

HACKATON – AI4BMI

SCENARIOS CONCERNÉS

● Scénario 3 (S3) : Fuite de données sensibles

Des hackers exploitent une vulnérabilité dans l'interface d'administration et téléchargent l'intégralité des historiques de maintenance. Ils revendent ces données à des concurrents qui apprennent précisément :

- quelles machines tombent le plus souvent en panne ;
- à quelle fréquence ;
- et quelles pièces de rechange sont utilisées.

● Scénario 5 (S5) : Attaque par inférence

En interrogeant intelligemment l'API, un attaquant déduit des informations confidentielles sur les cadences de production réelles, révélant que BMI fonctionne à 95% de sa capacité, information stratégique pour un concurrent préparant une Offre Publique d'Achat (OPA) hostile.

Livrable 1

OBJECTIFS

Contrôle des accès : S

- Code d'implémentation MFA
- Configuration des politiques de mot de passe
- Captures d'écran de l'interface de connexion
- Schéma d'échange des tokens JWT

REALISATIONS

01 — Code d'implémentation MFA

Outils, scripts et bibliothèques

- **pyotp** (Python) — bibliothèque TOTP/HOTP , interopérable avec Google Authenticator et tout client TOTP
- **qrcode[pil]** (Python) — génère le QR Code PNG encodant l'URI otpauth:// directement en mémoire (base64), sans fichier disque

- **Flask + SQLite — table qr_scans** — stocke un ticket unique (scanne=0/1) par session d'enrôlement pour le polling côté client
- **PyJWT + cryptography** (RSA 2048 bits) — émission et vérification des tokens JWT RS256 signés asymétriquement après validation MFA complète
- **auth.py — connexion_complete()** — orchestre les 3 étapes : check mot de passe → vérif TOTP → émission JWT. Chaque étape bloque si KO
- **app.py — routes /check-credentials, /api/qr-code, /api/qr-confirmed, /login** — séparation claire des étapes en routes distinctes, chacune avec sa propre validation
- **Protocole TOTP RFC 6238** — code à 6 chiffres, fenêtre de validité 30s, valid_window=1 pour tolérer ±30s de décalage réseau industriel

```

import io
import base64
import sqlite3
import secrets as secrets_module

import pyotp
import qrcode

from flask import (
    Flask, request, jsonify,
    make_response, render_template,
    render_template_string
)
from functools import wraps

from detecteur import (
    analyser_requete,
    init_tables_ids,
    enregistrer_alerte
)

from auth import (
    connexion_complete, verifier_jwt,
    renouveler_tokens, journaliser,
    est_bloque, verifier_mot_de_passe,
    enregistrer_tentative, enregistrer_tentative_echouee,
    reinitialiser_tentatives, compter_tentatives_recentes,
    MAX_TENTATIVES
)
from database import initialiser_db, creer_utilisateurs_test, set_must_change
from password_policy import (
    valider_mot_de_passe, sauvegarder_mot_de_passe
)

app = Flask(__name__)

```

Justification des choix

- **TOTP** — le TOTP fonctionne hors ligne, sans dépendance réseau GSM ou SMTP. Le secret ne transite jamais sur le réseau à chaque usage. La norme RFC

6238 garantit l'interopérabilité avec n'importe quelle application d'authentification.

- **QR Code à disparition unique** — , le QR Code est lié à un ticket unique en base (table qr_scans). Une fois scanné, il ne peut plus être affiché. Un attaquant qui photographierait l'écran ne peut pas réutiliser le QR.
- **JWT émis uniquement après les 3 étapes** — la route /login re-vérifie indépendamment le mot de passe ET le code TOTP. Même si l'étape 1 était contournée, l'étape 3 bloque sans code valide.
- **Algorithme RS256 asymétrique** — la clé privée signe, la clé publique vérifie. La clé privée ne quitte jamais le serveur. Contrairement à HS256 symétrique, une compromission du client ne permet pas de forger des tokens.

02 — Politique de mot de passe

Outils, scripts et bibliothèques

-
- **argon2-cffi — Argon2id** — vainqueur Password Hashing Competition 2015, résistant GPU/ASIC grâce à son coût mémoire (memory_cost=65536 Ko, time_cost=2, parallelism=2)
- **verifier_complexite()** — valide les 5 critères obligatoires par regex en une seule passe, retourne un message précis sur le critère manquant
- **verifier_historique()** — stocke SHA-256 des 5 derniers mots de passe en table password_history, empêche la rotation cyclique
- **verifier_liste_noire()** — liste de 9 mots de passe courants (password, azerty, admin123, bmi2026...), comparaison insensible à la casse
- **calculer_force()** — score 0-100 calculé dynamiquement, affiché en temps réel dans login.html via barre de progression
- **gen_mdp_temporaire()** dans add_user.py — génère 12 caractères avec garantie de satisfaire tous les critères via secrets.SystemRandom()
-

```

import re
import hashlib
from database import get_connection

POLITIQUE = {
    "longueur_min": 8,
    "longueur_max": 64,
    "requiert_majuscule": True,
    "requiert_minuscule": True,
    "requiert_chiffre": True,
    "requiert_special": True,
    "caracteres_speciaux": "!@#$%^&*()_+=[]{};:,.<>?",
    "historique_max": 5,
    "expiration_jours": 90,
}

MOTS_DE_PASSE_INTERDITS = [
    "password", "123456", "azerty", "qwerty",
    "admin123", "Bm12026", "motdepasse",
    "Password1!", "Admin123!", "Bm12026!",
]

def verifier_longueur(mot_de_passe):
    if len(mot_de_passe) < POLITIQUE["longueur_min"]:
        return False, f"Minimum {POLITIQUE['longueur_min']} caractères"
    if len(mot_de_passe) > POLITIQUE["longueur_max"]:
        return False, f"Maximum {POLITIQUE['longueur_max']} caractères"
    return True, ""

def verifier_complexite(mot_de_passe):
    erreurs = []
    if POLITIQUE["requiert_majuscule"] and \
            not re.search("[A-Z]", mot_de_passe):
        erreurs.append("une MAJUSCULE requise")
    if POLITIQUE["requiert_minuscule"] and \
            not re.search("[a-z]", mot_de_passe):
        erreurs.append("une minuscule requise")
    if POLITIQUE["requiert_chiffre"] and \
            not re.search("\d", mot_de_passe):
        erreurs.append("un chiffre requis")
    if POLITIQUE["requiert_special"]:
        speciaux = POLITIQUE["caracteres_speciaux"]
        if not any(c in speciaux for c in mot_de_passe):
            erreurs.append("des caractères spéciaux manquent")
    return True, erreurs

```

```

if erreurs:
    return False, " | ".join(erreurs)
return True, ""

def verifier_liste_noire(mot_de_passe):
    for interdit in MOTS_DE_PASSE_INTERDITS:
        if mot_de_passe.lower() == interdit.lower():
            return False, "Mot de passe trop courant"
    return True, ""

def verifier_historique(username, mot_de_passe):
    password_hash = hashlib.sha256(
        mot_de_passe.encode()
    ).hexdigest()

    conn = get_connection()
    cursor = conn.execute("""
        SELECT password_hash FROM password_history
        WHERE username = ?
        ORDER BY created_at DESC
        LIMIT ?
    """, (username, POLITIQUE["historique_max"]))

    historique = [row[0] for row in cursor.fetchall()]
    conn.close()

    if password_hash in historique:
        return False, "Mot de passe déjà utilisé récemment"
    return True, ""

def calculer_force(mot_de_passe):
    score = 0
    if len(mot_de_passe) >= 8: score += 20
    if len(mot_de_passe) >= 12: score += 10
    if len(mot_de_passe) >= 16: score += 10
    if re.search("[A-Z]", mot_de_passe): score += 15
    if re.search("[a-z]", mot_de_passe): score += 15
    if re.search("\d", mot_de_passe): score += 15
    speciaux = POLITIQUE["caracteres_speciaux"]
    if any(c in speciaux for c in mot_de_passe): score += 15

```

Justification des choix

- **Argon2id**— Argon2id combine résistance au temps (time_cost) et résistance mémoire (memory_cost=65536 Ko). Il est inefficace sur GPU et ASIC — les outils de crack modernes ne peuvent pas le paralléliser. Un crack offline après exfiltration de la base est économiquement non viable.
- **Historique des 5 derniers mots de passe** — une politique de rotation sans historique est contournable en changeant et rechargeant rapidement. L'historique force un écart réel entre les mots de passe successifs et empêche le retour aux habitudes.
- **Génération automatique par le système** — si l'admin choisissait les mots de passe temporaires, ils seraient prévisibles (prénom, date de naissance...). La génération aléatoire par secrets.SystemRandom() garantit une entropie maximale dès la création du compte.
- **must_change=1 positionné immédiatement** — même si l'email avec le mot de passe temporaire est intercepté, l'attaquant est bloqué par le décorateur @verifie_mdp avant tout accès aux données. Le vrai mot de passe n'est connu que de l'employé.

Impact

- Crack offline après exfiltration BDD → Argon2id rend le calcul massif $\times 10\ 000$ plus lent qu'un SHA-256

- Mots de passe triviaux conformes aux règles → liste noire + score minimal requis bloquent les cas limites
 - Réutilisation cyclique → historique des 5 derniers bloque la rotation (ex: azerty → motdepasse → azerty)
 - Mot de passe temporaire intercepté → must_change=1 force le changement avant tout accès aux données
 - Admin mal intentionné → ne connaît jamais le vrai mot de passe de l'employé après le premier login
 -
-

03 — Captures d'écran de l'interface de connexion

The screenshot shows a dark-themed login page for 'BMI SEC'. At the top, it says 'Plateforme IA — Maintenance Prédictive'. Below that is a 'Authentification MFA' button. The main area has two steps: 'Identifiants' (step 1) and 'Vérification' (step 2). Step 1 contains fields for 'ADRESSE E-MAIL' (kofi@bmi.bj) and 'MOT DE PASSE' (*****). Step 2 is currently empty. A blue 'Continuer -->' button is at the bottom. At the very bottom, there's small text: 'BMI Usine CDIZ | Clef Digitale Braine | TLU'.

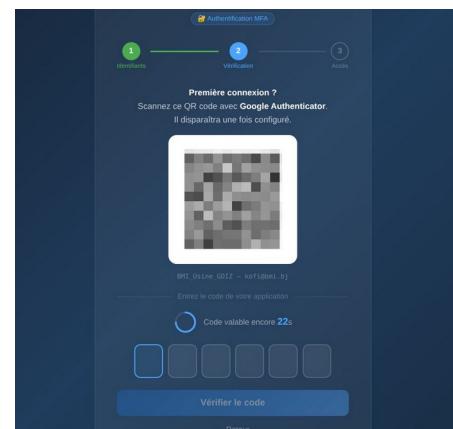
Formulaire email + mot de passe — Étape 1 / 3

L'utilisateur saisit son adresse email et son mot de passe. **La saisie est validée côté serveur** via POST /check-credentials avant de passer à l'étape suivante.

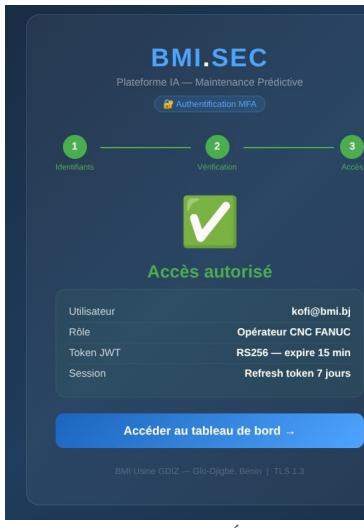
- **Anti brute-force** : 5 tentatives max / 5 min (username + IP). Réponse 429 avec temps restant affiché.
- **IDS niveau 2** : 20 tentatives / 10 min → ban IP automatique 30 min.
- Affichage/masquage du mot de passe via l'icône œil.

Le serveur génère un QR Code PNG encodant le secret TOTP. Le code est affiché avec un **compteur de 30s** indiquant la durée de validité.

- **Première connexion** : scanner avec Google Authenticator pour enrôler le compte.
- **Connexions suivantes** : saisir directement le code 6 chiffres.
- Ticket unique par session : QR inutilisable après confirmation.



QR Code TOTP + compteur 30s — Étape 2 / 3



Carte de session — Étape 3/3

Après validation des 3 facteurs, le serveur émet le JWT et confirme l'accès. La carte de session affiche **toutes les informations de sécurité pertinentes** pour l'utilisateur.

- **Utilisateur** : email confirmé (kofi@bmi.bj)
- **Rôle** : Opérateur CNC FANUC — accès restreint aux 3 machines FANUC
- **Token JWT** : RS256 — expire dans 15 min
- **Session** : Refresh token 7 jours (cookie HttpOnly, SameSite=Strict)

Si must_change = true : cette étape 3 est remplacée par l'écran de changement de mot de passe forcé — l'accès au tableau de bord est bloqué jusqu'à confirmation.

Justification des choix

- **JWT stocké en mémoire JS uniquement** — localStorage est accessible par n'importe quel script injecté (attaque XSS). La mémoire JavaScript est isolée par onglet et effacée à la fermeture.
- **4 sections JS sans recharge** — un recharge de page entre étapes MFA exposerait l'état de progression dans l'historique navigateur. Le flux JS pur reste invisible et non mémorisable.
- **@verifie_mdp côté serveur** — la vérification du must_change est faite côté serveur dans un décorateur, indépendamment du client. Un attaquant modifiant le JWT côté client serait bloqué car la signature RS256 invaliderait le token.

Impact

- Accès par bot ou script → MFA TOTP impossible à automatiser sans le téléphone physique
- Vol de session via XSS → JWT en mémoire JS, inaccessible depuis d'autres onglets ou scripts injectés
- Contournement de l'écran de changement → @verifie_mdp bloque côté serveur, indépendamment du front
- Phishing de session → cookie refresh HttpOnly invisible au JavaScript, non exfiltrable par script

04 — Schéma d'échange des tokens JWT

Outils

- **PyJWT — algorithme RS256** — algorithme asymétrique : clé privée signe, clé publique vérifie. La clé privée ne quitte jamais le serveur
- **cryptography — RSA 2048 bits** — génération de la paire de clés en RAM au démarrage du serveur, jamais écrites sur disque
- **Payload enrichi — champ must_changer** — flag booléen dans le JWT permettant au décorateur `@verifie_mdp` de bloquer les routes sans accès base de données
- **uuid4() — champ jti** — identifiant unique par token pour prévenir les attaques par rejeu
- **Refresh Token rotatif — table refresh_tokens** — à chaque renouvellement, l'ancien token est marqué used=1. Si un token used=1 est présenté → vol détecté → révocation totale
- **Cookie HttpOnly + SameSite=Strict** — le refresh token est stocké dans un cookie inaccessible au JavaScript

Flux complet d'échange

- **1. Login réussi (MFA complet)** — POST /login → vérif mdp + TOTP → créer_jwt() (RS256, 15min) + créer_refresh_token() (UUID, 7j, cookie HttpOnly)
- **2. Requête API protégée** — Authorization: Bearer JWT → @requiert_auth vérifie signature RS256 → @verifie_mdp vérifie must_changer → accès accordé ou 403
- **3. Renouvellement JWT expiré** — POST /refresh (cookie) → vérifie used=0 + non expiré → marque used=1 → nouveau JWT + nouveau refresh token
- **4. Détection de vol** — POST /refresh avec token used=1 → révocation TOTALE de tous les tokens de l'utilisateur → reconnexion MFA forcée

```
④ STRUCTURE JWT RS256 + PAYLOAD must_changer

Format : HEADER.PAYOUT.SIGNATURE (Base64URL)
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJrb2ZpQG...26GyCm2o59MXfsnwpC...

-----  
HEADER  
{  
  "alg": "RS256",  
  "typ": "JWT"  
}  
  
Asymétrique :  
privée → signe  
publique → vérifie  
Clés en RAM  
(régénérées  
démarrage)  
  
PAYLOAD  
{  
  "sub": "kofi@bmi.bj",  
  "role": "operateur_fanuc",  
  "must_changer": true,  
  "iat": <timestamp>,  
  "exp": <iat + 15min>,  
  "jti": "uuid-unique"  
}  
  
must_changer → écran 🚫  
jti → anti-rejeu  
exp 15min → sécurité  
  
SIGNATURE  
RSA_SIGN(  
  SHA256(  
    B64(header)  
    +  
    + B64(payload)  
  ),  
  clé_privee_2048  
)  
  
Infaillible  
sans clé privée  
Test live ci-dessous  
  
Vérification live :  
✓ Token valide - sub=kofi@bmi.bj - must_changer=True  
✓ Token falsifié détecté et rejeté
```

```

→ JWT expire après 15 minutes
→ Client → POST /refresh avec cookie refresh_token
→ Vérifier : token connu ? non expiré ? used = 0 ?
→ Marquer ancien token : used = 1
→ Générer nouveau JWT (15 min) + nouveau refresh (7 jours)
→ Réponse : { access_token } + Set-Cookie nouveau refresh
⚠ Si token already used=1 → DÉTECTION VOL
⚠ DELETE tous les refresh tokens de l'utilisateur
⚠ Forcer reconexion complète MFA

```

États du refresh token (table refresh_tokens)

Scénario	used	expires_at	Action
Token valide	0	futur	Renouveler → nouveau JWT
Token expiré	0	passé	Supprimer → reconnexion
Token déjà utilisé	1	futur	VOL → révoquer tout
Token inconnu	-	-	Rejeter → reconnexion

Justification des choix

- **RS256 plutôt que HS256** — HS256 utilise une clé symétrique : si un service consommateur du token est compromis, il peut créer de faux tokens. RS256 asymétrique élimine ce risque. La clé de vérification (publique) ne permet pas de créer de tokens.
- **Durée JWT 15 min** — une durée courte limite la fenêtre d'exploitation d'un token intercepté. Le refresh token (7 jours, HttpOnly) assure le confort utilisateur sans exposer le JWT longtemps.
- **Clés en RAM uniquement** — une paire de clés RSA régénérée à chaque démarrage invalide automatiquement tous les tokens précédents. Pas de gestion de rotation de clés, pas de fichier de clé à voler sur le système de fichiers.
- **must_change dans le payload** — décision d'autorisation stateless : le serveur n'a pas besoin de requêter la base de données à chaque appel API pour savoir si l'utilisateur doit changer son mot de passe.
- **Rotation du refresh token** — un refresh token à usage unique rend le vol de session détectable automatiquement. Si un attaquant vole et rejoue le token, le serveur détecte la double présentation et révoque tout.

Impact

- Falsification de token → impossible sans la clé privée RSA 2048 bits, jamais exposée
- Rejeu de token intercepté → expiré en 15 min + jti unique par token empêche la réutilisation
- Vol de session longue → détecté par rotation (used=1) → révocation totale automatique
- Contournement du must_change → vérifié côté serveur dans le décorateur, indépendant du client

