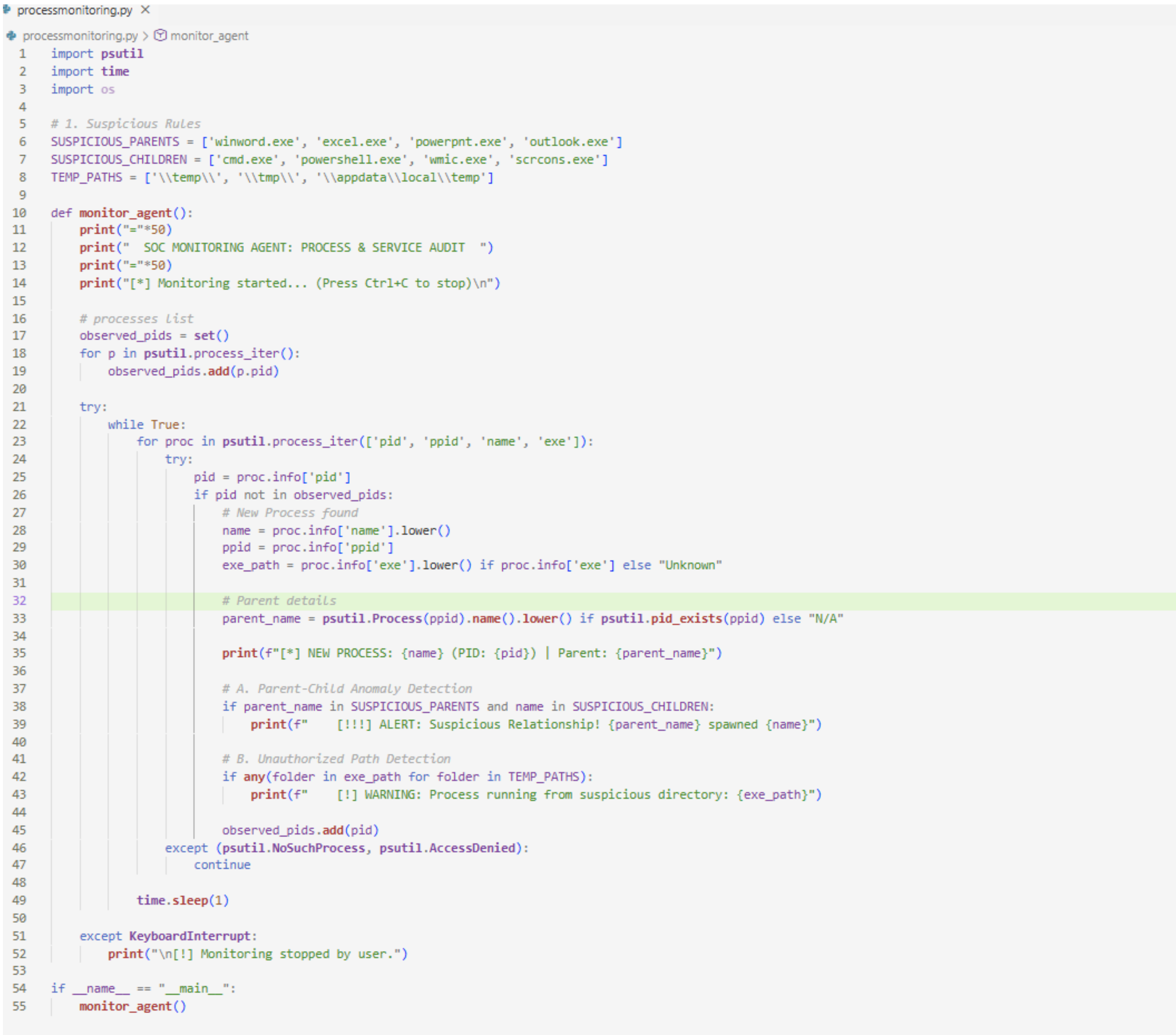
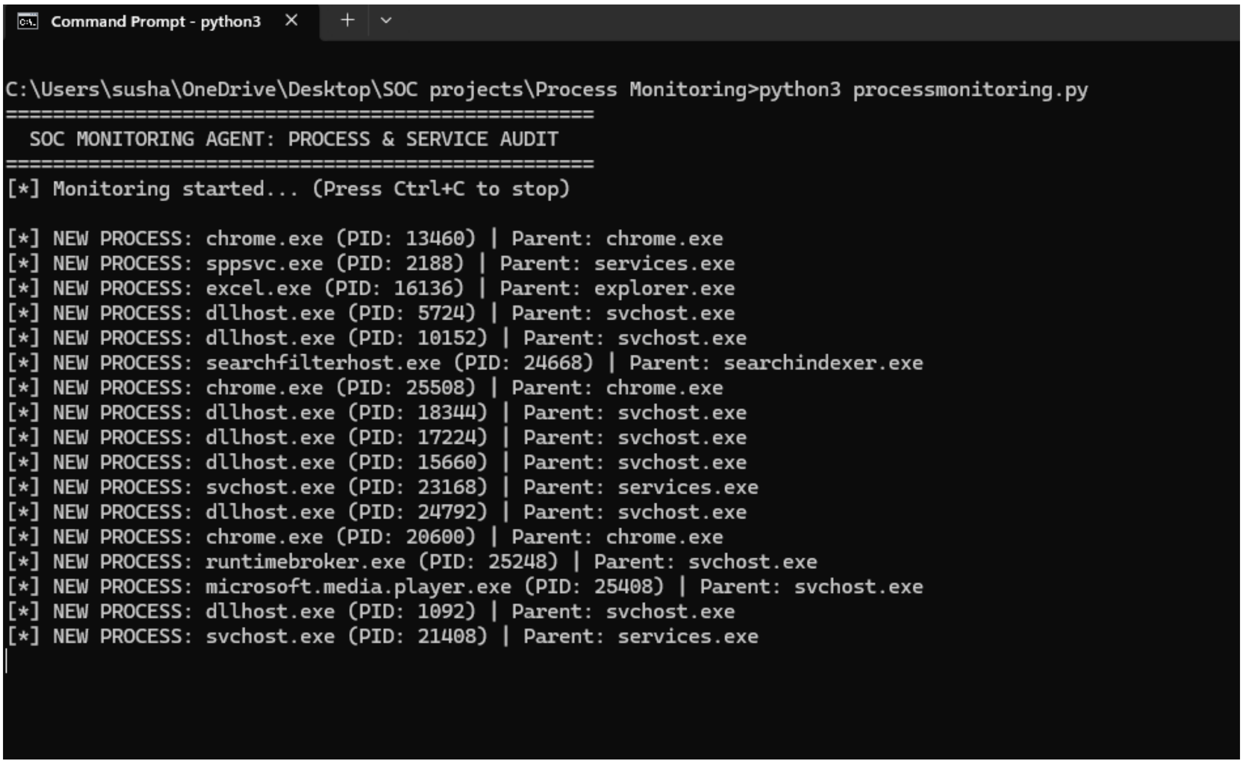
Windows Process & Service Monitoring

Code Screenshot



```python
import psutil
import time
import os

# 1. Suspicious Rules
SUSPICIOUS_PARENTS = ['winword.exe', 'excel.exe', 'powerpnt.exe', 'outlook.exe']
SUSPICIOUS_CHILDREN = ['cmd.exe', 'powershell.exe', 'wmic.exe', 'scrcons.exe']
TEMP_PATHS = ['\\temp\\', '\\tmp\\', '\\appdata\\local\\temp']

def monitor_agent():
    print("="*50)
    print("  SOC MONITORING AGENT: PROCESS & SERVICE AUDIT  ")
    print("="*50)
    print("[*] Monitoring started... (Press Ctrl+C to stop)\n")

    # processes list
    observed_pids = set()
    for p in psutil.process_iter():
        observed_pids.add(p.pid)

    try:
        while True:
            for proc in psutil.process_iter(['pid', 'ppid', 'name', 'exe']):
                try:
                    pid = proc.info['pid']
                    if pid not in observed_pids:
                        # New Process found
                        name = proc.info['name'].lower()
                        ppid = proc.info['ppid']
                        exe_path = proc.info['exe'].lower() if proc.info['exe'] else "Unknown"

                        # Parent details
                        parent_name = psutil.Process(ppid).name().lower() if psutil.pid_exists(ppid) else "N/A"

                        print(f"[*] NEW PROCESS: {name} (PID: {pid}) | Parent: {parent_name}")

                        # A. Parent-Child Anomaly Detection
                        if parent_name in SUSPICIOUS_PARENTS and name in SUSPICIOUS_CHILDREN:
                            print(f"    [!!!] ALERT: Suspicious Relationship! {parent_name} spawned {name}")

                        # B. Unauthorized Path Detection
                        if any(folder in exe_path for folder in TEMP_PATHS):
                            print(f"    [!] WARNING: Process running from suspicious directory: {exe_path}")

                        observed_pids.add(pid)
                except (psutil.NoSuchProcess, psutil.AccessDenied):
                    continue

            time.sleep(1)

    except KeyboardInterrupt:
        print("\n[!] Monitoring stopped by user.")

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

Output:



```
C:\Users\susha\OneDrive\Desktop\SOC projects\Process Monitoring>python3 processmonitoring.py
==================================================
  SOC MONITORING AGENT: PROCESS & SERVICE AUDIT
==================================================
[*] Monitoring started... (Press Ctrl+C to stop)

[*] NEW PROCESS: chrome.exe (PID: 13460) | Parent: chrome.exe
[*] NEW PROCESS: sppsvc.exe (PID: 2188) | Parent: services.exe
[*] NEW PROCESS: excel.exe (PID: 16136) | Parent: explorer.exe
[*] NEW PROCESS: dllhost.exe (PID: 5724) | Parent: svchost.exe
[*] NEW PROCESS: dllhost.exe (PID: 10152) | Parent: svchost.exe
[*] NEW PROCESS: searchfilterhost.exe (PID: 24668) | Parent: searchindexer.exe
[*] NEW PROCESS: chrome.exe (PID: 25508) | Parent: chrome.exe
[*] NEW PROCESS: dllhost.exe (PID: 18344) | Parent: svchost.exe
[*] NEW PROCESS: dllhost.exe (PID: 17224) | Parent: svchost.exe
[*] NEW PROCESS: dllhost.exe (PID: 15660) | Parent: svchost.exe
[*] NEW PROCESS: svchost.exe (PID: 23168) | Parent: services.exe
[*] NEW PROCESS: dllhost.exe (PID: 24792) | Parent: svchost.exe
[*] NEW PROCESS: chrome.exe (PID: 20600) | Parent: chrome.exe
[*] NEW PROCESS: runtimebroker.exe (PID: 25248) | Parent: svchost.exe
[*] NEW PROCESS: microsoft.media.player.exe (PID: 25408) | Parent: svchost.exe
[*] NEW PROCESS: dllhost.exe (PID: 1092) | Parent: svchost.exe
[*] NEW PROCESS: svchost.exe (PID: 21408) | Parent: services.exe
```

Project Report: Windows Process & Service Monitoring Agent

1. Project Overview: Its primary goal is to detect malicious behaviours, such as unauthorized process execution and suspicious parent-child relationships, which are common signs of a cyberattack.

2. Key Technical Features

- **Real-time Process Monitoring**: Continuously tracks every new process using PIDs (Process IDs) and Parent PIDs (PPIDs).
- **Behavioural Analysis**: Analyses the "lineage" of a process to see if a legitimate app (like Word) is starting a dangerous tool (like PowerShell).
- **Unauthorized Path Detection**: Flags any process running from high-risk directories like `\Temp\` or user-writable folders.
- **Persistence Auditing**: Monitors the system for new or modified startup services that malware uses to stay on a system.

3. Security Logic (Rule-Based Detection)

The agent uses a predefined security baseline to trigger alerts:

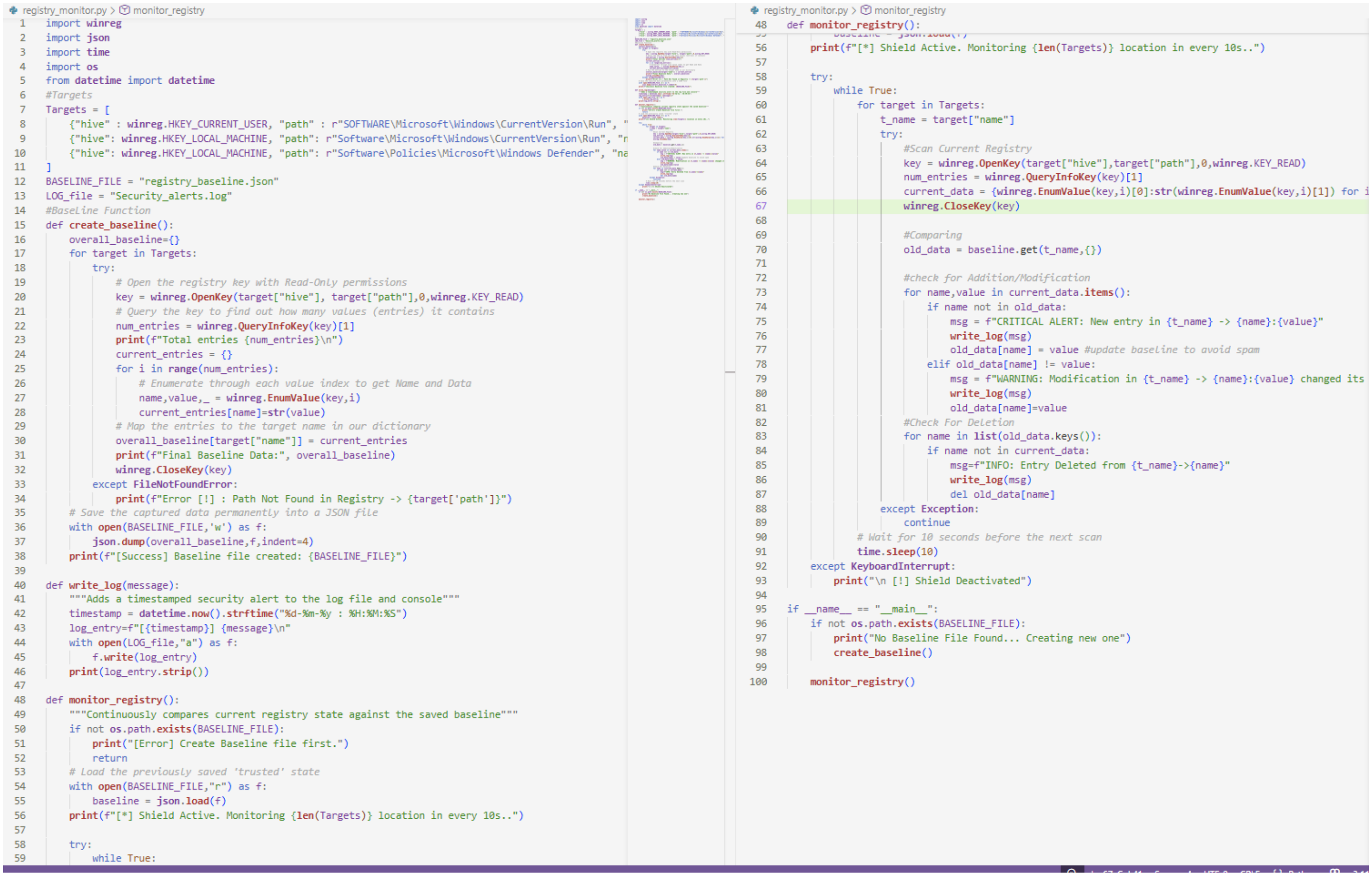
- **Suspicious Parents**: Watches Office applications (winword.exe, excel.exe).
- **Suspicious Children**: Detects unauthorized shells (cmd.exe, powershell.exe).
- **Path Rules**: Scans for execution in temporary directories.

4. SOC Value & Outcomes

- **Detection**: Identifies malware activity, privilege escalation, and intrusion attempts.
- **Visibility**: Provides a timestamped audit log of all suspicious events for incident response.
- **Prevention**: Helps strengthen the system security baseline by identifying unauthorized software.

Windows Registry Change Monitoring System

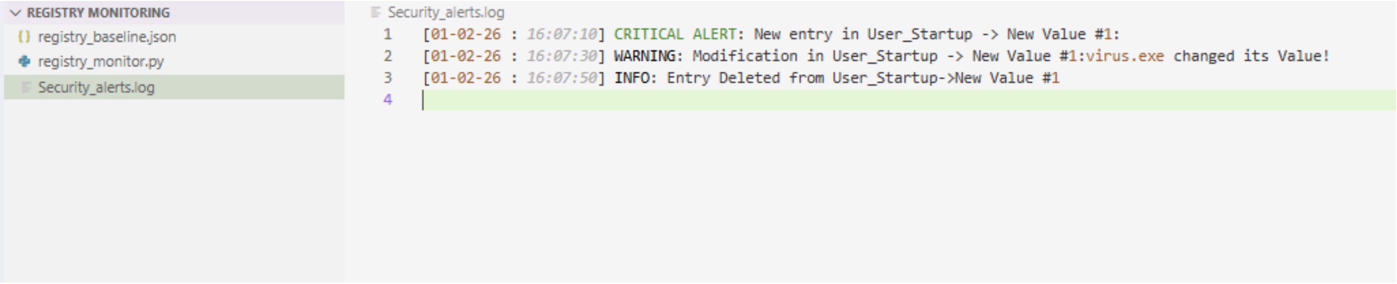
CODE Screenshot



```python
import winreg
import json
import time
import os
from datetime import datetime
#Targets
Targets = [
    {"hive" : winreg.HKEY_CURRENT_USER, "path" : r"SOFTWARE\Microsoft\Windows\CurrentVersion\Run", "
    {"hive": winreg.HKEY_LOCAL_MACHINE, "path": r"Software\Microsoft\Windows\CurrentVersion\Run", "r
    {"hive": winreg.HKEY_LOCAL_MACHINE, "path": r"Software\Policies\Microsoft\Windows Defender", "na
]
BASELINE_FILE = "registry_baseline.json"
LOG_file = "Security_alerts.log"
#BaseLine Function
def create_baseline():
    overall_baseline={}
    for target in Targets:
        try:
            # Open the registry key with Read-Only permissions
            key = winreg.OpenKey(target["hive"], target["path"],0,winreg.KEY_READ)
            # Query the key to find out how many values (entries) it contains
            num_entries = winreg.QueryInfoKey(key)[1]
            print(f"Total entries {num_entries}\n")
            current_entries = {}
            for i in range(num_entries):
                # Enumerate through each value index to get Name and Data
                name,value,_ = winreg.EnumValue(key,i)
                current_entries[name]=str(value)
            # Map the entries to the target name in our dictionary
            overall_baseline[target["name"]] = current_entries
            print(f"Final Baseline Data:", overall_baseline)
            winreg.CloseKey(key)
        except FileNotFoundError:
            print(f"Error [!] : Path Not Found in Registry -> {target['path']}")
    # Save the captured data permanently into a JSON file
    with open(BASELINE_FILE,'w') as f:
        json.dump(overall_baseline,f,indent=4)
    print(f"[Success] Baseline file created: {BASELINE_FILE}")

def write_log(message):
    """Adds a timestamped security alert to the log file and console"""
    timestamp = datetime.now().strftime("%d-%m-%y : %H:%M:%S")
    log_entry=f"[{timestamp}] {message}\n"
    with open(LOG_file,"a") as f:
        f.write(log_entry)
    print(log_entry.strip())

def monitor_registry():
    """Continuously compares current registry state against the saved baseline"""
    if not os.path.exists(BASELINE_FILE):
        print("[Error] Create Baseline file first.")
        return
    # Load the previously saved 'trusted' state
    with open(BASELINE_FILE,"r") as f:
        baseline = json.load(f)
    print(f"[*] Shield Active. Monitoring {len(Targets)} location in every 10s..")

    try:
        while True:
```

```python
def monitor_registry():
    print(f"[*] Shield Active. Monitoring {len(Targets)} location in every 10s..")

    try:
        while True:
            for target in Targets:
                t_name = target["name"]
                try:
                    #Scan Current Registry
                    key = winreg.OpenKey(target["hive"],target["path"],0,winreg.KEY_READ)
                    num_entries = winreg.QueryInfoKey(key)[1]
                    current_data = {winreg.EnumValue(key,i)[0]:str(winreg.EnumValue(key,i)[1]) for i
                    winreg.CloseKey(key)

                    #Comparing
                    old_data = baseline.get(t_name,{})

                    #check for Addition/Modification
                    for name,value in current_data.items():
                        if name not in old_data:
                            msg = f"CRITICAL ALERT: New entry in {t_name} -> {name}:{value}"
                            write_log(msg)
                            old_data[name] = value #update baseline to avoid spam
                        elif old_data[name] != value:
                            msg = f"WARNING: Modification in {t_name} -> {name}:{value} changed its
                            write_log(msg)
                            old_data[name]=value
                    #Check For Deletion
                    for name in list(old_data.keys()):
                        if name not in current_data:
                            msg=f"INFO: Entry Deleted from {t_name}->{name}"
                            write_log(msg)
                            del old_data[name]
                except Exception:
                    continue
            # Wait for 10 seconds before the next scan
            time.sleep(10)
    except KeyboardInterrupt:
        print("\n [!] Shield Deactivated")

if __name__ == "__main__":
    if not os.path.exists(BASELINE_FILE):
        print("No Baseline File Found... Creating new one")
        create_baseline()

    monitor_registry()
```

OUTPUT



```
[01-02-26 : 16:07:10] CRITICAL ALERT: New entry in User_Startup -> New Value #1:
[01-02-26 : 16:07:30] WARNING: Modification in User_Startup -> New Value #1:virus.exe changed its Value!
[01-02-26 : 16:07:50] INFO: Entry Deleted from User_Startup->New Value #1
```

Project Report: Windows Registry Change Monitoring System

1. Project Overview

This project is a Python-based security tool designed to monitor critical Windows Registry keys. It detects unauthorized changes, such as new startup entries or modified security policies, which are common indicators of malware persistence and system tampering.

2. Technical Workflow

The system operates in three distinct phases to ensure continuous protection:

- **Baseline Creation (JSON File)**: Upon the first run, the tool scans the target registry keys and saves their current "trusted" state into a file named `registry_baseline.json`. This serves as the golden standard for future comparisons.
- **Real-Time Monitoring**: The agent continuously rescans the registry every 10 seconds, comparing the live data against the saved JSON baseline.

- **Alert Generation (Log File)**: Any detected discrepancy (Addition, Modification, or Deletion) is immediately recorded in a file named `Security_alerts.log`. Each entry is timestamped for forensic auditing.

3. Key Detection Features

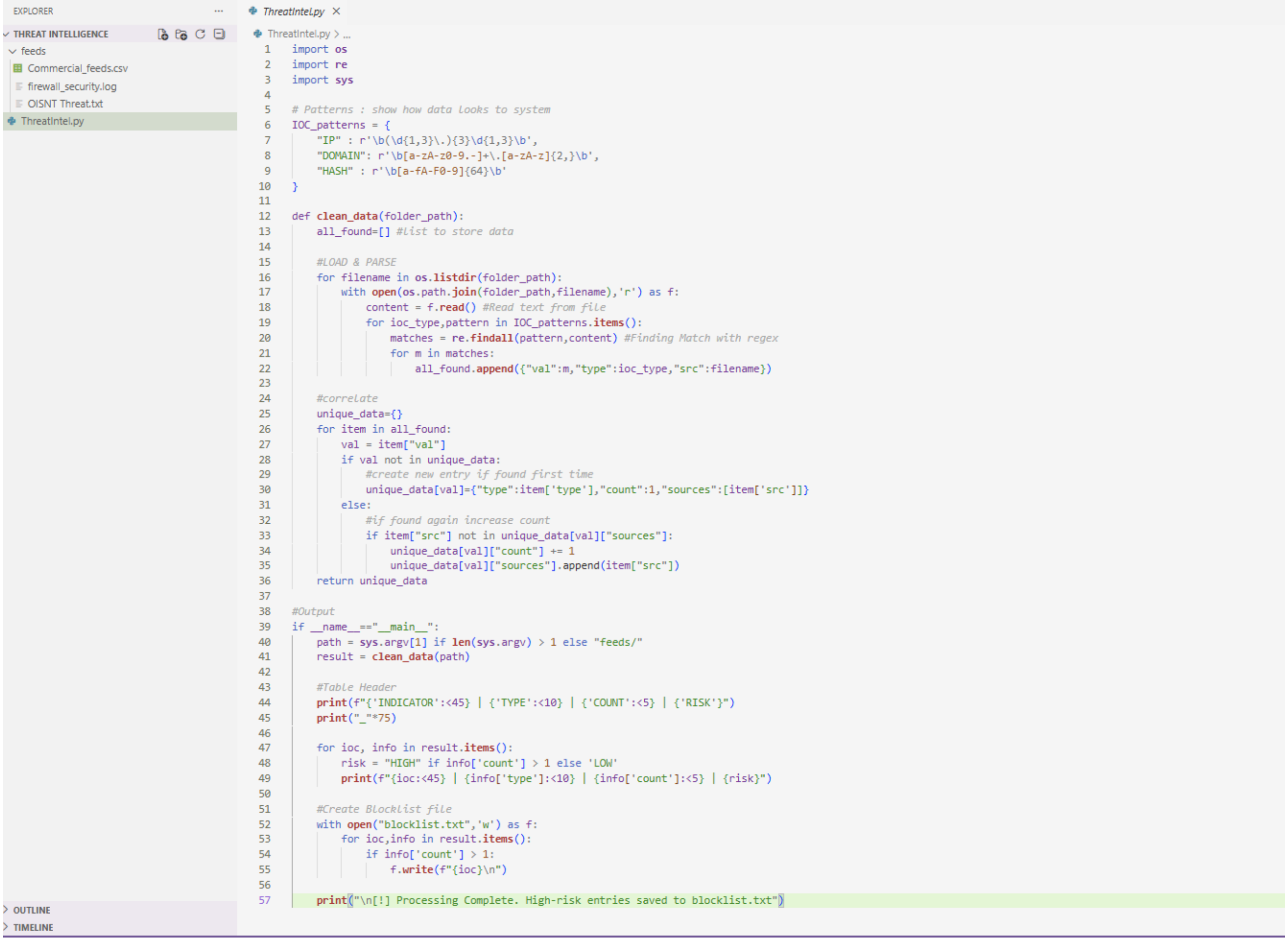
- **Critical Alerts**: Triggered when a new entry is added (e.g., malware adding itself to the "Run" key).
- **Warning Notifications**: Triggered when an existing registry value is changed.
- **Deletion Tracking**: Logs when a registry entry is removed, providing full visibility into system changes.

4. SOC Value

- **Persistence Detection**: Identifies malware attempting to survive a system reboot.
- **Integrity Monitoring**: Ensures security policies (like Windows Defender settings) are not being disabled.
- **Automated Logging**: Provides SOC analysts with a structured log of unauthorized changes for faster incident response.

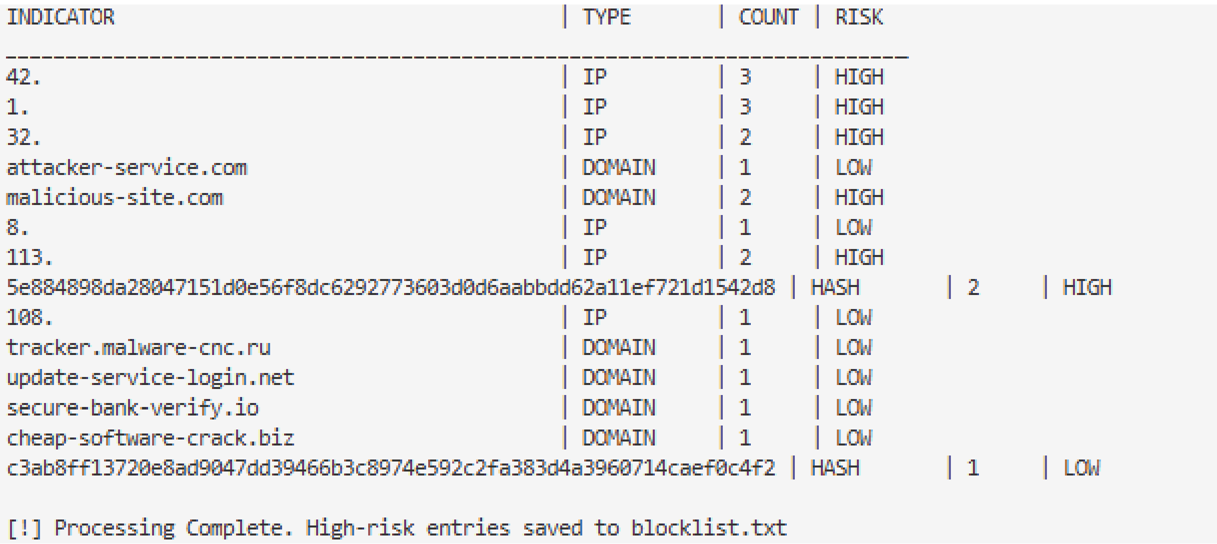
Threat Intelligence Aggregator

CODE

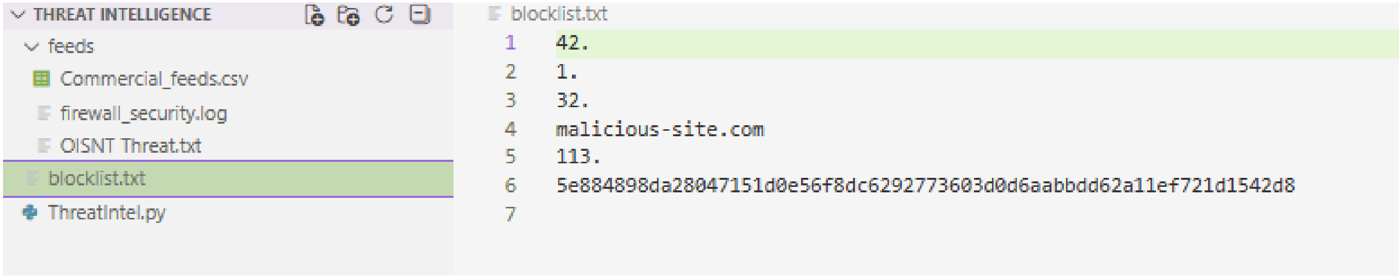


```python
import os
import re
import sys

# Patterns : show how data Looks to system
IOC_patterns = {
    "IP" : r'\b(\d{1,3}\.){3}\d{1,3}\b',
    "DOMAIN": r'\b[a-zA-z0-9.-]+\.[a-zA-z]{2,}\b',
    "HASH" : r'\b[a-fA-F0-9]{64}\b'
}

def clean_data(folder_path):
    all_found=[] #List to store data

    #LOAD & PARSE
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path,filename),'r') as f:
            content = f.read() #Read text from file
            for ioc_type,pattern in IOC_patterns.items():
                matches = re.findall(pattern,content) #Finding Match with regex
                for m in matches:
                    all_found.append({"val":m,"type":ioc_type,"src":filename})

    #correlate
    unique_data={}
    for item in all_found:
        val = item["val"]
        if val not in unique_data:
            #create new entry if found first time
            unique_data[val]={"type":item['type'],"count":1,"sources":[item['src']]}
        else:
            #if found again increase count
            if item["src"] not in unique_data[val]["sources"]:
                unique_data[val]["count"] += 1
                unique_data[val]["sources"].append(item["src"])
    return unique_data

#Output
if __name__=="__main__":
    path = sys.argv[1] if len(sys.argv) > 1 else "feeds/"
    result = clean_data(path)

    #Table Header
    print(f"{'INDICATOR':<45} | {'TYPE':<10} | {'COUNT':<5} | {'RISK'}")
    print("_"*75)

    for ioc, info in result.items():
        risk = "HIGH" if info['count'] > 1 else 'LOW'
        print(f"{ioc:<45} | {info['type']:<10} | {info['count']:<5} | {risk}")

    #Create Blocklist file
    with open("blocklist.txt",'w') as f:
        for ioc,info in result.items():
            if info['count'] > 1:
                f.write(f"{ioc}\n")

    print("\n[!] Processing Complete. High-risk entries saved to blocklist.txt")
```

OUTPUT



```
INDICATOR                                     | TYPE       | COUNT | RISK
___________________________________________________________________________
42.                                           | IP         | 3     | HIGH
1.                                            | IP         | 3     | HIGH
32.                                           | IP         | 2     | HIGH
attacker-service.com                          | DOMAIN     | 1     | LOW
malicious-site.com                            | DOMAIN     | 2     | HIGH
8.                                            | IP         | 1     | LOW
113.                                          | IP         | 2     | HIGH
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8 | HASH       | 2     | HIGH
108.                                          | IP         | 1     | LOW
tracker.malware-cnc.ru                        | DOMAIN     | 1     | LOW
update-service-login.net                      | DOMAIN     | 1     | LOW
secure-bank-verify.io                         | DOMAIN     | 1     | LOW
cheap-software-crack.biz                      | DOMAIN     | 1     | LOW
c3ab8ff13720e8ad9047dd39466b3c8974e592c2fa383d4a3960714caef0c4f2 | HASH       | 1     | LOW

[!] Processing Complete. High-risk entries saved to blocklist.txt
```



```
42.
1.
32.
malicious-site.com
113.
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8

```

SushantGavit

Project Report: Threat Intelligence Aggregator & Correlator

1. Project Overview

This project is a Python-based tool that automatically collects and analyzes **Indicators of Compromise (IOCs)** from multiple threat feeds. It converts raw, messy text into a clean, actionable **Blocklist** for SOC teams.

2. Technical Highlights (Code Logic)

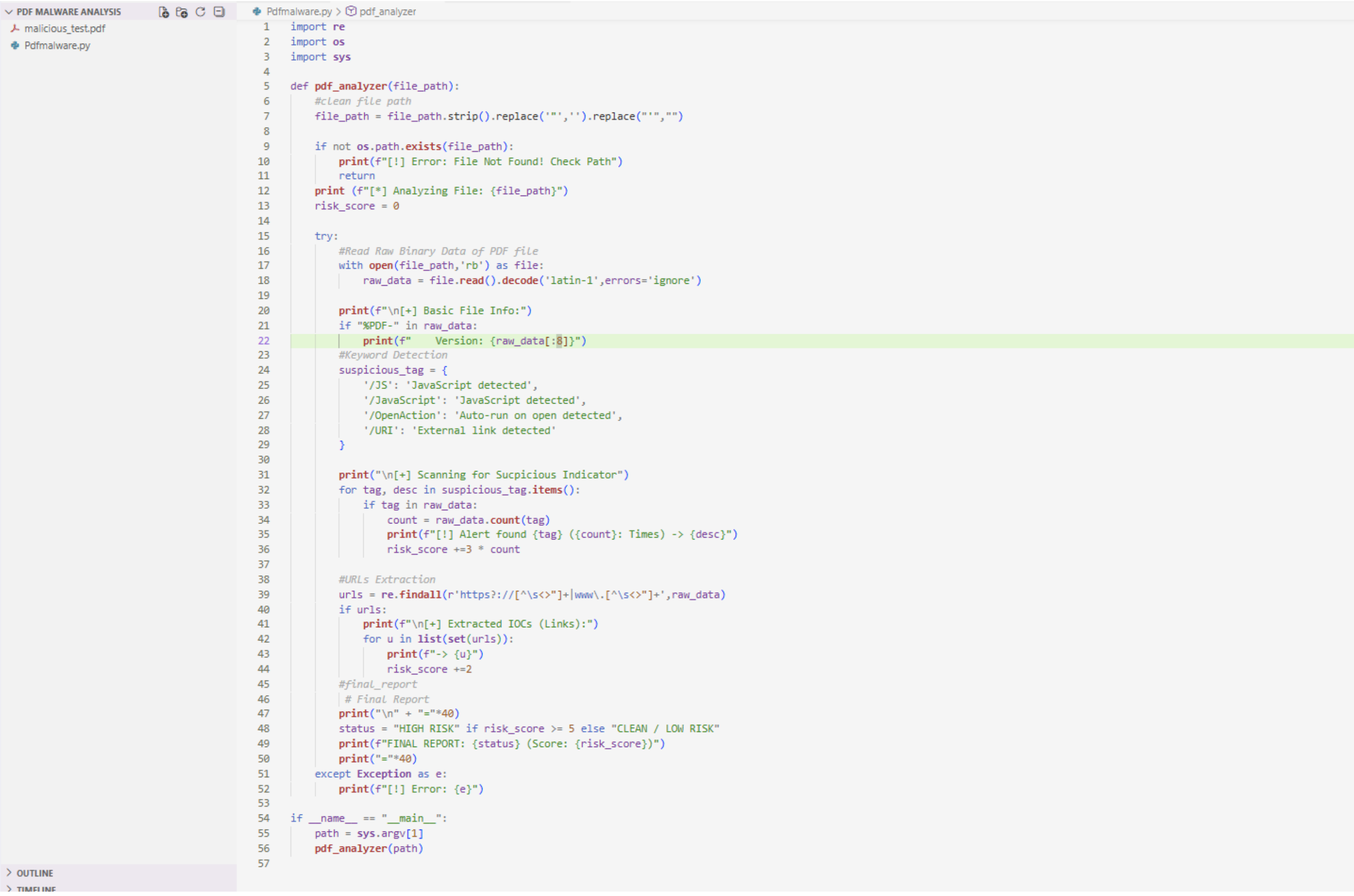
- **Pattern Recognition**: I used a dictionary called `IOC_patterns` with `Regex` **(Regular Expressions)**. This allows the system to identify exactly what an IP Address, Domain, or SHA-256 Hash looks like within thousands of lines of text.
- **Automated Extraction**: The script uses `os.listdir` to scan every file in the 'feeds' folder and applies `re.findall` to pull out every matching indicator instantly.
- **Intelligence Correlation**: This is the smartest part of the code. It uses a dictionary (`unique_data`) to track how many different sources mention the same threat (`info['count']`).
- **Risk Scoring**: I implemented a logic where if an indicator appears in more than one feed (`count > 1`), it is automatically tagged as `HIGH RISK`.

3. Key Outputs

- **Interactive Table**: Displays a clean summary in the terminal showing the Indicator, Type, Count, and Risk level.
- **Master Blocklist**: The code generates a `blocklist.txt` file. Only "Verified High-Risk" entries are saved here, which helps prevent **False Positives** when updating firewalls.

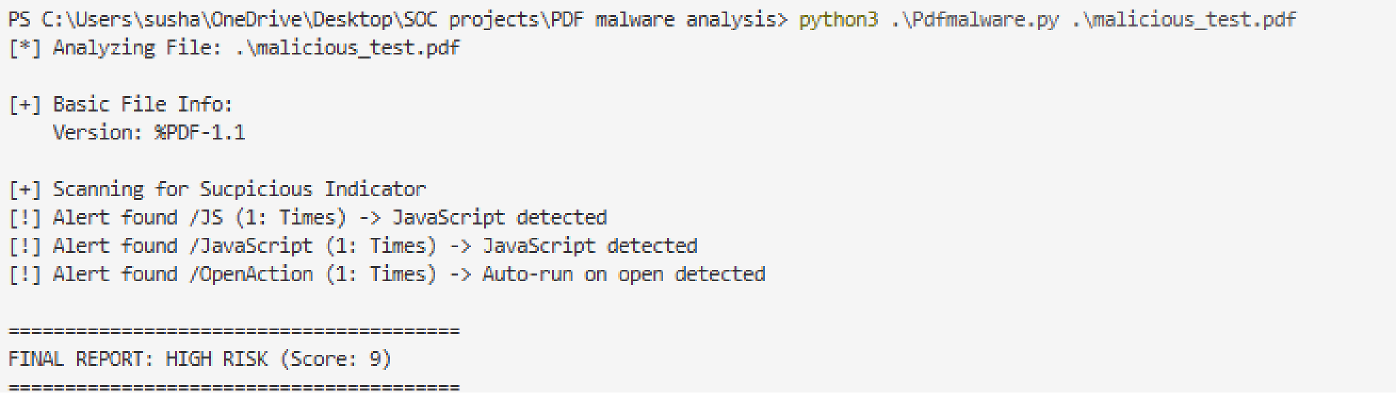
PDF Malware Analyzer

CODE SCREENSHOT



```python
import re
import os
import sys

def pdf_analyzer(file_path):
    #clean file path
    file_path = file_path.strip().replace('"','').replace("'","")

    if not os.path.exists(file_path):
        print(f"[!] Error: File Not Found! Check Path")
        return
    print (f"[*] Analyzing File: {file_path}")
    risk_score = 0

    try:
        #Read Raw Binary Data of PDF file
        with open(file_path,'rb') as file:
            raw_data = file.read().decode('latin-1',errors='ignore')

        print(f"\n[+] Basic File Info:")
        if "%PDF-" in raw_data:
            print(f"    Version: {raw_data[:8]}")
        #Keyword Detection
        suspicious_tag = {
            '/JS': 'JavaScript detected',
            '/JavaScript': 'JavaScript detected',
            '/OpenAction': 'Auto-run on open detected',
            '/URI': 'External link detected'
        }

        print("\n[+] Scanning for Sucpicious Indicator")
        for tag, desc in suspicious_tag.items():
            if tag in raw_data:
                count = raw_data.count(tag)
                print(f"[!] Alert found {tag} ({count}: Times) -> {desc}")
                risk_score +=3 * count

        #URLs Extraction
        urls = re.findall(r'https?://[^\s<>"]+|www\.[^\s<>"]+',raw_data)
        if urls:
            print(f"\n[+] Extracted IOCs (Links):")
            for u in list(set(urls)):
                print(f"-> {u}")
                risk_score +=2
        #final_report
         # Final Report
        print("\n" + "="*40)
        status = "HIGH RISK" if risk_score >= 5 else "CLEAN / LOW RISK"
        print(f"FINAL REPORT: {status} (Score: {risk_score})")
        print("="*40)
    except Exception as e:
        print(f"[!] Error: {e}")

if __name__ == "__main__":
    path = sys.argv[1]
    pdf_analyzer(path)
```

OUTPUT



```
PS C:\Users\susha\OneDrive\Desktop\SOC projects\PDF malware analysis> python3 .\Pdfmalware.py .\malicious_test.pdf
[*] Analyzing File: .\malicious_test.pdf

[+] Basic File Info:
    Version: %PDF-1.1

[+] Scanning for Sucpicious Indicator
[!] Alert found /JS (1: Times) -> JavaScript detected
[!] Alert found /JavaScript (1: Times) -> JavaScript detected
[!] Alert found /OpenAction (1: Times) -> Auto-run on open detected

========================================
FINAL REPORT: HIGH RISK (Score: 9)
========================================
```

Project Report: PDF Malware Static Analyzer

1. Project Overview

This is a lightweight Python tool designed for the **Static Analysis** of PDF files. It scans the internal structure of a PDF to find "Hidden Threats" like malicious scripts or phishing links without actually opening the file.

2. Technical Highlights (Code Keywords)

- **`raw_data` & `latin-1`**: The code reads the PDF as **Raw Binary** data using `latin-1` encoding. This ensures the tool never crashes, even if the PDF is corrupted or created by Microsoft Word.
- **`suspicious_tag` Scan**: The script searches for specific keywords like **`/JS`**, **`/JavaScript`**, and **`/OpenAction`**. If these are found, it alerts the user that the PDF might contain a hidden script that runs automatically.

- **`re.findall` (URL Extraction)**: I used **Regex** to extract every website link (**URLs**) hidden inside the PDF. In the SOC world, these are called **IOCs** (Indicators of Compromise).
- **`risk_score`**: The tool calculates a threat level. If the score is **`>= 5`**, the final result is marked as **`HIGH RISK`**, telling the analyst to block the file.

3. SOC Value

- **Safety**: It identifies threats without executing the file, preventing system infection.
- **Speed**: It provides a security verdict in seconds.
- **Automation**: It helps SOC teams quickly scan bulk PDF attachments from suspicious emails.