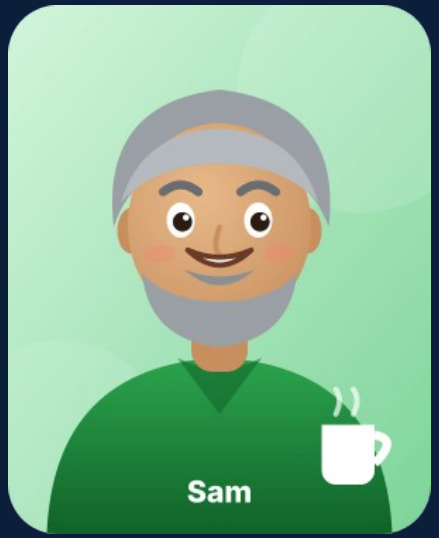
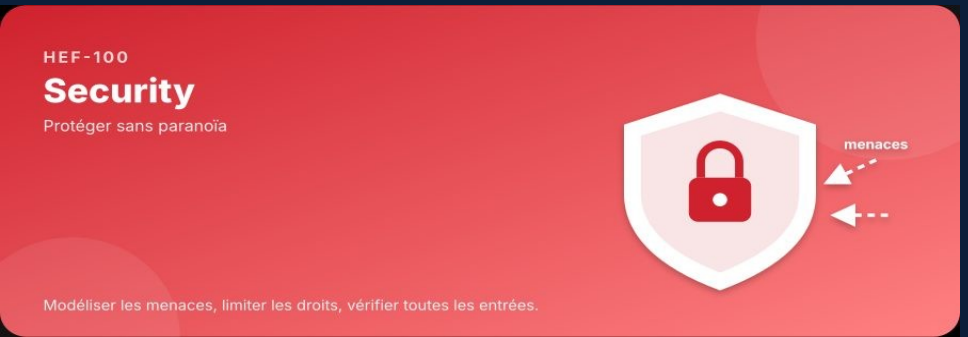
HEF-100 · MISSION 03 / 10

Identité, authentification et autorisation

Qui es-tu ? Peux-tu le prouver ? Et as-tu le droit de faire ça ? Trois questions, trois mécanismes.





Badge : Gardien des accès

100 min · intermédiaire

L'histoire

Un utilisateur de HashBox se plaint : « Quelqu'un a lu mes résultats ! » L'enquête montre que l'application vérifie bien le mot de passe… mais qu'ensuite, pour afficher un profil, elle fait confiance à l'identifiant dans l'URL : `/profil?id=42`. En changeant 42 en 43, on voit le profil de quelqu'un d'autre.

> *« Tu as résolu l'authentification, dit Sam, mais oublié l'autorisation. Savoir qui est là ne dit pas ce qu'il peut faire. »*

Tu apprends à distinguer identification, authentification et autorisation, à stocker des mots de passe correctement, à implémenter le second facteur et les contrôles d'accès (RBAC/ABAC).



Ta mission

À la fin, je serai capable de…

1. distinguer identification, authentification et autorisation
2. stocker et vérifier un mot de passe avec un hachage lent et salé
3. implémenter un mot de passe à usage unique temporel (TOTP) et des jetons de session
4. implémenter RBAC et ABAC et repérer les contrôles d'accès manquants (IDOR)
5. expliquer le flux OAuth 2.0 / OpenID Connect et ses pièges

Défi d'échauffement

Défi d'échauffement

À l'entrée d'un immeuble de bureaux : (1) tu dis ton nom à l'accueil, (2) on vérifie ta pièce d'identité, (3) on te remet un badge qui ouvre seulement certains étages. Quelle étape est l'**identification**, laquelle l'**authentification**, laquelle l'**autorisation** ? Que se passe-t-il si (2) est bien faite mais que le badge ouvre **tous** les étages ? Et si on vérifie les badges seulement au rez-de-chaussée ?

Le concept

Trois notions à ne pas confondre

Notion	Question	Exemple
Identification	qui prétends-tu être ?	saisir un identifiant
Authentification	peux-tu le prouver ?	mot de passe, code à usage unique, clé de sécurité
Autorisation	as-tu le droit de faire cette action sur cette ressource ?	« cet utilisateur peut lire ce cours »

Une erreur classique : bien authentifier, mal autoriser (mission 7 : le contrôle d'accès défaillant est la **première** catégorie de l'OWASP Top 10).

Le concept

Les facteurs d'authentification

Facteur	Principe	Exemples
Connaissance	ce que je sais	mot de passe, code PIN
Possession	ce que j'ai	téléphone (TOTP), clé de sécurité (FIDO2/WebAuthn), carte
Inhérence	ce que je suis	empreinte, visage

L'**authentification multifacteur (MFA)** combine au moins deux facteurs **de familles différentes**. Les clés FIDO2/WebAuthn résistent à l'hameçonnage (elles lient la preuve au site légitime).

Stocker les mots de passe

Jamais en clair, **jamais** chiffrés de façon réversible, **jamais** avec un hachage rapide seul (`md5`, `sha256`). On utilise une **fonction de dérivation de clé lente** : *argon2id*, *scrypt*, *bcrypt* ou *PBKDF2*, avec :

Le concept

Stocker les mots de passe (suite)

- un **sel** (*salt*) aléatoire unique par utilisateur ;
- un **coût** ajustable (temps, mémoire) ;
- une **comparaison en temps constant**.

```
import hashlib, hmac, os

def hacher(mot_de_passe, sel=None):
    sel = sel or os.urandom(16)
    h = hashlib.scrypt(mot_de_passe.encode(), salt=sel, n=2**14, r=8, p=1, dklen=3
   2)
    return sel, h

def verifier(mot_de_passe, sel, h):
    _, candidat = hacher(mot_de_passe, sel)
    return hmac.compare_digest(candidat, h)

sel, h = hacher("correct-cheval-pile-agrafe")
```

Le concept

Stocker les mots de passe (suite)

```
print(verifier("correct-cheval-pile-agrafe", sel, h), verifier("autre", sel, h))
      # True False
```

Le second facteur : TOTP

Le **TOTP** (*Time-based One-Time Password*, RFC 6238) génère un code à 6 chiffres qui change toutes les 30 secondes, à partir d'un **secret partagé** et de l'heure.

```
import hashlib, hmac, struct

def totp(secret, instant, pas=30, chiffres=6):
    compteur = struct.pack(">Q", int(instant) // pas)
    h = hmac.new(secret, compteur, hashlib.sha1).digest()
    decalage = h[-1] & 0x0F
    code = (struct.unpack(">I", h[decalage:decalage + 4])[0] & 0x7FFFFFFF) % (10 *
   * chiffres)
    return str(code).zfill(chiffres)
```

Le concept

Le second facteur : TOTP (suite)

```
# Vecteur de test de la RFC 6238 (secret ASCII de 20 octets, instant 59 s, 8 chiff
   res)
print(totp(b"12345678901234567890", 59, chiffres=8))        # 94287082
```

Sessions et jetons

Après l'authentification, le serveur remet une **preuve de session** que le client représente à chaque requête : cookie de session (identifiant aléatoire stocké côté serveur) ou jeton signé (JWT). Règles :

- identifiant **aléatoire** (`secrets.token_urlsafe(32)`), jamais prévisible ;
- **expiration** et **révocation** possibles ;
- cookie `HttpOnly`, `Secure`, `SameSite` (mission 6) ;
- régénérer l'identifiant après la connexion (contre la fixation de session) ;

Le concept

Sessions et jetons (suite)

- un JWT n'est **pas chiffré** (juste signé) : n'y mets rien de secret, vérifie toujours la signature **et** l'algorithme attendu.

Autorisation : RBAC et ABAC

Modèle	Principe	Exemple
RBAC (*Role-Based*)	les droits sont attachés à des **rôles** ; les utilisateurs ont des rôles	`formateur` peut créer un cours
ABAC (*Attribute-Based*)	une **politique** évalue des **attributs** (utilisateur, ressource, contexte)	un formateur modifie *ses* cours, pendant les heures ouvrées
ACL	liste d'autorisations par ressource	ce dossier est lisible par Ada et Bob

Règles de base : refuser par défaut ; vérifier l'autorisation **côté serveur, à chaque requête, sur chaque objet** ; ne jamais se fier à un identifiant ou à un rôle fourni par le client.

Le concept

Autorisation : RBAC et ABAC (suite)

```
PERMISSIONS = {
    "apprenant": {"cours:lire"},
    "formateur": {"cours:lire", "cours:creer", "cours:modifier_les_siens"},
    "admin": {"cours:lire", "cours:creer", "cours:modifier", "membre:supprimer"},
}

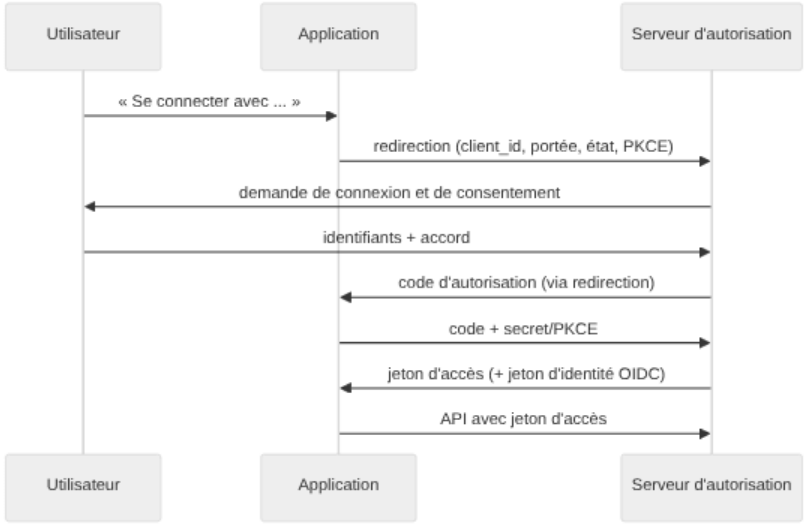
def autorise(utilisateur, action, ressource=None):
    droits = set().union(*(PERMISSIONS.get(r, set()) for r in utilisateur["roles"]
   ))
    if action in droits:
        return True
    if action == "cours:modifier" and "cours:modifier_les_siens" in droits:
    # contrôle d'attribut (ABAC)
        return ressource is not None and ressource["auteur"] == utilisateur["id"]
    return False
```

Le concept

OAuth 2.0 et OpenID Connect

OAuth 2.0 est un protocole d'**autorisation déléguée** : une application obtient un **jeton d'accès** limité pour agir au nom de l'utilisateur, sans connaître son mot de passe. **OpenID Connect (OIDC)** ajoute une couche d'**authentification** (un jeton d'identité).

OAuth 2.0 et OpenID Connect (suite)



Le concept

OAuth 2.0 et OpenID Connect (suite)

Description du schéma : l'application redirige l'utilisateur vers le serveur d'autorisation, qui l'authentifie et recueille son consentement, puis renvoie un code. L'application échange ce code contre des jetons et appelle ensuite les API avec le jeton d'accès.
Pièges classiques : ne pas vérifier le paramètre `state` (CSRF), accepter n'importe quelle URI de redirection, ne pas utiliser PKCE pour les applications publiques, confondre jeton d'accès et preuve d'identité, stocker les jetons en clair.

> 🧠 ***Astuce** : l'authentification se fait **une fois** par session ; l'autorisation se fait **à chaque requête**.*

Exemple pas à pas

Problème : un petit système de connexion avec hachage, TOTP, session et contrôle d'accès, puis la correction d'un défaut d'autorisation (IDOR).

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

# ---------- mots de passe ----------
def hacher(mot_de_passe, sel=None):
    sel = sel or os.urandom(16)
    return sel, hashlib.scrypt(mot_de_passe.encode(), salt=sel, n=2**14, r=8, p=1,
    dklen=32)

def verifier_mdp(mot_de_passe, sel, h):
    return hmac.compare_digest(hacher(mot_de_passe, sel)[1], h)

# ---------- TOTP ----------
def totp(secret, instant, pas=30, chiffres=6):
    compteur = struct.pack(">Q", int(instant) // pas)
    h = hmac.new(secret, compteur, hashlib.sha1).digest()
```

Exemple pas à pas (suite)

```
    decalage = h[-1] & 0x0F
    code = (struct.unpack(">I", h[decalage:decalage + 4])[0] & 0x7FFFFFFF) % (10 *
   * chiffres)
    return str(code).zfill(chiffres)

assert totp(b"12345678901234567890", 59, chiffres=8) == "94287082"            # ve
   cteur de la RFC 6238
assert totp(b"12345678901234567890", 1111111109, chiffres=8) == "07081804"

def verifier_totp(secret, code, instant, tolerance=1):
    return any(hmac.compare_digest(totp(secret, instant + d * 30), code) for d in
   range(-tolerance, tolerance + 1))

# ---------- comptes et sessions ----------
UTILISATEURS = {}
SESSIONS = {}

def inscrire(nom, mot_de_passe, roles):
    sel, h = hacher(mot_de_passe)
```

Exemple pas à pas (suite)

```
    UTILISATEURS[nom] = {"id": len(UTILISATEURS) + 1, "nom": nom, "sel": sel, "has
   h": h,
                        "secret_totp": os.urandom(20), "roles": roles, "echecs":
   0}

def connecter(nom, mot_de_passe, code, maintenant=None):
    maintenant = maintenant or time.time()
    u = UTILISATEURS.get(nom)
    sel = u["sel"] if u else b"\x00" * 16                  # même travail si l'ut
   ilisateur n'existe pas
    attendu = u["hash"] if u else b"\x00" * 32
    ok_mdp = verifier_mdp(mot_de_passe, sel, attendu) and u is not None
    if not ok_mdp or u["echecs"] >= 5:
        if u: u["echecs"] += 1
        return None                                        # message identique da
   ns tous les cas d'échec
    if not verifier_totp(u["secret_totp"], code, maintenant):
        u["echecs"] += 1
        return None
```

Exemple pas à pas (suite)

```
    u["echecs"] = 0
    jeton = secrets.token_urlsafe(32)                        # identifiant de sess
   ion aléatoire
    SESSIONS[jeton] = {"id": u["id"], "expire": maintenant + 1800}
    return jeton

# ---------- autorisation ----------
PERMISSIONS = {"apprenant": {"profil:lire_le_sien"}, "admin": {"profil:lire_tout"}
   }

def utilisateur_de(jeton, maintenant=None):
    s = SESSIONS.get(jeton)
    if not s or s["expire"] < (maintenant if maintenant is not None else time.time
   ()):
        return None
    return next(u for u in UTILISATEURS.values() if u["id"] == s["id"])

def lire_profil(jeton, id_demande, maintenant=None):
    u = utilisateur_de(jeton, maintenant)
```

Exemple pas à pas (suite)

```
    if u is None:
        return 401, "authentification requise"
    droits = set().union(*(PERMISSIONS[r] for r in u["roles"]))
    if "profil:lire_tout" in droits or ("profil:lire_le_sien" in droits and id_dem
   ande == u["id"]):
        return 200, f"profil de l'utilisateur {id_demande}"
    return 403, "accès refusé"                                  # contrôle de l'obje
   t, pas seulement de la session

inscrire("ada", "correct-cheval-pile-agrafe", ["apprenant"])
inscrire("bob", "autre-phrase-de-passe-longue", ["apprenant"])
inscrire("root", "phrase-administrateur-tres-longue", ["admin"])

maintenant = 1_700_000_000
code_ada = totp(UTILISATEURS["ada"]["secret_totp"], maintenant)
jeton = connecter("ada", "correct-cheval-pile-agrafe", code_ada, maintenant)
assert jeton is not None
assert connecter("ada", "mauvais", code_ada, maintenant) is None
assert connecter("ada", "correct-cheval-pile-agrafe", "000000", maintenant) is Non
```

Exemple pas à pas (suite)

```
   e
assert connecter("inconnu", "x", "000000", maintenant) is None
print(lire_profil(jeton, 1, maintenant))       # (200, ...)  son propre profil
print(lire_profil(jeton, 2, maintenant))       # (403, ...)  le profil de Bob : re
   fusé
print(lire_profil("jeton-bidon", 1, maintenant))     # (401, ...)
```

Lecture :

- le mot de passe est haché avec `scrypt` + sel ; la comparaison est en **temps constant** ;
- le second facteur TOTP est vérifié avec une tolérance d'un pas (dérive d'horloge) ;
- un échec renvoie **toujours** `None`, sans dire si c'est le nom, le mot de passe ou le code (pas d'énumération de comptes) ;
- après 5 échecs, le compte refuse de nouvelles tentatives (limitation) ;

Exemple pas à pas (suite)

- `lire_profil` vérifie l'autorisation **sur l'objet demandé** : changer `id=1` en `id=2` n'ouvre rien (c'est la correction du défaut IDOR de l'histoire).

À toi, avec aide

Implémente un contrôle d'accès **ABAC** pour les cours de HashBox :

- un formateur peut modifier un cours **qu'il a créé** ;
- un formateur peut lire tous les cours ;
- un apprenant peut lire un cours **publié** s'il y est inscrit ;
- un admin peut tout faire, **sauf supprimer son propre compte**.

Écris `autorise(utilisateur, action, ressource)` avec ces règles, **refus par défaut**, puis 12 tests (cas autorisés et refusés).

Indices progressifs

Indice 1

Représente un utilisateur par `{id, roles, cours_inscrits}` et un cours par `{id, auteur, publie}`.

Indice 2

Commence par `return False` en fin de fonction : le refus est la valeur par défaut.

Indice 3

Teste explicitement les cas limites : cours non publié, formateur qui modifie le cours d'un autre, admin qui se supprime.

À toi, avec aide · réponse

Réponse

```
def autorise(u, action, r=None):
    roles = set(u["roles"])
    if "admin" in roles:
        if action == "membre:supprimer" and r is not None and r["id"] == u["id"]:
            return False                                        # règle d'exception
        return True
    if action == "cours:lire":
        if "formateur" in roles:
            return True
        if "apprenant" in roles and r is not None:
            return r["publie"] and r["id"] in u["cours_inscrits"]
    if action == "cours:modifier" and "formateur" in roles and r is not None:
        return r["auteur"] == u["id"]
    return False                                                # refus par défaut

ada = {"id": 1, "roles": ["apprenant"], "cours_inscrits": {10}}
sam = {"id": 2, "roles": ["formateur"], "cours_inscrits": set()}
root = {"id": 3, "roles": ["admin"], "cours_inscrits": set()}
c_publie = {"id": 10, "auteur": 2, "publie": True}
```

Réponse (suite)

```
c_brouillon = {"id": 11, "auteur": 2, "publie": False}
c_autre = {"id": 12, "auteur": 9, "publie": True}

assert autorise(ada, "cours:lire", c_publie)
assert not autorise(ada, "cours:lire", c_brouillon)              # non publié
assert not autorise(ada, "cours:lire", c_autre)                  # pas inscrite
assert not autorise(ada, "cours:modifier", c_publie)
assert autorise(sam, "cours:lire", c_brouillon)
assert autorise(sam, "cours:modifier", c_publie)
assert not autorise(sam, "cours:modifier", c_autre)              # cours d'un aut
   re formateur
assert autorise(root, "cours:modifier", c_autre)
assert autorise(root, "membre:supprimer", {"id": 1})
assert not autorise(root, "membre:supprimer", {"id": 3})         # sa propre supp
   ression
assert not autorise(ada, "action:inconnue", c_publie)            # refus par défa
   ut
assert not autorise({"id": 4, "roles": [], "cours_inscrits": set()}, "cours:lire",
    c_publie)
```

À toi, avec aide · réponse

Réponse (suite)

```
print("politique ABAC validée")
```

À toi, en solo

Sessions robustes. Écris un gestionnaire de sessions `GestionnaireSessions` avec :

1. `creer(id_utilisateur, maintenant)` : jeton aléatoire de 256 bits ;
2. expiration **absolue** (8 h) **et** par inactivité (30 min) ;
3. `valider(jeton, maintenant)` qui renvoie l'utilisateur ou `None` et prolonge l'inactivité ;
4. `regenerer(jeton, maintenant)` : nouveau jeton, ancien invalidé (à appeler après connexion pour empêcher la fixation de session) ;
5. `revoquer_toutes(id_utilisateur)` (après changement de mot de passe) ;
6. stockage du **haché** du jeton (SHA-256) et non du jeton lui-même (si la table fuit, les jetons restent inutilisables).

Écris 12 tests, dont l'expiration, la régénération et la révocation.

À toi, en solo · réponse

Pistes de correction

```
import hashlib, secrets

class GestionnaireSessions:
    ABSOLUE, INACTIVITE = 8 * 3600, 30 * 60

    def __init__(self):
        self._sessions = {}                                    # haché du jeton -
   > données

    @staticmethod
    def _h(jeton):
        return hashlib.sha256(jeton.encode()).hexdigest()

    def creer(self, id_utilisateur, maintenant):
        jeton = secrets.token_urlsafe(32)
        self._sessions[self._h(jeton)] = {"id": id_utilisateur, "cree": maintenant
   , "vu": maintenant}
        return jeton
```

Pistes de correction (suite)

```
    def valider(self, jeton, maintenant):
        cle = self._h(jeton)
        s = self._sessions.get(cle)
        if s is None:
            return None
        if maintenant - s["cree"] > self.ABSOLUE or maintenant - s["vu"] > self.IN
ACTIVITE:
            del self._sessions[cle]
            return None
        s["vu"] = maintenant
        return s["id"]

    def regenerer(self, jeton, maintenant):
        id_u = self.valider(jeton, maintenant)
        if id_u is None:
            return None
        del self._sessions[self._h(jeton)]
        return self.creer(id_u, maintenant)
```

Pistes de correction (suite)

```
    def revoquer_toutes(self, id_utilisateur):
        for cle in [c for c, s in self._sessions.items() if s["id"] == id_utilisat
   eur]:
            del self._sessions[cle]

g = GestionnaireSessions()
t0 = 1000
j = g.creer(7, t0)
assert len(j) >= 43 and g.valider(j, t0 + 10) == 7
assert g.valider("n'importe quoi", t0) is None
assert g.valider(j, t0 + 10 + 29 * 60) == 7                          # actif : l'in
   activité est réinitialisée
assert g.valider(j, t0 + 10 + 29 * 60 + 31 * 60) is None             # inactif plus
    de 30 minutes
j2 = g.creer(7, t0)
for k in range(1, 17):                                              # activité régu
   lière toutes les 29 minutes
    g.valider(j2, t0 + k * 29 * 60)
assert g.valider(j2, t0 + 17 * 29 * 60) is None                     # durée absolue
```

Pistes de correction (suite)

```
    de 8 h dépassée
j3 = g.creer(9, t0)
nouveau = g.regenerer(j3, t0 + 5)
assert nouveau != j3 and g.valider(j3, t0 + 6) is None and g.valider(nouveau, t0 +
    6) == 9
a, b = g.creer(5, t0), g.creer(5, t0)
g.revoquer_toutes(5)
assert g.valider(a, t0 + 1) is None and g.valider(b, t0 + 1) is None
assert all(len(k) == 64 for k in g._sessions)                    # on ne stocke
   que des hachés
print("sessions robustes")
```

Stocker le haché du jeton est la même idée que pour les mots de passe : si la base fuit, on ne peut pas rejouer les jetons. Un SHA-256 simple suffit ici car le jeton a 256 bits d'aléa (pas besoin de fonction lente).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Authentifier sans autoriser l'objet	IDOR : `id=42` → `id=43`	Vérifier le droit sur chaque ressource, côté serveur
Hachage rapide pour les mots de passe	Casse en quelques heures sur GPU	scrypt/argon2id/bcrypt avec sel et coût
Messages d'échec différents	Énumération des comptes (« cet utilisateur n'existe pas »)	Même message et même temps pour tous les échecs
Faire confiance au rôle envoyé par le client	Élévation de privilèges triviale	Les rôles viennent du serveur (session)

Mini-boss

Mini-boss

Module d'identité de HashBox. Assemble un module complet testé :

1. inscription avec **politique de mot de passe** (longueur ≥ 12, refus d'une liste de mots de passe courants, vérification de non-contenu du nom d'utilisateur) ;
2. connexion avec hachage `scrypt`, **limitation des tentatives** (verrouillage temporaire croissant) et message d'échec unique ;
3. **MFA TOTP** avec codes de secours à usage unique (hachés) ;
4. **sessions robustes** (mission précédente) et déconnexion partout ;
5. **autorisation** par rôle (RBAC) et par attribut (ABAC) avec refus par défaut ;
6. **journal d'audit** (qui, quoi, quand, résultat) sans jamais écrire de mot de passe ni de jeton ;
7. 30 tests, dont : énumération de comptes impossible (temps et message), IDOR bloqué, rejeu d'un code de secours refusé, fixation de session neutralisée.

Mini-boss · réponse

Pistes de correction

Un bon module : un fichier `identite.py` avec des classes `Comptes`, `Sessions`, `Politique`; `verrouillage` calculé par `min(2 ** (echecs - 3), 900)` secondes après 3 échecs; les codes de secours sont 8 codes de 10 caractères générés avec `secrets`, stockés sous forme de hachés SHA-256 et supprimés à l'usage; le journal d'audit enregistre des événements structurés (`{"evt": "connexion", "utilisateur": id, "ok": False, "ip": "..."}`) en ôtant tout champ sensible; la fonction `autorise` garde la forme « refus par défaut ». Les tests mesurent aussi le temps de réponse pour un utilisateur inconnu et un utilisateur connu avec mauvais mot de passe (même ordre de grandeur) grâce au calcul d'un hachage factice.

Quiz éclair

Quiz éclair

1. L'authentification vérifie :

A. ce que tu as le droit de faire
B. que tu es bien qui tu prétends
C. la vitesse de ta connexion

2. Pour stocker un mot de passe, on utilise :

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

3. Un défaut IDOR est :

A. un contrôle d'accès manquant sur l'objet demandé

Quiz éclair

Quiz éclair (suite)

B. une faille de chiffrement
C. un mauvais mot de passe

4. 🔁 *Révision mission 02* : STRIDE est :

A. un algorithme de chiffrement
B. une méthode de classification des menaces en six catégories
C. un protocole réseau

5. 🔁 *Révision mission 02* : Les menaces se concentrent particulièrement :

A. au centre des bases de données
B. dans les commentaires du code
C. aux frontières de confiance

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Combien de tes comptes importants utilisent un second facteur ? Lesquels devraient en avoir un d'urgence ?
2. Quel contrôle d'accès t'a déjà surpris (un document partagé trop largement, un accès resté après un changement de poste) ?

À retenir

1. Identification, authentification, autorisation : trois choses distinctes.
2. Mots de passe : scrypt/argon2id/bcrypt + sel, comparaison en temps constant.
3. MFA : TOTP, clés FIDO2.
4. Autorisation côté serveur, sur chaque objet, refus par défaut.
5. OAuth 2.0 / OIDC : délégation et identité, avec `state` et PKCE.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Gardien des accès

Prochaine mission : Cryptographie appliquée

