Sistema de Monitoramento Contínuo de Arquivos para Detecção de Erros e Ameaças Digitais

Ryann Flávyo Alves Honorato Lessa

23 Abril de 2025

1 - Introdução

1.1 - Objetivo

Este projeto propõe o desenvolvimento de um sistema automatizado de monitoramento contínuo de arquivos com foco em verificação de integridade, identificação de modificações não autorizadas e prevenção de incidentes de segurança da informação. Com base em algoritmos criptográficos de hash e varredura recursiva de diretórios, o sistema visa atuar como uma ferramenta preventiva e reativa contra corrupção de dados, infecções por malware e manipulações indevidas em ambientes sensíveis.

1.2 - Motivação

Em um cenário onde ataques como ransomware, rootkits e adulteração de pacotes são cada vez mais frequentes, assegurar a integridade de arquivos tornou-se uma prioridade estratégica. Muitas soluções tradicionais, como antivírus ou agendadores de varredura, atuam de forma passiva ou com janelas de detecção espaçadas, o que permite que ataques persistentes passem despercebidos. Um sistema de monitoramento contínuo baseado em assinaturas únicas dos arquivos (hashes) permite detectar mudanças em tempo quase real, mesmo que silenciosas ou intencionais, mitigando falhas que podem comprometer a segurança de dados em servidores, estações de trabalho e sistemas embarcados.

2 - Organização do Projeto

2.1 - Estrutura Modular

O sistema desenvolvido para monitoramento contínuo de arquivos foi arquitetado de forma modular, respeitando os princípios da coesão e do baixo acoplamento. Isso garante manutenibilidade,

extensibilidade e clareza, atributos fundamentais em projetos de software seguro e escalável. A seguir, descrevemos cada módulo com base em seus fundamentos teóricos, propósitos específicos e estratégias de implementação.

📁 verificador.py

```python
import hashlib
import os

def calcular_hash(caminho_arquivo):
    sha256 = hashlib.sha256()
    try:
        with open(caminho_arquivo, "rb") as f:
            while chunk := f.read(8192):
                sha256.update(chunk)
        return sha256.hexdigest()
    except (PermissionError, FileNotFoundError):
        return None

def carregar_hashes_existentes(arquivo_hashes):
    if not os.path.exists(arquivo_hashes):
        return {}
    with open(arquivo_hashes, "r") as f:
        linhas = f.readlines()
    return dict(linha.strip().split("  ") for linha in linhas if "  " in linha)

def salvar_hashes(arquivo_hashes, dicionario):
    with open(arquivo_hashes, "w") as f:
        for caminho, hash_valor in dicionario.items():
            f.write(f"{caminho}  {hash_valor}\n")
```

Propósito: Responsável por garantir a integridade dos arquivos através do cálculo e verificação de funções hash.

Fundamento Teórico: Utiliza o algoritmo SHA-256, uma função criptográfica da família SHA-2, que gera um identificador único para o conteúdo de arquivos. Essa abordagem é amplamente utilizada em mecanismos de blockchain, verificação de integridade de backups e certificação digital.

Implementação: A função '**calcular_hash**' lê arquivos em blocos de 8192 bytes — um trade-off entre performance e uso de memória — e atualiza um objeto hash incrementalmente. '**carregar_hashes_existentes**' e '**salvar_hashes**' implementam um modelo persistente de verificação de estado anterior, utilizando uma estrutura de dicionário para acesso em tempo constante(1).

Eficiência: O custo computacional para o cálculo de hash é linear no tamanho do arquivo O(n), onde n é o número de bytes.

📁 monitor.py

```python
def monitorar_diretorio(pasta, arquivo_hashes="hashes.txt", intervalo=10):
    print(f"Iniciando monitoramento de: {pasta}")
    hashes_anteriores = carregar_hashes_existentes(arquivo_hashes)

    while True:
        hashes_atuais = {}
        for root, _, arquivos in os.walk(pasta):
            for nome in arquivos:
                caminho = os.path.join(root, nome)
                hash_atual = calcular_hash(caminho)
                if hash_atual is None:
                    continue
                hashes_atuais[caminho] = hash_atual

                if caminho not in hashes_anteriores:
                    registrar_alerta(f"[NOVO] Arquivo detectado: {caminho}")
                elif hash_atual != hashes_anteriores[caminho]:
                    registrar_alerta(f"[ALTERADO] {caminho}")

        for antigo in hashes_anteriores:
            if antigo not in hashes_atuais:
                registrar_alerta(f"[REMOVIDO] {antigo}")

        salvar_hashes(arquivo_hashes, hashes_atuais)
        hashes_anteriores = hashes_atuais.copy()
        time.sleep(intervalo)
```

Propósito: Implementa o mecanismo contínuo de monitoramento e comparação entre estados anteriores e atuais dos arquivos.

Fundamento Teórico: Baseia-se na técnica de *polling ativo*, onde a varredura é executada em um intervalo regular. É uma abordagem simples porém eficaz para detectar alterações em sistemas que não oferecem suporte a notificações em tempo real (como inotify).

Implementação: Utiliza '**os.walk()**' para varredura recursiva dos diretórios, '**time.sleep()**' para espaçamento de ciclos, e faz uso do dicionário de hashes carregado via '**verificador.py**'. A cada iteração, arquivos novos, modificados e removidos são identificados e registrados.

Complexidade: O custo de uma varredura completa é O(n * m), sendo n o número de arquivos e m o custo médio de leitura e hash de cada um.

📁 alerta.py

```python
from datetime import datetime

def registrar_alerta(mensagem):
    hora = datetime.now().strftime("[%d/%m/%Y %H:%M:%S]")
    alerta = f"{hora} {mensagem}"
    print(alerta)
    with open("log.txt", "a") as log:
        log.write(alerta + "\n")
```

Propósito: Gerencia a geração e persistência de alertas de integridade para análise posterior

Fundamento Teórico: Atua como mecanismo de logging simples, utilizando a estratégia de persistência em arquivos texto para rastreamento de eventos. Pode ser integrado a sistemas mais complexos via sockets ou APIs REST.

Implementação: '**registrar_alerta**' gera mensagens timestampadas utilizando `**datetime.now()**` e escreve em '**log.txt**'. A escrita é feita em modo append, mantendo o histórico completo dos eventos.

📁 main.py

```
def menu():
    if len(sys.argv) < 2:
        print("Uso: python src/main.py <diretório> [intervalo_segundos]")
        sys.exit(1)

    pasta = sys.argv[1]
    intervalo = int(sys.argv[2]) if len(sys.argv) > 2 else 10

    if not os.path.isdir(pasta):
        print(f"Erro: o caminho '{pasta}' não é um diretório válido.")
        sys.exit(2)

    monitorar_diretorio(pasta, intervalo=intervalo)

if __name__ == "__main__":
    menu()
```

Propósito: Integra todos os módulos, interpretando argumentos da linha de comando e iniciando a execução do sistema.

Fundamento Teórico: Adota a convenção '**if __name__ == "__main__**' para permitir que o script seja reutilizado como biblioteca, se necessário.

Implementação: Utiliza '**sys.argv**' para capturar argumentos do usuário. Exibe mensagem de ajuda se os parâmetros forem insuficientes. Executa a função '**monitorar_diretorio**' com base nas entradas recebidas.

Robustez: Trata exceções como ausência de argumentos ou entrada malformada, encerrando o programa com código de erro apropriado (**'sys.exit(1)'**).

Considerações Finais: A estrutura modular proposta favorece a expansão do projeto para diferentes camadas de segurança, como análise heurística, verificação em rede, ou integração com antivírus comerciais. A clareza da separação de responsabilidades facilita testes unitários, substituição de módulos e evolução tecnológica do sistema.

2.2 - Execução

A execução do sistema é feita por meio de terminal (linha de comando), garantindo portabilidade e independência de interface gráfica. O usuário deve fornecer dois parâmetros:

- O caminho absoluto ou relativo do diretório a ser monitorado;
- Opcionalmente, o intervalo de tempo (em segundos) entre cada varredura.

Exemplo de execução padrão:

python main.py ./meus_arquivos 10

Esse comando inicia a varredura contínua do diretório **./meus_arquivos**, realizando verificações a cada 10 segundos.

Fluxo interno:

1. O módulo **main.py** valida os argumentos e inicia o monitoramento.

2. O módulo **monitor.py** percorre todos os arquivos da pasta.

3. Cada arquivo tem seu hash calculado pelo **verificador.py**.

4. Se ocorrer qualquer divergência com o histórico, o **alerta.py** gera registros com timestamp.

Boas práticas:

- Execute o sistema em background com **nohup** ou via **systemd** para produção.
- Redirecione logs para arquivos rotativos com **logrotate**.

3 - Funcionamento Técnico

3.1 - Estrutura de Dados Utilizadas

A aplicação faz uso primário de:

- **Dicionários (hash maps):** Para mapeamento de arquivos e seus respectivos hashes.
- **Listas:** Implícitas nas iterações de arquivos e armazenamento temporário de resultados.

3.2 - Algoritmo de Hash e Justificativa

Foi escolhido o **SHA-256**, da família SHA-2, por ser atualmente resistente a colisões e ataques por segunda-pré imagem, diferentemente de algoritmos legados como MD5 ou SHA-1. O uso do SHA-256 garante uma unicidade forte na representação dos arquivos monitorados, reduzindo falsos positivos.

3.3 - Fluxo de Operação

O sistema implementa um laço infinito onde, a cada iteração:

- Percorre todos os arquivos com **os.walk( )**.
- Calcula seus hashes.
- Compara com os valores anteriores salvos em disco.
- Gera alertas para novos, alterados ou removidos.

O tempo entre as varreduras é definido pelo usuário, com uso de **time.sleep(intervalo)**

3.4 - Eficiência e Escalabilidade

A complexidade do algoritmo principal é O(n·m), sendo:

- **n** o número de arquivos monitorados.
- **m** o tempo necessário para leitura e hash de cada arquivo.

Limitações atuais:

- Não diferencia entre alterações autorizadas e maliciosas.
- Pode causar carga alta de disco em sistemas com muitos arquivos ou arquivos grandes.

Possibilidades de otimização:

- Cache inteligente com base em metadados (**mtime**, **size**).
- Notificações reativas com **watchdog** ou **inotify** (Linux).

4 - Conclusão

O sistema de monitoramento contínuo desenvolvido neste projeto representa uma solução leve, extensível e eficaz para ambientes que exigem preservação da integridade de dados. Sua implementação modular, baseada em princípios clássicos da segurança da informação (como controle de integridade e rastreabilidade), permite sua aplicação em contextos variados, desde servidores até ambientes corporativos de alta criticidade.

Além de seu valor instrucional como projeto de graduação, a arquitetura adotada permite futuras expansões, como:

- Integração com bancos de dados de assinaturas de malware;
- Detecção comportamental via aprendizado de máquina;
- Notificações por e-mail, Telegram ou SIEMs.

Trata-se, portanto, de um sistema simples em aparência, mas capaz de ilustrar e aplicar com fidelidade conceitos fundamentais de segurança computacional, programação modular e verificação contínua de integridade.