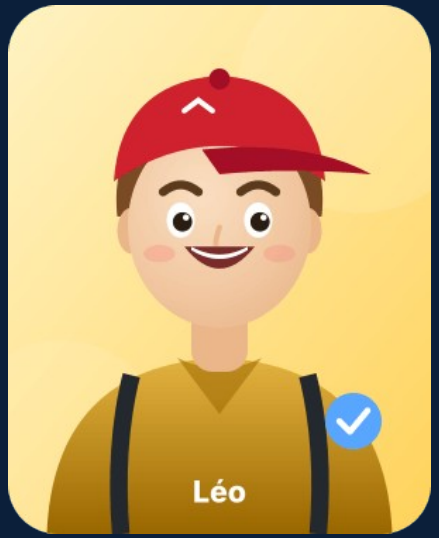
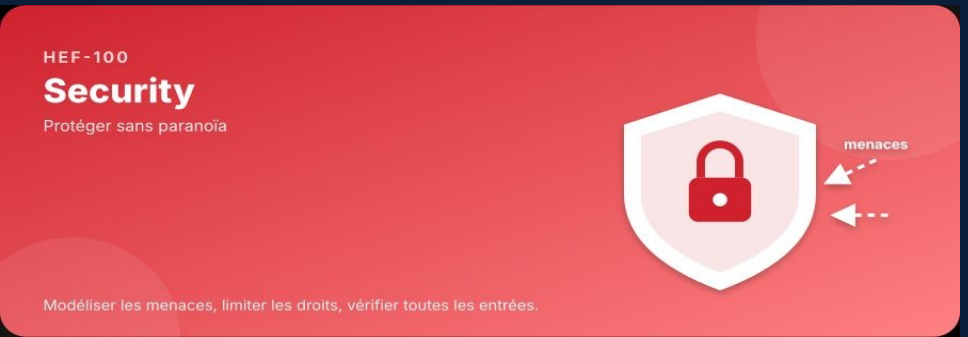
HEF-100 · MISSION 04 / 10

Cryptographie appliquée

Ne réinvente jamais la cryptographie. Apprends à bien l'utiliser, et à reconnaître les erreurs d'usage.





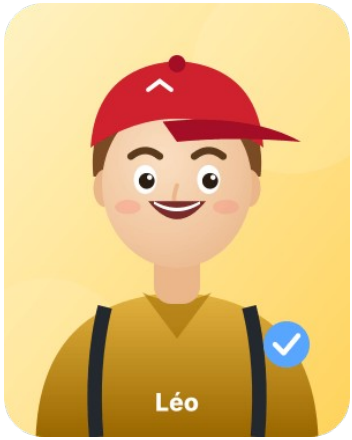
Badge : Utilisateur·rice éclairé·e de crypto

95 min · intermédiaire

L'histoire

Un stagiaire de HashBox a eu une idée : « Pour protéger les numéros de téléphone, je les chiffre avec un XOR et la clé 'hashbox'. » Sam lui montre en deux minutes comment retrouver la clé à partir de quelques données connues.

> *« La cryptographie moderne est fiable, dit Sam, mais seulement si on l'utilise comme prévu. 90 % des failles viennent d'un mauvais usage : mauvais algorithme, mauvais mode, clé mal gérée, aléa insuffisant. »*



Tu apprends les briques (hachage, MAC, chiffrement symétrique et asymétrique, signature, dérivation de clé), leurs usages corrects et leurs pièges.

Ta mission

À la fin, je serai capable de…

1. choisir la bonne primitive : hachage, MAC, KDF, chiffrement, signature
2. expliquer le rôle du sel, du nonce et de l'aléa cryptographique
3. utiliser `hashlib`, `hmac` et `secrets` correctement
4. montrer pourquoi réutiliser un flux de chiffrement est fatal
5. énoncer les règles d'or : pas de cryptographie maison, clés bien gérées, algorithmes standard

Défi d'échauffement

Défi d'échauffement

Tu veux envoyer à une amie un secret dans une enveloppe : (1) elle doit seule pouvoir le lire, (2) elle doit pouvoir vérifier que **personne ne l'a modifié**, (3) elle doit être sûre que **c'est toi** qui l'as envoyé, (4) tu veux pouvoir vérifier plus tard qu'un document est identique à l'original **sans avoir à le garder**. Quels objets (cadenas, sceau de cire, signature, empreinte digitale du document) associes-tu à chaque besoin ?

Le concept

Quelle brique pour quel besoin ?

Besoin	Primitive	Exemples
Empreinte d'un contenu (intégrité sans secret)	**fonction de hachage**	SHA-256, SHA-3, BLAKE2
Intégrité **et** authenticité avec un secret partagé	**MAC** (code d'authentification)	HMAC-SHA256
Stocker un mot de passe, dériver une clé d'un secret faible	**KDF** (dérivation de clé)	argon2id, scrypt, PBKDF2
Confidentialité (avec intégrité)	**chiffrement authentifié** (AEAD)	AES-GCM, ChaCha20-Poly1305
S'entendre sur un secret sans le transmettre	**échange de clés**	Diffie-Hellman, X25519
Prouver l'origine d'un message, sans secret partagé	**signature numérique**	Ed25519, ECDSA, RSA-PSS

Le concept

Hachage

Une **fonction de hachage cryptographique** transforme n'importe quelle donnée en une **empreinte de taille fixe**. Propriétés :

- **déterministe** : même entrée, même empreinte ;
- **à sens unique** : on ne retrouve pas l'entrée à partir de l'empreinte ;
- **résistante aux collisions** : on ne trouve pas deux entrées qui donnent la même empreinte ;
- **effet d'avalanche** : un bit modifié change environ la moitié de l'empreinte.

```
import hashlib

print(hashlib.sha256(b"bonjour").hexdigest()[:16])
print(hashlib.sha256(b"bonjouR").hexdigest()[:16])       # complètement différent
   e
```

Le concept

Hachage (suite)

⚠ **MD5 et SHA-1 sont cassés** pour la résistance aux collisions : ne les utilise plus pour la sécurité.

Sel, coût et mots de passe

Un hachage rapide est une **mauvaise** protection de mot de passe : un attaquant teste des milliards de candidats par seconde. Remèdes :

- le **sel** : une valeur aléatoire par utilisateur, qui rend inutiles les tables précalculées (*rainbow tables*) et empêche de repérer les mots de passe identiques ;
- un **coût** (temps, mémoire) paramétrable qui ralentit chaque tentative.

MAC et HMAC

Le concept

MAC et HMAC (suite)

Un **HMAC** combine un hachage et une **clé secrète** : seul un détenteur de la clé peut produire l'étiquette valide. À utiliser pour authentifier des messages entre services, des webhooks, des cookies signés. Comparaison toujours avec `hmac.compare_digest` (temps constant).

Aléa : `secrets`, pas `random`

`random` (Mersenne Twister) est **prévisible** : ne l'utilise **jamais** pour un jeton, une clé ou un sel. `secrets` (ou `os.urandom`) puise dans le générateur cryptographique du système.

Chiffrement symétrique

Une même **clé** chiffre et déchiffre. Règles :

- utiliser un mode **authentifié** (AES-GCM, ChaCha20-Poly1305), pas AES-ECB (qui laisse deviner les motifs) ni un chiffrement sans intégrité ;

Le concept

Chiffrement symétrique (suite)

- un **nonce** (*number used once*) **unique par message** pour une même clé : le réutiliser peut détruire la sécurité ;
- la clé doit être **aléatoire** (ou dérivée par un KDF) et protégée ;
- en Python, on utilise une bibliothèque réputée (`cryptography`, PyNaCl) : la bibliothèque standard ne propose pas de chiffrement authentifié prêt à l'emploi.

Chiffrement asymétrique et signatures

Une **paire de clés** : publique (diffusable) et privée (secrète). Usages :

- **échange de clés** pour établir un secret partagé (c'est ce que fait TLS) ;
- **signature** : on signe avec la clé privée, n'importe qui vérifie avec la clé publique ;
- **chiffrement hybride** : on chiffre les données avec une clé symétrique, que l'on protège avec la clé publique.

Le concept

Les règles d'or

1. **Ne jamais inventer** un algorithme ou un protocole.
2. **Utiliser des bibliothèques éprouvées** et leurs valeurs par défaut sûres.
3. **La sécurité repose sur la clé**, pas sur le secret de l'algorithme.
4. **Gérer les clés** : génération, stockage (coffre), rotation, révocation, sauvegarde.
5. **Prévoir l'agilité** : pouvoir changer d'algorithme (versionner les formats).
6. **Penser aux métadonnées** : le chiffrement ne cache pas la taille, l'heure ni les destinataires.

Une démonstration d'erreur : réutiliser un flux

Le concept

Une démonstration d'erreur : réutiliser un flux (suite)

Un chiffrement « à flux » fait un XOR entre le message et une suite pseudo-aléatoire (le *keystream*). Si on réutilise le même flux pour deux messages, l'attaquant fait **XOR des deux chiffrés** et obtient le **XOR des deux messages**, qui révèle presque tout.

```
import secrets

def xor(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

flux = secrets.token_bytes(32)
m1 = b"virement de 100 euros a Ada....."
m2 = b"virement de 900 euros a Bob....."
c1, c2 = xor(m1, flux), xor(m2, flux)
print(xor(c1, c2) == xor(m1, m2))        # True : le flux disparaît, il reste m1 X
   OR m2
```

C'est exactement la faille que l'on évite avec un **nonce unique par message**.

Exemple pas à pas

Problème : utiliser correctement hachage, sel, KDF, HMAC et aléa, et constater ce que les erreurs coûtent.

```
import hashlib, hmac, os, secrets, time

# 1. Avalanche : un caractère change, l'empreinte change complètement
a, b = hashlib.sha256(b"virement 100").hexdigest(), hashlib.sha256(b"virement 900"
   ).hexdigest()
differences = sum(x != y for x, y in zip(bin(int(a, 16))[2:].zfill(256), bin(int(b
   , 16))[2:].zfill(256)))
print("bits différents :", differences, "sur 256")

# 2. Sans sel : deux utilisateurs avec le même mot de passe ont la même empreinte
   (et une table précalculée les trouve)
sans_sel = lambda mdp: hashlib.sha256(mdp.encode()).hexdigest()
table = {sans_sel(m): m for m in ["123456", "password", "azerty", "soleil", "motde
   passe"]}   # table précalculée
print("retrouvé sans sel :", table.get(sans_sel("azerty")))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# 3. Avec sel et KDF lent : les empreintes diffèrent, et chaque essai coûte du tem
   ps
def kdf(mdp, sel, iterations):
    return hashlib.pbkdf2_hmac("sha256", mdp.encode(), sel, iterations)

s1, s2 = os.urandom(16), os.urandom(16)
assert kdf("azerty", s1, 1000) != kdf("azerty", s2, 1000)
for iterations in (1_000, 100_000, 600_000):
    t = time.perf_counter()
    kdf("azerty", s1, iterations)
    print(f"{iterations:>8} itérations : {1000 * (time.perf_counter() - t):7.1f} m
   s par essai")

# 4. HMAC : authentifier un message entre deux services
cle = secrets.token_bytes(32)
def signer(message): return hmac.new(cle, message, hashlib.sha256).hexdigest()
def verifier(message, etiquette): return hmac.compare_digest(signer(message), etiq
   uette)
```

Exemple pas à pas (suite)

```
msg = b'{"commande": 42, "montant": 100}'
etiquette = signer(msg)
assert verifier(msg, etiquette)
assert not verifier(b'{"commande": 42, "montant": 900}', etiquette)          # alt
   ération détectée

# 5. Aléa : jeton imprévisible
jeton = secrets.token_urlsafe(32)
print(len(jeton), "caractères,", 32 * 8, "bits d'aléa")
```

Lecture :

- **avalanche** : environ la moitié des 256 bits changent pour une modification minime ;
- **sans sel**, une simple table de correspondance retrouve les mots de passe courants ; avec un sel aléatoire par utilisateur, elle est inutile ;

Exemple pas à pas (suite)

- **le coût** : multiplier les itérations multiplie le temps de chaque essai de l'attaquant, sans gêner un utilisateur légitime qui se connecte une fois ;
- **HMAC** : toute modification du message invalide l'étiquette ;
- **aléa** : `secrets` fournit un jeton imprévisible.

Deux erreurs à reconnaître :

```
import random, secrets

# ❌ jeton prévisible : le générateur de random est déterministe à partir de sa gra
   ine
random.seed(1234)
faux_jeton = "".join(random.choice("abcdef0123456789") for _ in range(16))
random.seed(1234)
assert faux_jeton == "".join(random.choice("abcdef0123456789") for _ in range(16))
       # reproductible → prévisible !
```

Exemple pas à pas (suite)

```
# ✅ jeton cryptographique
vrai_jeton = secrets.token_hex(16)
```

À toi, avec aide

Choisis la bonne primitive (hachage, HMAC, KDF, chiffrement authentifié, signature numérique) pour chaque cas et justifie en une phrase :

1. Vérifier qu'un fichier téléchargé n'a pas été altéré (l'empreinte officielle est publiée sur le site).
2. Authentifier les notifications qu'un fournisseur de paiement envoie à ton serveur (vous partagez un secret).
3. Stocker les mots de passe des utilisateurs.
4. Stocker des numéros de téléphone de façon à pouvoir les relire.
5. Permettre à n'importe qui de vérifier qu'une mise à jour logicielle vient bien de ton équipe.
6. Dériver une clé de chiffrement à partir d'une phrase secrète saisie par l'utilisateur.

À toi, avec aide · réponse

Indices progressifs

Indice 1

« Relire » exige de pouvoir déchiffrer : ce n'est donc pas un hachage.

Indice 2

« N'importe qui peut vérifier » sans secret partagé = signature.

Indice 3

Pour une phrase secrète saisie par un humain, il faut une fonction lente.

Réponse

1. **Hachage** (SHA-256) : on compare l'empreinte calculée à l'empreinte publiée (sur un canal fiable).
2. **HMAC** avec le secret partagé (plus horodatage / identifiant d'événement contre le rejeu).
3. **KDF lente et salée** (argon2id, scrypt, bcrypt) : aucune possibilité de les relire, seulement de les vérifier.
4. **Chiffrement authentifié** (AES-GCM / ChaCha20-Poly1305) avec une clé gérée dans un coffre et un nonce unique par valeur.
5. **Signature numérique** (Ed25519) : la clé privée de signature reste chez toi, la clé publique est distribuée avec le logiciel.
6. **KDF** (scrypt/argon2id/PBKDF2 avec grand nombre d'itérations) avec un sel, pour obtenir une clé de la taille voulue.

À toi, en solo

Démontre la fragilité d'un « chiffrement maison ». Voici le « chiffrement » du stagiaire : XOR avec une clé courte répétée.

```
def chiffrer(message, cle):
    return bytes(m ^ cle[i % len(cle)] for i, m in enumerate(message))
```

1. Chiffre un texte français de 200 caractères avec la clé `b"hashbox"`.
2. Écris une attaque qui retrouve la clé **sans la connaître** en supposant que le texte clair commence par `"Bonjour"` (attaque à clair connu), puis une attaque statistique qui estime la longueur de la clé (indice de coïncidence ou répétitions) et trouve chaque octet par fréquence (l'espace est le caractère le plus fréquent).
3. Conclus : quelles propriétés manquent (confusion, diffusion, clé aussi longue que le message, nonce, intégrité) ?

À toi, en solo · réponse

Pistes de correction

```
def chiffrer(message, cle):
    return bytes(m ^ cle[i % len(cle)] for i, m in enumerate(message))

texte = ("Bonjour a toutes et a tous, bienvenue dans la formation de securite de H
   ashBox. Aujourd'hui nous allons voir "
         "comment proteger les comptes des utilisateurs et pourquoi il ne faut jam
   ais inventer son propre chiffrement. "
         "Merci de votre attention et bon courage pour la suite du programme.").en
   code()
cle = b"hashbox"
c = chiffrer(texte, cle)

# Attaque à clair connu : XOR du début du chiffré avec le début supposé du clair d
   onne le début de la clé
connu = b"Bonjour"
fragment = bytes(a ^ b for a, b in zip(c, connu))
print("clé retrouvée :", fragment)                          # b'hashbox'
assert fragment == cle
```

À toi, en solo · réponse

Pistes de correction (suite)

```
# Attaque statistique sans clair connu : devine la longueur, puis chaque octet par
    la fréquence de l'espace
def indice_coincidence(octets):
    n = len(octets)
    from collections import Counter
    return sum(v * (v - 1) for v in Counter(octets).values()) / (n * (n - 1))

meilleure = max(range(2, 15), key=lambda L: sum(indice_coincidence(c[i::L]) for i
   in range(L)) / L - 0.001 * L)
print("longueur estimée :", meilleure)
cle_estimee = bytes(max(range(256), key=lambda k: sum((b ^ k) == 32 for b in c[i::
   meilleure])) for i in range(meilleure))
print("clé estimée :", cle_estimee)
```

Pistes de correction (suite)

Conclusion : un XOR avec une clé courte répétée n'a ni **diffusion** (chaque octet du clair n'influence qu'un octet du chiffré), ni **confusion** suffisante, ni clé assez longue, ni intégrité (on peut modifier le chiffré). Un chiffrement moderne authentifié (AES-GCM, ChaCha20-Poly1305) apporte tout cela, avec un nonce unique.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Chiffrement maison	Cassé en quelques minutes	Bibliothèques standard éprouvées
`random` pour des jetons	Prévisible	`secrets` ou `os.urandom`
MD5 / SHA-1 pour la sécurité	Collisions pratiques	SHA-256/SHA-3/BLAKE2
Réutiliser un nonce	Fuite du contenu ou de la clé	Un nonce unique par message

Mini-boss

Format de données chiffrées versionné. Conçois et teste un petit module `coffre.py` pour stocker des valeurs sensibles (ici avec les primitives disponibles dans la bibliothèque standard, à titre pédagogique) :

1. dérivation d'une clé depuis une phrase secrète avec `scrypt` et un sel stocké dans l'en-tête ;
2. **authentification** de l'en-tête et des données avec HMAC-SHA256 (encrypt-then-MAC, clés distinctes pour le chiffrement et le MAC dérivées de la clé principale) ;
3. un **flux** pseudo-aléatoire dérivé de HMAC-SHA256 en mode compteur avec un nonce unique par valeur (pédagogique : **en production on utilise AES-GCM ou ChaCha20-Poly1305 d'une bibliothèque**) ;
4. un en-tête versionné `{version, sel, nonce}` pour permettre le changement d'algorithme ;
5. des tests : aller-retour, mauvais mot de passe, donnée modifiée refusée, deux chiffrements du même clair différents (nonce), refus d'une version inconnue ;
6. un texte de 10 lignes : pourquoi cette construction reste inférieure à une bibliothèque éprouvée.

Mini-boss · réponse

Pistes de correction

Un bon module : `deriver(phrase, sel)` retourne 64 octets avec `hashlib.scrypt` ; les 32 premiers servent au chiffrement, les 32 suivants au MAC ; le flux = concaténation de `hmac(cle_chiffrement, nonce + compteur)`; `chiffrer` renvoie `version || sel || nonce || donnees_chiffrees || etiquette`, avec `etiquette = HMAC(cle_mac, en-tête + donnees_chiffrees)`; `dechiffrer` **vérifie l'étiquette avant** de déchiffrer (pour éviter les attaques sur le déchiffrement) et utilise `hmac.compare_digest`. Les tests incluent la modification d'un seul octet de n'importe quelle zone. La conclusion rappelle que la construction est correcte pédagogiquement mais qu'elle n'a pas bénéficié de la revue de la communauté, des optimisations matérielles ni de la résistance aux attaques par canaux auxiliaires : on utilise une bibliothèque.

Quiz éclair

Quiz éclair

1. Un sel (salt) sert à :

A. empêcher les tables précalculées et distinguer les empreintes identiques
B. chiffrer le mot de passe
C. accélérer le hachage

2. Pour authentifier un message avec un secret partagé, on utilise :

A. un simple hachage
B. un HMAC
C. `random`

3. Pour générer un jeton imprévisible en Python :

A. `random`

Quiz éclair

Quiz éclair (suite)

B. `time`
C. `secrets`

4. 🔁 *Révision mission 03* : Un défaut IDOR est :

A. un contrôle d'accès manquant sur l'objet demandé
B. une faille de chiffrement
C. un mauvais mot de passe

5. 🔁 *Révision mission 03* : Pour stocker un mot de passe, on utilise :

A. un chiffrement réversible
B. une fonction de dérivation lente avec un sel
C. un hachage rapide seul

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans quels domaines de ta vie fais-tu confiance à un expert plutôt que d'inventer ta propre solution ? Pourquoi est-ce sage en cryptographie ?
2. Quelle clé ou quel secret de ton environnement serait le plus catastrophique s'il fuyait ?

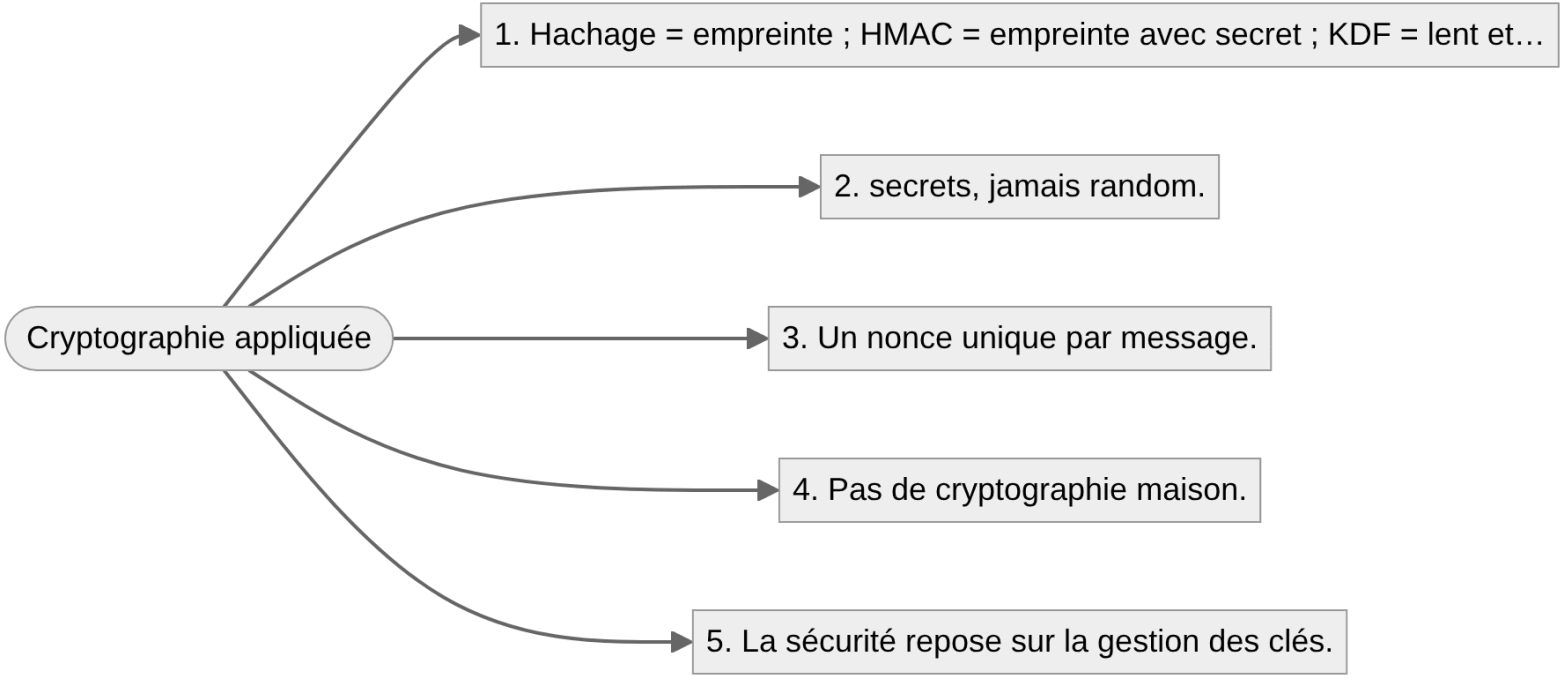
À retenir

À retenir

1. Hachage = empreinte ; HMAC = empreinte avec secret ; KDF = lent et salé ; AEAD = confidentialité + intégrité ; signature = origine.
2. `secrets`, jamais `random`.
3. Un nonce unique par message.
4. Pas de cryptographie maison.
5. La sécurité repose sur la gestion des clés.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Utilisateur·rice éclairé·e de crypto

Prochaine mission : Secrets et données sensibles

