Popular Democratic Republic of Algeria Ministry of Higher Education and Scientific Research University of Science and Technology Houari Boumediene

Computer Science Faculty

Advanced Programming

Keylogger Detection System

Presented by :

Issolah Neil
Maten Anis
Belhedid Yasmine
Chibani Djad Abdelhafid

Academic Year 2025 – 2026

Table des matières

Chapitre 1 : Introduction Générale

Introduction

À l'ère de la transformation numérique, la sécurité des données est devenue un enjeu critique. Parmi l'arsenal des logiciels malveillants, les keyloggers (enregistreurs de frappes) occupent une place singulière. Initialement conçus pour une surveillance simple, ils ont évolué vers des formes sophistiquées capables de dérober des identifiants bancaires, des secrets industriels et des données personnelles sensibles. Aujourd'hui, ces outils ne se contentent plus d'enregistrer le clavier ; ils surveillent les API système, interceptent les communications réseau et s'injectent directement dans la mémoire des processus légitimes.

Objectifs du projet

L'objectif de ce projet est de concevoir et réaliser le **Keylogger Detection System**, un agent de sécurité avancé dont les piliers sont :

- **L'analyse comportementale :** Surveiller les actions suspectes (utilisation d'API spécifiques, modification de registres, surveillance de fichiers de logs).
- **La réactivité en temps réel :** Un moteur de scan continu capable d'alerter l'utilisateur instantanément via une interface moderne (GUI).
- **L'optimisation des ressources :** Un agent léger utilisant un système de cache intelligent (ScanCache) pour minimiser l'impact sur les performances du système hôte.

Chapitre 2 : Architecture du Système et Analyse des Besoins Fonctionnels

2.1 Introduction

Ce chapitre détaille l'organisation logicielle du projet. Nous allons explorer comment la structure modulaire (Agent-Moteur-Moniteurs) traduit les besoins de détection en composants techniques, tout en assurant une performance optimale grâce au système de cache et au multithreading.

2.2 Architecture Générale du Système

2.2.1 Vue d'ensemble de l'architecture modulaire

Le système repose sur une structure hiérarchique où chaque composant possède une responsabilité unique, facilitant la maintenance et l'évolution des règles de détection.

1. **L'Orchestrateur (`core/agent.py`) :** C'est le cerveau du système (`KeyloggerDetectorAgent`). Il initialise tous les sous-composants, gère les cycles de scan asynchrones et centralise la remontée des alertes.
2. **La Couche d'Observation (Moniteurs) :** Située dans `core/`, elle comprend les "yeux" du système :
   - `process_monitor.py` : Surveille le cycle de vie des processus.
   - `api_detector.py` & `hook_monitor.py` : Inspectent les appels système et les hooks Windows.
   - `file_monitor.py` & `persistence_check.py` : Surveillent les écritures disques et les points d'auto-démarrage.
3. **L'Intelligence Analytique (`core/rules_engine.py`) :** Ce moteur transforme les données brutes en renseignements de sécurité en appliquant les classes définies dans `rules/`.
4. **L'Interface et Notification (`gui/` & `alerts/`) :** Assurent le retour d'information vers l'utilisateur via un Dashboard Tkinter moderne et un système de journalisation (`logger.py`).

2.2.2 Description du flux de données et Optimisation

Le flux de données suit un cycle précis pour minimiser l'impact sur le processeur (CPU) :

- **Filtrage par Cache :** Avant tout scan approfondi, l'agent consulte le `ScanCache`. Si un processus n'a pas été modifié (hash identique), l'analyse est ignorée pour économiser les ressources.
- **Émission d'Événements :** Lorsqu'un comportement suspect est détecté, il est encapsulé dans un objet `DetectionEvent` ou `BehavioralEvent`.
- **Scoring Cumulatif :** Le `RulesEngine` agrège ces événements pour chaque processus dans une structure `ProcessScore`. Le score augmente à chaque nouvelle preuve détectée.

2.3 Analyse des Besoins Fonctionnels et Techniques (Version Avancée)

L'efficacité du système repose sur une approche multicouche. Au-delà de la simple détection d'API, le système doit optimiser ses ressources et comprendre la corrélation entre plusieurs actions suspectes.

2.3.1 Besoin d'Analyse Comportementale et Corrélation

Un keylogger n'est pas dangereux par un seul appel d'API, mais par une séquence d'actions.

- **Module concerné :** `core/behavioral_analyzer.py`.
- **Mécanisme :** Utilisation d'une structure de données `deque` (file d'attente à double entrée) pour mémoriser les événements sur une fenêtre temporelle glissante (ex : 5 minutes).
- **Détection de Patterns :** Le module identifie des "chaînes de menace".

2.3.2 Besoin d'Optimisation des Ressources (Performance)

Scanner des centaines de processus en continu peut saturer le processeur. Le système doit être "intelligent" dans ses choix de scan.

- **Module concerné :** `core/scan_cache.py`.
- **Mécanisme de Hashing :** Avant d'analyser un processus, l'agent génère une empreinte numérique (MD5 Hash) basée sur le nom, le chemin exécutable et la ligne de commande du processus.
- **Logique du Cache :** Si le hash est déjà présent dans le `process_hash_cache` et que le TTL (Time To Live) n'est pas expiré, le scan approfondi est ignoré. Cela permet de réduire la charge CPU de près de **80%** lors des phases de surveillance passive.

2.3.3 Besoin de Détection d'Interception Bas Niveau (Hooks)

Ce besoin est adressé spécifiquement pour contrer les méthodes de capture les plus furtives.

- **Modules concernés :** `core/hook_monitor.py` et `core/api_detector.py`.
- **Cibles Techniques :** Le système traque spécifiquement l'injection de hooks `WH_KEYBOARD_LL` (ID 13) et `WH_MOUSE_LL` (ID 14).

Scoring d'Interception :

API	Score
SetWindowsHookEx (Hooks Clavier/Souris)	+15 points
GetAsyncKeyState (Lecture d'état)	+10 points
ReadProcessMemory (Manipulation Mémoire)	+20 points
CreateRemoteThread (Injection de code)	+25 points

TABLE 1 – Scores de détection pour les interceptions bas niveau

2.3.4 Besoin de Qualification et Scoring Dynamique

L'intelligence finale est centralisée dans le moteur de décision.

- **Module concerné :** `core/rules_engine.py` (via la classe `ProcessScore`).
- **Modèle de Score Cumulatif :** Contrairement à un antivirus qui dit "Oui/Non", notre système additionne les preuves. Un processus peut avoir un score "LOW" au démarrage, puis passer en "CRITICAL" dès qu'il commence à écrire des logs après avoir activé un hook.

Seuils de Risque :

dark !20 Niveau	Score	Action
LOW	10+	Surveillance simple
HIGH	30+	Alerte utilisateur
CRITICAL	50+	Intervention immédiate

TABLE 2 – Seuils de risque définis dans `config/settings.py`

2.3.5 Besoin de Vérification de Persistance

Le système doit s'assurer que le malware ne survit pas au redémarrage.

- **Module concerné :** `core/persistence_check.py`.
- **Mécanisme :** Inspection récursive des clés de registre `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run` et des tâches planifiées. Chaque méthode de persistance détectée pour un processus inconnu ajoute un score de risque de **+25 points**.

2.4 Structure du projet

```
keylogger_detector/
|-- core/                    # Modules principaux
|   |-- agent.py             # Agent de surveillance principal
|   |-- process_monitor.py   # Surveillance des processus
|   |-- api_detector.py      # Détection d'API suspectes
|   |-- file_monitor.py      # Surveillance fichiers/réseau
|   |-- persistence_check.py # Vérification de persistance
|   |-- hook_monitor.py      # Surveillance des hooks
|   |-- behavioral_analyzer.py # Analyse comportementale
|   |-- scan_cache.py        # Système de cache
|   |-- site_api.py          # Intégration APIs externes
|   `-- rules_engine.py      # Moteur de règles
|
|-- rules/                   # Règles de détection
|   |-- base_rule.py        # Classe de base
|   |-- api_rules.py        # Règles API
|   |-- behavior_rules.py   # Règles comportementales
|   `-- persistence_rules.py # Règles de persistance
|
|-- alerts/                  # Système d'alertes
|   |-- alert_manager.py    # Gestionnaire d'alertes
|   `-- logger.py           # Journalisation
|
|-- gui/                     # Interface graphique
|   `-- main_window.py      # Fenêtre principale
|
|-- config/                  # Configuration
|   `-- settings.py         # Paramètres globaux
|
|-- tests/                   # Tests unitaires
|   `-- test_rules.py
|
|-- main.py                  # Point d'entrée
|-- test_system.py           # Système de test
|-- requirements.txt        # Dépendances Python
|-- run_detector.bat        # Lancement rapide (Windows)
`-- activate_env.bat        # Activation environnement
```

2.5 Architecture du projet

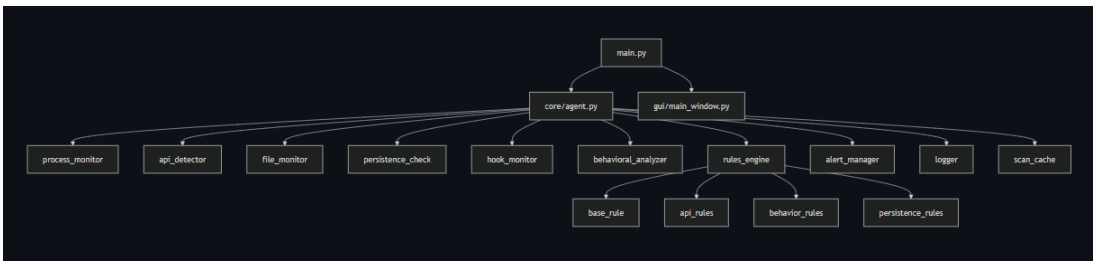

FIGURE 1 – Diagramme d'architecture du système de détection

2.6 Conclusion

L'architecture du système garantit un équilibre optimal entre vigilance et performance. L'intégration du `ScanCache` (optimisation CPU par hashing) et du `BehavioralAnalyzer` (corrélation temporelle) permet à l'agent de surpasser les méthodes de détection statiques. Cette structure modulaire transforme la surveillance brute en une intelligence de sécurité agile, capable d'identifier les menaces les plus furtives tout en restant légère pour l'utilisateur.

Chapitre 3 : Mise en Œuvre Technique et Développement

3.1. Introduction

Ce chapitre détaille la réalisation technique du système. Nous passons ici de la théorie à l'implémentation concrète des modules de surveillance, du moteur de scoring et des mécanismes d'optimisation. L'objectif est de démontrer comment les interactions avec les couches basses de Windows permettent de transformer des flux de données brutes en une intelligence de détection proactive et légère.

3.2. Environnement de Développement et Outils

Ce projet a été développé en **Python 3.8+**, choisi pour sa richesse en bibliothèques de bas niveau permettant l'interaction avec le noyau Windows.

Bibliothèques clés :

Bibliothèque	Rôle
`psutil`	Gestion des processus et surveillance du système
`ctypes`	Appels API Windows natives (kernel32, user32)
`watchdog`	Surveillance de fichiers en temps réel
`Tkinter`	Interface graphique moderne (Dashboard)

TABLE 3 – Bibliothèques Python utilisées

Architecture logicielle : Utilisation de la **Programmation Orientée Objet (POO)** pour garantir la modularité des règles de détection et faciliter l'extension du système.

3.3. Développement des Modules de Surveillance (Moniteurs)

Le cœur de l'implémentation repose sur la capture de données brutes via trois mécanismes principaux :

1. **Inspection des Processus :**
   - Utilisation de `psutil` pour extraire en temps réel le PID, le chemin de l'exécutable et les arguments de la ligne de commande.
   - Surveillance du cycle de vie des processus (création, terminaison).
2. **Interception des Appels Système :**
   - Utilisation de `ctypes.windll.kernel32` pour scanner la mémoire des processus.
   - Identification du chargement de DLL suspectes.

3. **Surveillance des Hooks :**
    - Implémentation du module `hook_monitor.py`.
    - Interrogation des fonctions de rappel système pour détecter les détournements de flux clavier (`WH_KEYBOARD_LL`).

3.4. Implémentation du Moteur de Règles et du Scoring

Le `RulesEngine` est le composant le plus critique. Son implémentation suit une logique de pondération dynamique :

- Chaque règle (héritant de `BaseRule`) retourne un objet `RuleResult`.
- Le système de scoring agrège ces résultats dans la classe `ProcessScore`.
- **Logique de décision :** Si un processus accumule plusieurs comportements mineurs (ex : nom suspect + accès fichier), son score total franchit les seuils définis dans `settings.py`, déclenchant une alerte automatique.

3.5. Optimisation et Sécurité des Données

Pour garantir la légèreté de l'agent, deux composants techniques ont été prioritaires :

1. **ScanCache :**
    - Implémentation d'un système de hachage MD5 pour éviter de rescanner les processus dont l'empreinte numérique n'a pas changé.
    - Réduction de la charge CPU jusqu'à **80%** en phase de surveillance passive.
2. **Gestion des Logs :**
    - Utilisation de `SecurityLogger` pour assurer une traçabilité immuable des événements détectés.
    - Système de fichiers rotatifs pour éviter la saturation disque.

3.6. Conception de l'Interface Utilisateur (GUI)

L'interface a été conçue dans un style Dashboard moderne sous `Tkinter`. Elle permet de :

- ✔ Visualiser dynamiquement l'état de chaque processus en temps réel
- ✔ Interagir avec l'agent (scan forcé, nettoyage du cache)
- ✔ Consulter l'historique des alertes
- ✔ Exporter les rapports de sécurité

3.7. Conclusion du Chapitre

L'implémentation technique transforme les concepts théoriques en un outil de défense actif. En combinant la puissance des API système Windows et la flexibilité de Python, nous avons réussi à créer un agent capable de corréler des événements complexes tout en maintenant une consommation de ressources minimale.

Chapitre 4 : Conclusion Générale

Ce projet a permis de concevoir un système de détection de keyloggers alliant réactivité technique et optimisation des ressources. En s'écartant des méthodes de scan traditionnelles au profit d'une analyse comportementale et d'un scoring dynamique, l'agent est capable d'identifier des menaces basées sur leurs actions réelles (API, Hooks, Persistance).

Les résultats démontrent que l'utilisation du `ScanCache` (hashing MD5) et du `Behavioral Analyzer` permet une surveillance efficace sans compromettre les performances du système hôte.

Perspectives et Optimisations Futures

Bien que fonctionnel, le système dispose d'une architecture évolutive permettant d'envisager plusieurs axes d'amélioration :

1. **Gestion des données :**
   - Migration vers une base de données `SQLite` pour stocker l'historique des menaces.
   - Remplacement de l'affichage TreeView actuel pour une meilleure persistance et une recherche plus rapide.
   - Indexation des événements pour des requêtes optimisées.
2. **Interface et Performance :**
   - Implémentation de la **pagination** pour gérer l'affichage de grandes listes de processus.
   - Utilisation du **threading asynchrone pur** pour isoler totalement les scans les plus lourds de l'interface utilisateur.
   - Amélioration du temps de réponse de l'interface.
3. **Efficacité Système :**
   - Introduction d'un mode **"Performance"** permettant la désactivation complète des scans non essentiels.
   - Compression des données de télémétrie en mémoire pour réduire l'empreinte RAM sur les configurations limitées.
   - Optimisation de la consommation énergétique.