Projet Master : SIEM Minimaliste

Master 2 Cyber 2024 - 2025

Livrable n°3

Sablayrolles Adrien

FACULTÉ
GESTION, ÉCONOMIE,
SCIENCES
Université Catholique
de Lille 1875

SIEM Minimaliste – Rapport Final

Lien Github : https://github.com/Adrien-s/SIEM_minimalist

Sommaire :

1. Introduction

Ce rapport présente le développement d'un système d'information et d'événements de sécurité (SIEM) minimaliste, réalisé dans le cadre de mon projet de Master.

L'objectif principal est de démontrer qu'il est possible collecter, stocker, corréler et visualiser les journaux Windows à l'aide d'un ensemble d'outils légers entièrement basés sur Python.

Le prototype se veut simple à installer, autonome, et suffisamment complet pour montrer quelques-unes des principales fonctionnalités attendues d'un SIEM, sans dépendances lourdes ni infrastructure externe.

De plus le projet devait répondre aux critères suivants :

- **Collecte de logs** (Syslog, Windows Event Logs, journaux SSH, etc.) sans perte d'information.
- **Analyse des données** via détection d'anomalies et application de règles statiques.
- **Visualisation et suivi des alertes** par le biais d'un tableau de bord web simple et responsive.
- **Notification et gestion des alertes**, incluant l'archivage, la consultation historique et la possibilité de clôturer ou de commenter chaque alerte.

2. Contexte et motivations

Face à la multiplication des cybermenaces, la surveillance continue des journaux d'événements devient une exigence critique pour les organisations. Les systèmes SIEM permettent de centraliser cette surveillance, d'automatiser l'analyse, et de détecter les comportements suspects. Cependant, la plupart des solutions commerciales sont complexes à déployer, coûteuses, et nécessitent une infrastructure avancée.

Dans ce contexte, ce projet vise à fournir une alternative pédagogique, qui soit :

- Gratuite et facile à installer sur un poste Windows ;
- Fonctionnelle en tant que preuve de concept ;
- Capable d'illustrer les briques essentielles d'un SIEM
- Réalisée exclusivement avec Python pour le back-end et HTML/CSS/JS pour le front-end, sans frameworks ni conteneurs lourds.

3. État de l'art

Il existe plusieurs solutions open source matures pour la détection d'événements de sécurité, notamment :

- **ELK Stack** (Elasticsearch, Logstash, Kibana) ;
- **Wazuh**, basé sur OSSEC ;
- **HELK** (Hunting ELK).

Bien que puissantes, ces solutions requièrent souvent :

- de nombreux services tiers (Java, Elasticsearch, Kibana…) ;
- une infrastructure Docker ou des machines virtuelles ;
- une configuration complexe ;
- une consommation importante de mémoire et de ressources système.

L'approche retenue ici fait le choix inverse : viser la simplicité maximale. Le système repose sur :

- un unique exécutable Python ;
- la bibliothèque pywin32 pour accéder directement aux journaux Windows ;
- SQLite comme base de données embarquée ;

- une interface web locale basée sur Chart.js.

Ce positionnement rend le SIEM minimaliste particulièrement adapté à des usages pédagogiques, des démonstrations hors ligne, ou des tests rapides en environnement local.

4. Architecture globale

4.1 Vue d'ensemble

· Système : le système SIEM minimaliste repose sur une architecture en quatre couches fonctionnelles. Cette organisation modulaire facilite la compréhension, la maintenance et l'évolution du système.

· Collecte : deux agents Python s'exécutent localement sur le poste ou le serveur surveillé. Ils utilisent la bibliothèque pywin32 pour accéder directement aux journaux d'événements Windows (par exemple, Security, System, Application). Chaque événement est lu, filtré, puis transmis à la couche de stockage.

· Stockage : les événements collectés sont stockés dans une base de données SQLite locale. Cette solution de stockage embarquée garantit une persistance fiable (ACID), sans nécessiter de serveur externe ni de configuration complexe. L'accès est optimisé via un thread dédié (`DB Writer`) qui effectue les écritures en batch pour améliorer les performances disque.

· Corrélation : un moteur de règles simple, écrit en Python, parcourt les événements stockés pour identifier des modèles ou comportements anormaux (ex. : tentatives de connexion multiples, élévation de privilèges, arrêt de services critiques). Ce moteur peut être enrichi progressivement avec des règles personnalisées pour renforcer la détection.

· Visualisation : un dashboard minimaliste est intégré grâce à un serveur HTTP standard Python et à Chart.js pour la représentation graphique. L'interface permet de :

- Visualiser les événements récents ou filtrés par type,
- Surveiller les alertes générées par la corrélation,
- Afficher des statistiques et tendances (histogrammes, séries temporelles, etc.).

4.2 Choix techniques clés

· pywin32 – accès natif aux journaux Windows sans dépendance externe ,

- Permet d’accéder directement à l’Event Log Windows via win32evtlog, sans passer par des solutions comme WMI ou agents tiers.
- Léger, fiable, intégré aux systèmes Windows.
- Convient pour une installation sans infrastructure lourde.

· SQLite – base embarquée ACID, zéro configuration :

- Pas besoin d’un serveur de base de données externe (MySQL, PostgreSQL).
- Très adaptée pour des charges légères à modérées.
- Facilement portable, un seul fichier .db à déplacer/sauvegarder.

· Thread DBWriter – pour améliorer les performances d’insertion dans la db:

- Améliore les performances d’écriture en réduisant les accès disques fréquents.
- Permet de découpler la collecte des logs de leur insertion en base.
- Réduit les risques de blocage ou ralentissement du thread principal.

· Chart.js – bibliothèque JavaScript légère pour graphiques interactifs :

- Visualisation intuitive de données (comptes d’événements, timelines, heatmaps...).
- Zéro dépendance serveur, s'intègre directement dans une page HTML.
- Idéale pour une interface web simple mais efficace.

- Serveur HTTP Python standard :

- Utilisation de http.server ou BaseHTTPServer selon la version Python.
- Réduit la surface d’attaque et les dépendances.
- Parfait pour un tableau de bord local ou des cas de test/démonstration.

En conclusion, mes solutions montrent un choix de simplicité, de portabilité et de performance sans complexité inutile. Le système peut être lancé rapidement, avec peu de ressources, tout en restant *extensible* pour des environnements plus complexes.

4.3 Diagramme UML

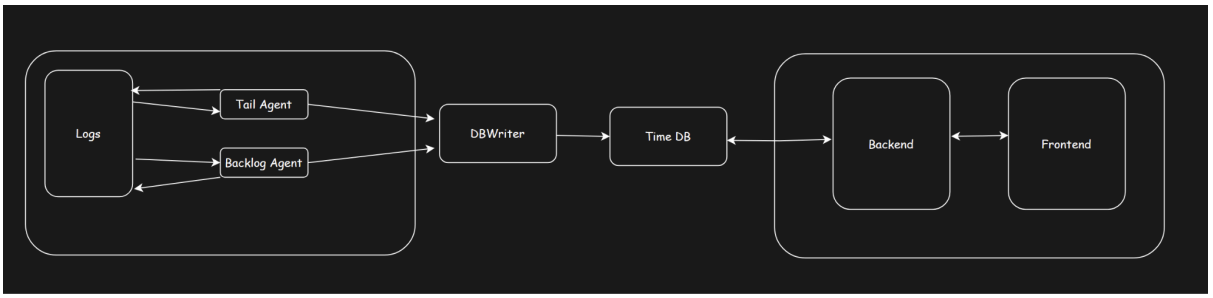

Figure 1: Diagramme UML

Légende :

- **Agents Collecte** : scripts Python tournant en service ou tâche planifiée.
- **DBWriter** : composant assurant l'insertion performante des événements.
- **Time db (Base SQLite)** : point central du stockage, partagé entre analyse et visualisation.
- **Backend** : analyse les données, déclenche des alertes.
- **Frontend** : permet à l'utilisateur de consulter événements et alertes.

Figure 2 : Alerte sur le dashboard

5. Implémentation détaillée

5.1 Collecte des journaux

La collecte repose sur deux fils d'exécution complémentaires, chacun ayant une responsabilité spécifique pour garantir l'exhaustivité des données :

- BacklogAgent : lit l'historique complet des journaux Windows au démarrage de l'agent. Il s'assure que tous les événements passés sont récupérés, même ceux générés avant l'activation du système.

- TailAgent : suit les nouveaux événements en temps réel (mode "tail"). Il interroge régulièrement les journaux afin de capturer les entrées dès leur apparition.

Ce découpage permet une transition fluide entre la collecte initiale et la surveillance continue, sans risque de perte d'événements.

5.2 Stockage SQLite

La base de données SQLite a été retenue pour sa simplicité, sa portabilité et son support ACID. Elle ne nécessite aucun serveur ou configuration complexe.

Optimisations mises en œuvre :

- **Mode WAL (Write-Ahead Logging)** : activé pour permettre des lectures simultanées pendant l'écriture, ce qui améliore la réactivité du tableau de bord et du moteur de corrélation.

- **Thread DBWriter** : un thread dédié reçoit les événements via une file de messages interne, les regroupe, puis les écrit en batches. Cela réduit la charge disque et augmente les performances globales, en particulier dans les environnements avec un volume modéré mais constant d'événements.

5.3 Moteur de règles

La gestion de règles repose sur une logique simple mais efficace, adaptée aux détections basées sur la fréquence. Il s'exécute toutes les 60 secondes, et évalue un ensemble de règles définies de manière déclarative.

Chaque règle comprend :

- **Canal** : le journal concerné (ex. : Security, System ou autre) ;

- **Identifiant d'événement** : l'ID spécifique à surveiller ;

- **Fenêtre temporelle (W)** : durée (en minutes) pendant laquelle les événements sont comptabilisés ;

- **Seuil (N)** : nombre d'occurrences à partir duquel une alerte est générée.

Si la condition de fréquence est remplie, une entrée d'alerte est ajoutée à la base.

5.4 Tableau de bord web

L'interface utilisateur est conçue pour être légère, autonome et responsive, sans dépendre de frameworks lourds.

Composition :

- Un **fichier HTML unique**, enrichi de CSS minimalistes pour la présentation ;
- Un **script JavaScript** basé sur Chart.js**,** utilisé pour tracer des graphiques dynamiques à partir des données collectées ;
- Une **API REST légère**, servie par le module HTTP intégré de Python, fournit les événements, statistiques et les alertes.

Cette architecture permet une visualisation efficace des données locales, tout en restant facile à déployer et sans dépendance serveur externe.

6. Les tests

La suite pytest vérifie le bon fonctionnement des composants :

· Agents de collecte – simulation d'événements Windows ;

· Base SQLite – création, insertion et requêtes ;

· Service des définitions d'événements – recherche par ID ;

· Moteur de règles – détection et non-détection selon les seuils.

Le système dispose d'une couverture de tests unitaires développés à l'aide du framework pytest, choisi pour sa simplicité, sa lisibilité et son intégration aisée dans un projet Python.

Objectifs des tests :

- **Vérifier l'intégrité fonctionnelle** des composants clés ;
- **Détecter les régressions** lors des évolutions du code ;
- **Valider les cas limites** (seuils, absence d'événements, etc.) ;

Composants testés :

● **Agents de collecte**

Des événements Windows simulés permettent de tester :

- la capacité à lire correctement différents types d'entrées ;
- le filtrage et la normalisation des données ;
- la transmission correcte vers le thread de stockage.

● Base SQLite

Les tests couvrent :

- la création de la base et des tables ;
- l'insertion d'événements unitaires et en lot ;
- la cohérence des requêtes (filtrage, agrégation, etc.).

● Service des définitions d'événements

Ce service permet d'associer un ID d'événement Windows à une description lisible. Les tests valident :

- la recherche par ID ;
- la gestion des ID inconnus ou non définis ;
- la conformité des métadonnées retournées.

● Moteur de règles

Des jeux de données synthétiques testent le moteur dans des situations contrôlées :

- **détection conforme** lorsque le seuil d'occurrences est atteint dans la fenêtre définie ;
- **non-détection** lorsque les conditions ne sont pas remplies (ex. : seuil trop bas, période trop longue) ;
- **robustesse** face à des règles mal formées ou incohérentes.

Cette stratégie de tests vise à assurer la fiabilité du SIEM, tout en facilitant l'ajout de nouvelles fonctionnalités dans un cadre contrôlé. Elle peut être étendue avec des tests d'intégration ou de performance selon les besoins futurs.

7. Discussion

Le prototype atteint pleinement son objectif de simplicité, tant sur le plan technique que fonctionnel. Grâce à l'utilisation exclusive de composants légers (agents Python, SQLite, serveur HTTP standard), il reste facilement déployable dans un environnement local, de test ou d'apprentissage.

Cependant, des limitations apparaissent dès que le volume de données augmente :

- **SQLite**, bien qu'efficace pour des charges modérées, montre ses limites en termes de performances et de concurrence dès que l'on dépasse quelques centaines de milliers d'événements.

- L'absence de file de messages rend le traitement sensible aux pics de charge, et empêche de découpler proprement la collecte de l'analyse.

Si le projet est destiné à une mise en production ou une montée en charge, il serait pertinent d'envisager :

- L'adoption d'un **SGBD client-serveur** comme **PostgreSQL**, pour une meilleure gestion de la concurrence et du volume.

- L'introduction d'une **file de messages** telle que **Apache Kafka** ou **RabbitMQ**, afin de fiabiliser la transmission des événements et d'absorber les pics.

8. Perspectives

Plusieurs axes d'amélioration et d'enrichissement sont envisageables pour faire évoluer ce prototype vers une solution plus complète :

- **Support d'autres sources de logs** : intégration de Syslog, AWS CloudTrail, ou encore des logs applicatifs personnalisés.

- **Notifications en temps réel** : envoi d'alertes via e‑mail, Slack, Microsoft Teams, ou autres systèmes de messagerie.

- **Contrôle d'accès** : mise en place d'une authentification et de droits utilisateurs différenciés pour accéder aux différents modules (lecture, administration...).

- **Conteneurisation avec Docker** et intégration dans une **chaîne CI/CD**, pour un déploiement automatisé et reproductible.

9. Argumentaire sur le choix des technologies

Choix prévu (L1/L2)	Raison principale de l'abandon	Commentaire
Conteneurisation Docker	Sa mise en œuvre aurait nécessité : la gestion des images et des dépendances ; la configuration des montages de volumes pour accéder aux fichiers journaux EVTX ; l'exposition des ports réseau pour le tableau de bord ; et surtout, des droits administrateurs pour permettre à un conteneur de lire les journaux système, ce qui complique l'accès direct aux fichiers EVTX. Dans le cadre d'un démonstrateur mono-poste, un simple exécutable Python natif est plus adapté : il est rapide à installer, facile à dépanner, et fonctionne sans surcouche.	Docker reste pertinent pour un déploiement en production ;
Base PostgreSQL	Besoin d'un serveur externe, droits réseau et sauvegardes ; la cible est un poste Windows autonome. SQLite offre la conformité et la simplicité nécessaire au début du projet.	Le schéma SQL a été conçu pour rester compatible avec PostgreSQL afin de faciliter une future migration. (La compatibilité n'a pas été testée mais cela devrait normalement fonctionner). (

Collecte multi-sources (Syslog, SSH, …)	Manque de temps et de matériels de test. Se concentrer sur Windows Security / Application a permis d'aboutir à un livrable fonctionnel.	Les modules de collecte sont écrits de façon modulaire ; un connecteur Syslog pourra être ajouté sans toucher aux autres couches. (En théorie, un connecteur Syslog pouvait être ajouté mais cela n'a pas été testé).

10. Conclusion

La preuve de concept montre qu'un SIEM léger et portable est réalisable avec peu de dépendances. Le projet constitue une base pédagogique et un point de départ pour des évolutions futures plus robustes.

Ce projet démontre qu'il est possible de concevoir un **SIEM minimaliste, fonctionnel et portable** en s'appuyant uniquement sur des technologies standard et peu gourmandes.

La solution proposée constitue :

- un **outil pédagogique** pour comprendre les principes fondamentaux de la sécurité SIEM (collecte, stockage, corrélation, visualisation) ;
- un **point de départ évolutif** vers des architectures plus robustes et évolutives.