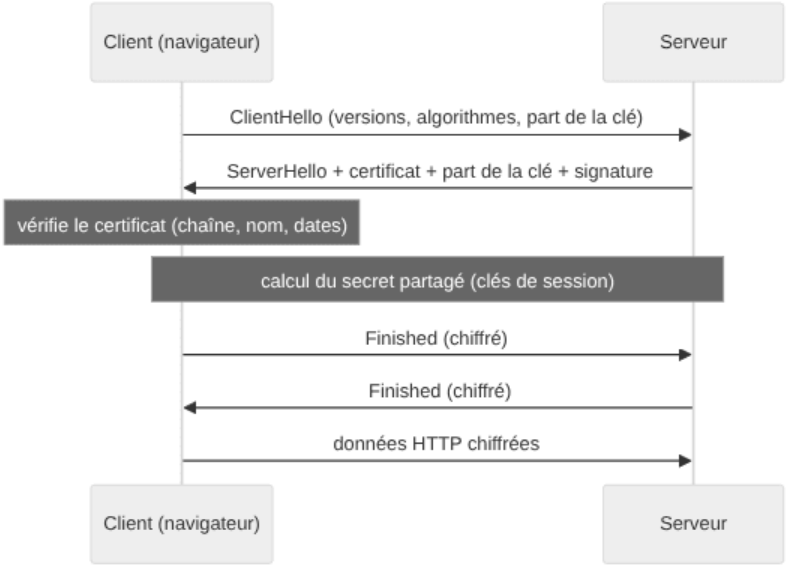
Comment savoir que la clé publique est bien celle de `hashbox.example` ? Grâce à un **certificat** :

- il lie un **nom de domaine** à une **clé publique** ;
- il est **signé** par une **autorité de certification** (AC) de confiance.

Ton système contient la liste des AC de confiance. Le navigateur vérifie la **chaîne** : certificat du site → AC intermédiaire → AC racine de confiance, ainsi que la validité (dates), le nom, et la non-révocation.

Le concept

La poignée de main TLS (version simplifiée) (suite)



Le concept

La poignée de main TLS (version simplifiée) (suite)

Description du schéma : le client propose des paramètres, le serveur répond avec son certificat. Le client vérifie le certificat. Les deux calculent un secret partagé, confirment avec un message « Finished » chiffré, puis échangent les données HTTP chiffrées.

Intégrité en Python

Une **fonction de hachage** transforme un message en empreinte de taille fixe. Un changement d'un seul bit change complètement l'empreinte. Un **HMAC** ajoute une clé secrète : seul un détenteur de la clé peut produire l'empreinte valide.

```
import hashlib, hmac

message = b"virement de 100 euros a Ada"
print(hashlib.sha256(message).hexdigest()[:16])
print(hashlib.sha256(b"virement de 900 euros a Ada").hexdigest()[:16])     # compl
   ètement différent
```

Le concept

Intégrité en Python (suite)

```
cle = b"secret-partage"
etiquette = hmac.new(cle, message, hashlib.sha256).hexdigest()
print(hmac.compare_digest(etiquette, hmac.new(cle, message, hashlib.sha256).hexdig
   est()))   # True
```

Les erreurs de certificat

Un avertissement du navigateur signifie : nom incorrect, certificat expiré, autorité inconnue, chaîne incomplète. **Ne jamais l'ignorer sur un site sensible** : c'est peut-être une attaque de l'homme du milieu.

> 🧠 ***Astuce*** *: TLS protège* ***le transport****, pas ce que fait le site de tes données. Un site en HTTPS peut être malhonnête.*

Exemple pas à pas

Problème : voir pourquoi Diffie-Hellman fonctionne, avec de petits nombres (⚠ simulation pédagogique, jamais à utiliser telle quelle).

```
# Paramètres publics (connus de tous, y compris un espion)
p, g = 23, 5

# Secrets privés
a = 6        # secret d'Alice
b = 15       # secret de Bob

A = pow(g, a, p)          # Alice envoie A
B = pow(g, b, p)          # Bob envoie B

secret_alice = pow(B, a, p)
secret_bob = pow(A, b, p)
print(A, B, secret_alice, secret_bob)       # 8 19 2 2
assert secret_alice == secret_bob
```

Exemple pas à pas (suite)

Lecture : Alice et Bob obtiennent le **même secret (2)** sans qu'il n'ait jamais circulé. Un espion qui voit `p`, `g`, `A`, `B` doit retrouver `a` ou `b` : c'est le **logarithme discret**, difficile avec de grands nombres (des centaines de chiffres) mais facile ici (23).

Vérifier l'intégrité d'un téléchargement :

```
import hashlib

contenu = b"contenu du fichier" * 1000
empreinte_publiee = hashlib.sha256(contenu).hexdigest()

recu = contenu[:-1] + b"X"              # un octet modifié en route
print(hashlib.sha256(recu).hexdigest() == empreinte_publiee)    # False
```

Une API signée avec HMAC :

Exemple pas à pas

Exemple pas à pas (suite)

```
import hashlib, hmac, json

CLE = b"cle-partagee-entre-client-et-serveur"

def signer(corps):
    return hmac.new(CLE, corps, hashlib.sha256).hexdigest()

def verifier(corps, signature):
    return hmac.compare_digest(signer(corps), signature)

corps = json.dumps({"membre": 42, "montant": 100}).encode()
sig = signer(corps)
print(verifier(corps, sig))                                      # True
print(verifier(json.dumps({"membre": 42, "montant": 900}).encode(), sig))   # Fals
   e : modifié
```

À toi, avec aide

Complète le tableau avec la garantie principale apportée et le mécanisme :

Situation	Garantie	Mécanisme
Un espion sur le Wi-Fi ne peut pas lire le mot de passe	?	?
Un attaquant ne peut pas modifier le montant d'un virement en transit	?	?
Le navigateur vérifie qu'il parle bien à la banque	?	?
Le fichier téléchargé n'a pas été altéré	?	?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Lire = confidentialité. Modifier = intégrité. Vérifier l'identité = authenticité.

Indice 2

Les certificats lient un nom à une clé.

Indice 3

Une empreinte (hachage) change si le contenu change.

Réponse

Situation	Garantie	Mécanisme
Espion qui ne lit pas	**Confidentialité**	chiffrement symétrique (clés de session)
Montant non modifiable	**Intégrité**	code d'authentification (HMAC / mode authentifié)
Parler à la vraie banque	**Authenticité**	certificat signé par une AC + vérification de la chaîne
Fichier non altéré	**Intégrité**	empreinte (SHA-256) comparée à la valeur publiée

À toi, en solo

Chaîne de confiance simulée. Simule des certificats avec HMAC en guise de signature (par pédagogie : de vraies signatures utilisent des clés asymétriques) :

- une AC racine `RACINE` avec sa clé secrète ;
- une AC intermédiaire signée par la racine ;
- un certificat de site `hashbox.example` signé par l'intermédiaire, avec dates de validité.

Écris `verifier_chaine(certificat_site, intermediaire, racine_de_confiance, nom_demande, date)` qui contrôle les signatures, les noms, les dates, et la confiance dans la racine. Teste 4 cas : valide, nom incorrect, certificat expiré, signature falsifiée.

À toi, en solo · réponse

Pistes de correction

```
import hashlib, hmac, json

def signer(cle, contenu):
    donnees = json.dumps(contenu, sort_keys=True).encode()
    return hmac.new(cle, donnees, hashlib.sha256).hexdigest()

CLE_RACINE, CLE_INTER = b"cle-racine", b"cle-intermediaire"

def creer_certificat(sujet, emetteur, cle_emetteur, debut, fin):
    contenu = {"sujet": sujet, "emetteur": emetteur, "debut": debut, "fin": fin}
    return {**contenu, "signature": signer(cle_emetteur, contenu)}

def contenu_de(cert):
    return {k: v for k, v in cert.items() if k != "signature"}

def verifier_chaine(site, inter, racine_de_confiance, nom, date):
    if site["sujet"] != nom:
        return "refusé : nom incorrect"
    if not (site["debut"] <= date <= site["fin"]):
```

Pistes de correction (suite)

```
        return "refusé : certificat expiré ou pas encore valide"
    if site["emetteur"] != inter["sujet"] or not hmac.compare_digest(site["signatu
   re"], signer(CLE_INTER, contenu_de(site))):
        return "refusé : signature du site invalide"
    if inter["emetteur"] not in racine_de_confiance or not hmac.compare_digest(int
   er["signature"], signer(CLE_RACINE, contenu_de(inter))):
        return "refusé : intermédiaire non reconnu ou falsifié"
    return "accepté"

inter = creer_certificat("INTER", "RACINE", CLE_RACINE, "2024-01-01", "2034-01-01"
   )
site = creer_certificat("hashbox.example", "INTER", CLE_INTER, "2025-01-01", "2026
   -01-01")
print(verifier_chaine(site, inter, {"RACINE"}, "hashbox.example", "2025-06-01"))
     # accepté
print(verifier_chaine(site, inter, {"RACINE"}, "autre.example", "2025-06-01"))
     # nom incorrect
print(verifier_chaine(site, inter, {"RACINE"}, "hashbox.example", "2027-01-01"))
     # expiré
```

Pistes de correction (suite)

```
faux = dict(site, sujet="hashbox.example", fin="2099-01-01")
      # date modifiée, signature inchangée
print(verifier_chaine(faux, inter, {"RACINE"}, "hashbox.example", "2030-01-01"))
     # signature invalide
```

Dans la vraie vie, la signature est asymétrique (RSA, ECDSA) : n'importe qui peut la **vérifier** avec la clé publique de l'autorité, sans pouvoir la **produire**.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Ignorer un avertissement de certificat	Risque d'attaque de l'homme du milieu	Comprendre la cause avant de continuer
Inventer son propre chiffrement	Presque toujours cassable	Utiliser des bibliothèques éprouvées
Comparer des signatures avec `==`	Attaque par mesure du temps	`hmac.compare_digest`
Croire que HTTPS = site honnête	TLS protège le transport, pas l'intention	Rester vigilant sur le contenu

Mini-boss

Mini-boss

Canal sécurisé miniature. Combine les ingrédients de la mission pour envoyer des messages entre un client et un serveur locaux :

1. Échange de clé **Diffie-Hellman** avec un grand nombre premier public (utilise le groupe de 2048 bits du RFC 3526 si tu veux, ou un premier de 256 bits généré en dur) ;
2. Dérive une clé de session avec `hashlib.sha256(secret)` ;
3. Chaque message est envoyé avec son HMAC ; le récepteur refuse un message modifié ou rejoué (numéro de séquence) ;
4. Teste : message normal, message modifié, message rejoué.

⚠ Écris dans le fichier que c'est un exercice pédagogique et qu'en production on utilise TLS.

Mini-boss · réponse

Pistes de correction

```
import hashlib, hmac, json, secrets

P = 2**255 - 19               # nombre premier de 255 bits (démonstration uniqueme
   nt)
G = 2

class Partie:
    def __init__(self):
        self.prive = secrets.randbelow(P - 2) + 1
        self.public = pow(G, self.prive, P)
        self.cle = None
        self.seq_envoi = 0
        self.seq_recu = -1

    def deriver(self, public_autre):
        secret = pow(public_autre, self.prive, P)
        self.cle = hashlib.sha256(str(secret).encode()).digest()

    def envoyer(self, texte):
```

Mini-boss · réponse

Pistes de correction (suite)

```
        corps = json.dumps({"seq": self.seq_envoi, "texte": texte}).encode()
        self.seq_envoi += 1
        return corps, hmac.new(self.cle, corps, hashlib.sha256).hexdigest()

    def recevoir(self, corps, etiquette):
        attendu = hmac.new(self.cle, corps, hashlib.sha256).hexdigest()
        if not hmac.compare_digest(attendu, etiquette):
            return "refusé : message modifié"
        msg = json.loads(corps)
        if msg["seq"] <= self.seq_recu:
            return "refusé : message rejoué"
        self.seq_recu = msg["seq"]
        return "accepté : " + msg["texte"]

client, serveur = Partie(), Partie()
client.deriver(serveur.public); serveur.deriver(client.public)
assert client.cle == serveur.cle

m1 = client.envoyer("bonjour")
```

Mini-boss · réponse

Pistes de correction (suite)

```
print(serveur.recevoir(*m1))
print(serveur.recevoir(*m1))                                  # rejoué
corps, tag = client.envoyer("virement 100")
print(serveur.recevoir(corps.replace(b"100", b"900"), tag))   # modifié
```

Ce montage protège l'intégrité et le rejeu, mais il manque l'**authentification** des parties (rien ne prouve que le serveur est le vrai serveur) : c'est exactement le rôle des certificats dans TLS.

Quiz éclair

Quiz éclair

1. Une clé publique peut être :

A. gardée secrète
B. utilisée uniquement pour déchiffrer
C. partagée avec tout le monde

2. Un certificat lie :

A. un nom de domaine à une clé publique
B. une adresse IP à une adresse MAC
C. un mot de passe à un utilisateur

3. Pourquoi TLS utilise le chiffrement symétrique pour les données ?

A. il est plus sûr que tout autre

Quiz éclair

Quiz éclair (suite)

B. il est beaucoup plus rapide
C. il n'a pas besoin de clé

4. 🔁 *Révision mission 07* : Une méthode idempotente :

A. est toujours sûre
B. ne renvoie jamais de corps
C. donne le même résultat si on la répète

5. 🔁 *Révision mission 07* : Un code 404 signifie :

A. la ressource demandée est introuvable
B. une erreur du serveur
C. une redirection

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. À quels signes reconnais-tu un site de confiance, et lesquels sont faciles à imiter ?
2. Quel équivalent humain de la « chaîne de confiance » (une personne te présente une autre) connais-tu ?

À retenir

1. TLS : confidentialité, intégrité, authenticité.
2. Asymétrique pour s'entendre, symétrique pour échanger.
3. Un certificat lie un nom à une clé publique, signé par une AC.
4. Hachage et HMAC vérifient l'intégrité.
5. Ne jamais ignorer un avertissement de certificat.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Gardien du cadenas

Prochaine mission : NAT, pare-feu et services

