VIGIE — construisez votre détecteur d'intrusion

Le contexte

Vous venez d'arriver au SOC d'une PME. Cette nuit, plusieurs choses se sont passées sur le réseau. Personne ne sait quoi : il y a seulement trois fichiers de journaux et 490 lignes à lire à la main.

Vous n'allez pas les lire à la main. Vous allez écrire l'outil qui les lit pour vous, et qui continuera à le faire chaque nuit.

C'est exactement ce que fait le moteur d'un SIEM.

Ce qui est fourni

Fichier	Contenu
`journaux/auth.log`	Connexions SSH au serveur — 181 lignes
`journaux/firewall.log`	Connexions acceptées et rejetées par le pare-feu — 183 lignes
`journaux/web.log`	Requêtes reçues par le serveur web — 126 lignes
`vigie.py`	Le squelette : des fonctions à compléter
`test_vigie.py`	Les tests. **Un exercice est réussi quand ses tests passent.**

Pour démarrer

```
cd vigie
python vigie.py          affiche « 0 alerte(s) » : normal, rien n'est codé
pytest -q                14 échecs : c'est votre feuille de route
```

Aucune bibliothèque à installer. Tout est dans la bibliothèque standard de Python. Si quelqu'un vous propose d'installer quelque chose, c'est qu'il s'est trompé d'exercice.

Au fil des exercices, les tests passent au vert un par un. À la fin, vous devez obtenir **20 passed**.

Exercice 1 — Lire un journal

Notions : ouverture de fichier, boucle sur les lignes, liste.

Complétez `lire(chemin)` : elle renvoie la liste des lignes du fichier, sans les retours à la ligne, en ignorant les lignes vides.

```
lire("journaux/auth.log")      →  181 lignes
lire("journaux/firewall.log")  →  183 lignes
lire("journaux/web.log")       →  126 lignes
```

Vérification : `pytest -k exercice1`

Pièges : oublier `encoding="utf-8"` (les accents casseront plus tard) · compter une dernière ligne vide · ne pas enlever le `\n` final.

Exercice 2 — Extraire une adresse IP

Notions : `split()`, recherche dans une liste, valeur de retour.

Une ligne SSH ressemble à ceci :

```
Oct  3 04:10:02 srv-web01 sshd[2501]: Failed password for invalid user root from 185.203.44.17
port 51234 ssh2
```

L'adresse est **le mot qui suit** `from`. Complétez `ip_apres_from(ligne)`. Si la ligne ne contient pas `from`, renvoyez `None`.

```
ip_apres_from("… from 185.203.44.17 port 51234 ssh2")  →  "185.203.44.17"
ip_apres_from("une ligne sans le mot attendu")         →  None
```

Vérification : `pytest -k exercice2`

Pièges : découper sur un nombre fixe de mots — la position de `from` change d'une ligne à l'autre, il faut la **chercher** · oublier le cas où `from` est absent, et planter sur un `ValueError`.

Exercice 3 — Détecter une attaque par force brute

Notions : dictionnaire de comptage, seuil de détection.

Une force brute, c'est un grand nombre d'échecs d'authentification depuis une même adresse. Complétez `regle_force_brute(lignes_auth)`.

Pour chaque adresse dépassant `SEUIL_ECHECS` (20), produisez une alerte :

```
{"source": "auth.log",
 "severite": "critique",
 "titre": "Attaque par force brute",
 "ip": "185.203.44.17",
 "details": "94 échecs d'authentification depuis cette adresse (seuil : 20).",
 "horodatage": "Oct  3 04:10:02"}
```

L'horodatage, ce sont les **15 premiers caractères** de la première ligne où cette adresse apparaît.

```
Résultat attendu : 1 alerte, pour 185.203.44.17, avec 94 échecs.
```

Attention au bruit : l'adresse `91.207.183.44` a 17 échecs. Elle ne doit **pas** déclencher d'alerte. C'est tout l'intérêt d'un seuil — et c'est ce qui distingue un outil utilisable d'un outil qui crie en permanence.

Vérification : `pytest -k exercice3`

Pièges : compter toutes les lignes au lieu des seuls échecs · créer une alerte par ligne au lieu d'une par adresse · `KeyError` sur la première occurrence d'une adresse (utilisez `collections.Counter`).

Exercice 4 — Détecter une compromission

Notions : état accumulé pendant un parcours, ordre de lecture.

La force brute est une **tentative**. Si elle est suivie d'une connexion réussie depuis la même adresse, c'est une **intrusion**.

Complétez `regle_compromission(lignes_auth)` : parcourez les lignes **dans l'ordre** en comptant les échecs par adresse. Quand vous rencontrez un `Accepted password`, regardez combien d'échecs cette adresse a déjà produits. Au-delà du seuil, c'est une compromission.

Le compte visé est le mot qui suit `for`.

```
Résultat attendu : 1 alerte critique.
                   Le constat doit nommer le compte compromis.
```

Vérification : `pytest -k exercice4`

Pourquoi c'est la règle la plus importante : c'est la seule qui vous dit qu'il faut réveiller quelqu'un cette nuit. Les autres peuvent attendre lundi.

Pièges : compter tous les échecs d'abord, puis chercher les succès — vous détecteriez un succès survenu **avant** l'attaque · oublier que `for` apparaît aussi dans « Failed password **for** ».

Exercice 5 — Détecter un scan de ports

Notions : `set`, comptage de valeurs distinctes.

Une ligne de pare-feu :

```
2026-10-05 03:12:00 45.134.26.88 192.168.1.10 415 DROP
[0] date   [1] heure [2] source  [3] dest     [4]  [5] action
```

Quelqu'un qui cherche une porte ouverte frappe à beaucoup de portes. Complétez `regle_scan_ports(lignes_firewall)` : pour chaque adresse source, comptez les **ports distincts** qu'elle a contactés, en ne regardant que les lignes `DROP`.

Au-delà de `SEUIL_PORTS` (20), c'est un scan. Gravité `elevee`.

```
Résultat attendu : 1 alerte, pour 45.134.26.88, avec 120 ports.
```

Vérification : `pytest -k exercice5`

Le point clé : un `set`, pas une liste. Si une adresse contacte 50 fois le port 443, cela fait **un** port, pas 50. Avec une liste vous compteriez 50 et déclencheriez une fausse alerte sur du trafic parfaitement normal.

Pièges : compter toutes les lignes au lieu des ports distincts · oublier de filtrer sur `DROP` · déclencher sur les machines internes `192.168.x.x`.

Exercice 6 — Détecter les attaques applicatives

Notions : recherche de motifs, comparaison insensible à la casse.

Une ligne de journal web :

```
203.0.113.199 - [2026-10-05 14:22:17] "GET /produits?id=1 UNION SELECT username,password FROM users--" 500 0
```

La liste `MOTIFS_WEB` contient les traces d'attaque à rechercher : injections SQL, XSS, traversée de répertoire. Complétez `regle_attaques_web(lignes_web)`.

Pour chaque adresse, comptez les requêtes malveillantes et le type de chacune. Une alerte `elevee` par adresse.

```
Résultat attendu : 1 alerte, pour 203.0.113.199, 21 requêtes malveillantes.
```

Comparez en minuscules : `motif in ligne.lower()`. Un attaquant qui écrit `UnIoN SeLeCt` passerait sinon au travers — et c'est une technique d'évasion classique.

Vérification : `pytest -k exercice6`

Pièges : comparer sans passer en minuscules · créer une alerte par requête au lieu d'une par adresse · déclencher sur des pages normales (`/recherche?q=chaussures` ne doit rien produire).

Exercice 7 — Détecter les connexions hors horaires

Notions : extraction d'un entier, intervalle.

Une connexion réussie à 4 heures du matin n'est pas une preuve. C'est un signal faible — et un bon SOC les remonte quand même, en gravité moindre.

Complétez `regle_hors_horaires(lignes_auth)` : signalez les `Accepted password` survenus entre `HEURE_DEBUT_NUIT` et `HEURE_FIN_NUIT`. Gravité `moyenne`.

L'heure est dans le 3e mot : `Oct 3 04:25:18` → `04` → `4`.

```
Résultat attendu : 1 alerte moyenne.
```

Vérification : `pytest -k exercice7`

Piège principal : comparer des heures comme du texte. `"04" > "20"` est **vrai** en Python, parce que la comparaison est alphabétique. Convertissez en entier avec `int()`.

Exercice 8 — Assembler le détecteur

Notions : composition de fonctions, tri par clé, écriture de fichier.

1. Complétez `analyser(dossier)` : lire les trois journaux, appliquer les cinq règles, et trier les alertes **par gravité, les plus graves en premier**.

Indice pour le tri : `alertes.sort(key=lambda a: SEVERITES.index(a["severite"]))`

2. Complétez `afficher(alertes)` : une présentation lisible à l'écran.
3. Complétez `ecrire_rapport(alertes, chemin)` : le même contenu, dans un fichier, pour le joindre à un ticket d'incident.

```
Résultat attendu : 5 alertes au total — 2 critiques, 2 élevées, 1 moyenne.
                   Les trois journaux sont représentés.
```

Vérification : `pytest -k exercice8`, puis `pytest -q` → **20 passed**

Ce que raconte votre outil

Une fois terminé, relisez vos cinq alertes dans l'ordre. Elles racontent une histoire, et c'est à vous de la reconstituer :

> Qui a attaqué ? Par où ? A-t-il réussi ? Sur quel compte ? Et les deux autres adresses : même attaquant, ou trois incidents distincts ?

Rédigez cette histoire en dix lignes. C'est le livrable d'un analyste SOC : pas une liste d'alertes, un récit avec des preuves.

Pour aller plus loin

Piste	Ce que vous ajoutez
Une sixième règle	Choisissez-la vous-même et justifiez son seuil
Liste blanche	Ne jamais alerter sur les adresses internes de confiance
Sévérité dynamique	20 échecs = élevée, 90 échecs = critique
Export JSON	Pour envoyer vos alertes à un autre outil
Interface web	Reprendre Flask et afficher le tableau de bord dans un navigateur

---

Grille d'évaluation

Compétence du référentiel	Vérifiée par
C16 — langages mobilisés, standards respectés	Exercices 1 à 8
C17 — risques identifiés, robustesse aux attaques	Exercices 3 à 7
C23 — analyse des journaux, revue de sécurité	Exercice 8 et le récit
C31 — les logs de connexion sont analysés pour avoir un contexte	Exercices 4 et 8
C31 — les utilisateurs ayant eu accès aux données sont identifiés	Exercice 4

Barème	Points
Exercices 1 et 2 — lire et extraire	3
Exercice 3 — force brute	3
Exercice 4 — compromission	4
Exercice 5 — scan de ports	3
Exercice 6 — attaques applicatives	3
Exercice 7 — hors horaires	2
Exercice 8 — assemblage et rapport	2

Bonus : +2 pour une sixième règle pertinente, avec son seuil justifié.

Retrait : −2 si votre outil déclenche une alerte sur du trafic normal. Un détecteur qui crie tout le temps n'est jamais écouté : on finit par le couper, et c'est ce jour-là qu'on se fait avoir.
