Intelligent Task Manager

Introduction

The **Intelligent Task Manager** is an advanced monitoring and security tool designed to enhance system integrity and protect user privacy. It integrates multiple functionalities to provide comprehensive oversight of system activities. By extracting data from the task manager, it analyzes active processes, their resource usage, and associated files. It performs security scans on all source and destination IP addresses, as well as executing files, using VirusTotal, while leveraging Metadefender for files larger than 32MB. The tool monitors network traffic in real time, mapping data transfers with the respective process names and IDs, enabling detection of unusual patterns or unauthorized activities. Additional features include a keylogger for tracking input activities, monitoring USB and physical port access to prevent unauthorized data exfiltration, and logging camera and microphone usage to detect privacy breaches. This tool ensures real-time threat detection, enhanced system security, and operational transparency, making it valuable for corporate IT monitoring, cybersecurity operations, forensic investigations, and personal privacy protection.

Methodology

1. Data extraction from task manager

The primary aim of this system is to monitor active processes on a Windows machine, collect detailed runtime and attribute information, and track process-specific privileges. The collected data is structured and stored in a CSV file for analysis, facilitating better understanding and management of system behavior.

Key Functionalities

1. Process Monitoring

The system leverages the `psutil` library to gather information about active processes in real-time.

Attributes Monitored

- **Basic Details**:
  - Process ID (PID)
  - Process Name
  - Executable Path
  - Parent Process ID and Name
- **Performance Metrics**:
  - CPU Usage Percentage
  - Memory Usage Percentage
  - Disk I/O Counters

- **Other Details**:
    - Process Status (e.g., running, sleeping)
    - Username of the process owner
    - Runtime Duration
    - Priority Class and its descriptive name

2. Privilege Tracking

Using Windows API calls through the `ctypes` library, the system retrieves detailed information about the privileges assigned to each process.

- Tracks both **enabled** and **disabled** privileges for the processes.
- Handles restricted access scenarios by bypassing or skipping inaccessible processes without interruption.

3. Lifecycle Management

The system maintains a lifecycle dictionary to track:

- **Start Time**: When the process is first detected.
- **End Time**: When the process terminates.

This enables identification of both long-running and short-lived processes.

4. Data Storage

The collected process information is saved in a structured format within a CSV file.

Headers Included:

- PID, Process Name, Executable Path, CPU and Memory Usage, I/O Counters, Privileges, Parent Process Details, Runtime, Start Time, and End Time.

Data is preprocessed to include human-readable timestamps for process creation and termination times, as well as privilege descriptions.

5. Background Execution

The system runs as a background task using threading, allowing it to collect data over a specified duration (default: 180 minutes) without interrupting other tasks on the machine.

Workflow

1. Initialization:

- CPU usage monitoring is initialized to ensure accurate measurements.
- A dictionary is created to maintain active processes and their lifecycles.

2. Data Collection:

- The system periodically (every 5 seconds) checks all active processes:
  - Attributes and performance metrics are recorded.
  - Privileges are fetched using Windows API calls.
- Newly started processes are added to the lifecycle dictionary, while terminated processes are updated with their end times.

3. Data Processing:

- Data is formatted into a CSV-compatible structure, converting timestamps and privilege data into human-readable formats.

4. Data Storage:

- All collected and processed information is saved into a CSV file for further analysis.

Outputs

- A CSV file containing detailed data about all processes, including their runtime, performance metrics, and privileges.
- Structured fields ensure compatibility with data analysis tools like Excel, Python, or business intelligence platforms.

Challenges and Resolutions

1. Access Denials

- Certain system processes restrict access to their details.
- **Resolution**: The system skips such processes gracefully and ensures proper closure of handles to avoid resource leaks.

2. System Performance Impact

- Continuous data collection over an extended duration can strain system resources.
- **Resolution**: The monitoring interval is optimized with a 5-second delay, balancing performance and monitoring accuracy.

3. Windows-Only Functionality

- The use of Windows API for privilege tracking limits the system to Windows platforms.
- **Resolution**: Future development could focus on abstracting platform-specific logic for cross-platform support.

Potential Improvements

1. Logging

- Implementing a logging mechanism for better tracking of inaccessible processes and system events.

2. Real-Time Alerts

- Adding alerts for high CPU or memory usage processes to identify potential issues proactively.

3. Visualization

- Developing a dashboard to visualize process performance metrics and privileges in real time using tools like Dash or Matplotlib.

4. Cross-Platform Compatibility

- Abstracting platform-dependent functions to support Linux and macOS alongside Windows.

2. VirusTotal Scanning and Network Monitoring

1. IP Address Scanning and Network Monitoring

- **Objective:**
  - Monitor network activity and applications to identify and scan IP addresses for potential threats using VirusTotal API.
  - Concurrently scan IP addresses and monitor network traffic and app connections.
- **Key Components:**
  - `scan_ip_with_virustotal()`: Function to scan IP addresses using the VirusTotal API. The script rotates API keys to avoid hitting rate limits.
  - `monitor_network()`: Monitors network traffic by capturing packets and extracting source and destination IPs for scanning.

    - `monitor_apps()`: Monitors running applications and their connections to external IPs, then scans these IPs for threats.
    - **Concurrent Scanning**: Using `ThreadPoolExecutor`, multiple IPs are scanned concurrently by rotating through available API keys.
- **Concurrency and Locking Mechanisms:**
    - `scanned_ips_lock`: Ensures thread-safe management of the scanned IPs.
    - `ThreadPoolExecutor`: Manages concurrent execution of IP scanning tasks.
    - **API Key Management**: The script rotates between multiple VirusTotal API keys to distribute requests evenly.
- **Strengths:**
    - Efficient concurrent scanning.
    - Continuous monitoring of network and application-level connections.
    - Thread-safe management of scanned IPs using locks.
- **Limitations:**
    - Reliant on Wireshark for network capture, requiring proper configuration.
    - Limited to IP-based monitoring; additional context (e.g., packet details) is not examined.

2. File Scanning and Monitoring Running Applications

- **Objective:**
    - Scan running files with specific extensions (e.g., `.exe`, `.msi`, `.bat`) for malware using VirusTotal.
    - Files are uploaded to VirusTotal based on their hash (SHA-256), and the script ensures files are not re-scanned unnecessarily.
- **Key Components:**
    - `generate_sha256()`: Function to calculate the SHA-256 hash of a file.
    - `upload_to_virustotal()`: Uploads the file to VirusTotal for scanning.
    - `get_virustotal_report()`: Retrieves the VirusTotal scan report for the given file hash.
    - `scan_file()`: Manages the file scanning process, ensuring files are not re-scanned.
    - `get_running_files_with_extension_and_scan()`: Collects running files with specified extensions and scans them if under 32 MB in size.
- **Concurrency and Locking Mechanisms:**
    - `scanned_files_lock`: Ensures thread-safe management of scanned files.
    - `ThreadPoolExecutor`: Manages concurrent execution of file scanning tasks.

- **Strengths:**
  - Efficient use of SHA-256 hashing to track files and avoid redundant scans.
  - Concurrent file scanning using multiple API keys to speed up the process.
  - Scans files below 32 MB in size, which is appropriate for common executables.
- **Limitations:**
  - Only scans files with specific extensions, limiting coverage.
  - Size limitation of 32 MB could potentially miss larger files that are also important for scanning.

3. File Scanning with MetaDefender API

- **Objective:**
  - Scan running files greater than 32 MB for malware or threats using MetaDefender Cloud API.
  - Ensure files are not re-scanned by tracking previously scanned files in a local history file.
- **Key Components:**
  - **`load_scan_history()`**: Loads previously scanned files and their corresponding MetaDefender `data_id` from a local JSON file (`scan_history.json`).
  - **`save_scan_history()`**: Saves the updated scan history to the local file.
  - **`scan_file()`**: Uploads a file to MetaDefender for scanning and retrieves a `data_id`, which is used to query the scan results.
  - **`get_scan_report()`**: Queries MetaDefender's API with the `data_id` to retrieve the scan report, waiting for the scan to complete if it is still in progress.
  - **`get_large_running_files()`**: Identifies files greater than 32 MB that are currently running on the system, using `psutil` to check running processes.
- **API Key and URL Configuration:**
  - The script uses an API key (`API_KEY`) to authenticate with the MetaDefender API and a base URL (`BASE_URL`) for the API endpoint.

File Scanning Process

- **File Size Limitation:**
  - The script targets files greater than 32 MB in size, which is appropriate for scanning potentially significant executables or processes that are likely to pose a security risk.

- **Scanning Workflow:**
    - **File Identification:** The script identifies running files larger than 32 MB using `psutil` and the file size is checked.
    - **File Scanning:** For each file, the script uploads the file to MetaDefender and retrieves a `data_id`.
    - **Report Retrieval:** The script repeatedly checks the scan status using the `data_id`. If the scan is in progress, it waits 5 seconds before retrying. Once complete, the scan report is fetched and printed.
    - **Avoiding Redundant Scans:** The script maintains a history of scanned files in `scan_history.json` to avoid resubmitting files that have already been scanned.

Concurrency and Flow Management

- **File Scanning Process Flow:**
    - The script performs the scanning sequentially for each file identified. Although it does not explicitly use concurrency (such as multi-threading or multiprocessing), it could benefit from parallel execution to scan multiple files simultaneously, especially if a large number of large files are detected.
- **History Management:**
    - The `scan_history` dictionary ensures that files are only scanned once, reducing redundant API requests. This is managed by checking the file path against the history file before initiating a scan.
- **Error Handling:**
    - The script includes error handling in the form of `try-except` blocks, particularly in the `psutil` module to skip files that cannot be accessed (due to permission issues or missing processes).
    - API response errors are handled gracefully by checking the status code and printing appropriate messages.

Strengths

- **Efficient File Scanning:** The script efficiently targets running files larger than 32 MB, focusing on potentially risky executable files.
- **Local Scan History:** The script maintains a history of scanned files, avoiding redundant scans, which is an excellent feature for optimizing API usage and processing time.
- **Graceful Error Handling:** The script is well-equipped to handle file access issues (via `psutil`) and API failures (via status code checks), ensuring smoother execution.
- **Real-time Scan Monitoring:** The script actively checks the status of each file scan and waits for completion before proceeding, ensuring accurate and up-to-date results.

Limitations

- **Lack of Concurrency:** The current design scans files sequentially, which may lead to inefficiencies when dealing with a large number of files. Using multi-threading or multiprocessing could significantly speed up the scanning process.
- **File Type Limitation:** The script only scans executable files (`exe`) and does not handle other potentially dangerous file types (e.g., `.dll`, `.sys`). Extending the file type checking could improve the coverage.
- **Limited to Running Files:** The script focuses only on files that are currently running, which may miss potentially dangerous files that are not currently active but still present on the system.

Potential Improvements

- **Concurrency:** To improve performance, the script could be modified to use multi-threading or multi-processing to handle multiple file scans in parallel, especially useful when scanning multiple large files.
- **File Type Extension:** The script could be extended to check for other file types that might present security risks, such as system files or large document files.
- **Enhanced Logging:** More detailed logging could be added for better traceability of each scan process, including timestamps, file sizes, and scan progress.
- **Real-time Reporting:** Instead of just printing results to the console, the script could store scan results in a file or database for later analysis and tracking.

4. USB and Audio Device Monitoring

Overview

- **USB Device Monitoring (`monitor_usb_devices`)**:
  - The function uses `wmi.WMI()` to connect to the WMI interface, which is a Windows management system that provides detailed information about system components.
  - The function retrieves all USB controller devices via `c.Win32_USBControllerDevice()`, which returns objects representing each connected USB device.
  - For each USB device, the script prints the device name using `usb_device.Dependent.Name`.
- **Audio Device Monitoring (`monitor_audio_devices`)**:
  - Similarly, the function retrieves audio devices via `c.Win32_SoundDevice()`, which returns audio device objects.

    - For each audio device, the function prints the device's name using `audio_device.Name`.

Key Components

- **`wmi.WMI()`**:
    - This establishes a connection to the WMI service, which allows for querying system information.
    - It's crucial for retrieving detailed hardware and system status information in Windows environments.
- **`Win32_USBControllerDevice`**:
    - A WMI class that provides details about USB devices connected to the system. This script retrieves the `Dependent.Name` attribute to display the name of the device.
- **`Win32_SoundDevice`**:
    - A WMI class that provides details about sound (audio) devices. The script retrieves the `Name` attribute to display the name of each audio device.

Output

- **USB Devices**:
    - The script lists all USB devices connected to the system, including the names of USB controllers.
- **Audio Devices**:
    - It also lists all audio devices (such as speakers, microphones, and sound cards) attached to the system.

5. Network Traffic Monitoring

Overview

The network monitoring component is designed to track **active network connections** of processes and gather their associated details. It provides insights into how applications interact with the network and tracks the data flow (bytes sent and received).

Key Features

1. **Connection Data Retrieval**
    - The code uses `psutil.net_connections(kind='inet')` to list active internet connections, capturing both IPv4 and IPv6 connections.
    - Each connection is associated with a process ID (`pid`) and contains:
        - **Source IP** and **Source Port**
        - **Destination IP** and **Destination Port**

        - **Connection Status** (e.g., ESTABLISHED, LISTENING, TIME_WAIT)
2. **Process-Network Relationship**
    - Each connection is linked to its parent process using the `pid` attribute.
    - This allows for correlating the network activity with process-level information such as:
        - Process name, memory usage, CPU usage, and runtime.
        - Privileges of the process accessing the network.
3. **Data Flow Metrics**
    - Bytes sent and received are tracked using `psutil.Process.io_counters` for processes with associated network activity:
        - **Bytes Sent**: Total data sent by the process.
        - **Bytes Received**: Total data received by the process.
4. **Integration with Process Data (Mapping)**
    - Network data is combined with detailed process information, ensuring a comprehensive view of system activities:
        - Each network connection entry includes:
            - Source/Destination IP and Port
            - Connection Status
            - Process-level metrics like memory percent, CPU percent, and privileges.

Strengths

1. **Granular Data Collection**
    - Collects both network and process-specific details for fine-grained monitoring.
2. **Real-Time Monitoring**
    - Designed to fetch updated connection and data flow metrics periodically using the `time.sleep()` function.
3. **Process Awareness**
    - Links network activity to specific processes, enabling a clear understanding of which application is using the network.
4. **Bytes Sent/Received**
    - Includes total bytes sent and received, giving visibility into data usage by individual processes.

Limitations

1. **Limited Connection Types**
    - Focuses only on `inet` connections (IPv4/IPv6). Other types like `inet6` or raw sockets are not covered.
2. **Access Restrictions**
    - For some processes, `psutil.AccessDenied` or `psutil.NoSuchProcess` exceptions may prevent full data collection.
3. **Performance Overhead**
    - Collecting detailed network and process metrics for extended periods may impact system performance, especially with frequent updates.

Recommendations for Improvement

1. **Expand Connection Types**
    - Include other connection types such as `unix` or raw sockets to widen the scope.
2. **Error Logging**
    - Implement robust error handling and logging for failed data retrieval attempts to improve reliability.
3. **Historical Tracking**
    - Log changes in bytes sent/received over time to analyze trends in network activity.
4. **Data Compression**
    - Compress large volumes of collected data before saving, especially if monitoring for long durations.

Overview of Network Traffic Monitor using PyShark

This script uses PyShark to monitor network traffic in real-time. It captures packets transmitted between devices on a network, extracts relevant data (such as source IP, destination IP, and packet size), and logs the data transfer statistics to a CSV file for analysis.

Key Features

1. **Real-Time Packet Capture**: Uses PyShark to continuously monitor packets on a specified network interface.
2. **Bidirectional Traffic Accounting**: Records data transfer statistics for both directions (source-to-destination and destination-to-source).
3. **CSV Logging**: Periodically saves captured traffic data to a CSV file for persistent storage and analysis.
4. **Customizable Capture Duration**: Allows users to specify how long each capture cycle should last before saving data.
5. **Automatic Data Reset**: Clears the traffic data after each save, ensuring efficient memory usage.

Dependencies

1. Python

- **Version**: Python 3.6 or higher.
- **Installation**: Download Python.

2.PyShark

- **Description**: Python wrapper for Wireshark, used for packet capturing.

Installation:
`pip install pyshark`

- **Requirements**:
  - **Wireshark and TShark**: Install from Wireshark's official website.
  - **Permissions**: Requires admin/root access to capture packets.

3.Pandas

- **Description**: Library for data manipulation, used for saving traffic data to CSV.
- **Installation**: `pip install pandas`

Usage

1.Prerequisites:
Install PyShark and Pandas:
`pip install pyshark pandas`

- Wireshark must be installed on the system for PyShark to function correctly.
- Run the script with appropriate permissions (e.g., as an administrator or with elevated privileges).

2.Running the Script:
Start the script by executing:
`python traffic_monitor.py`

- Specify the network interface to monitor (e.g. `'Wi-Fi'`)
- Captured data will be periodically saved to a CSV file (`network_traffic.csv`).

Workflow

1. **Packet Capture**:
   - Captures packets in real-time from a specified interface using `pyshark.LiveCapture`.
   - Extracts the source IP, destination IP, and packet length.
2. **Data Processing**:
   - Updates the `traffic_data` dictionary to accumulate the total data transferred between source and destination IPs.
3. **Data Logging**:
   - Periodically saves the accumulated traffic data to a CSV file every `capture_duration` seconds.
   - Resets the `traffic_data` dictionary after saving.
4. **Continuous Monitoring**:
   - Restarts the capture cycle indefinitely, allowing for continuous traffic monitoring.

Limitations

1. **Performance**:
    - High network traffic can impact performance due to the continuous packet capturing and processing.
    - Writing to the CSV file periodically can introduce minor delays.
2. **Permission Requirements**:
    - The script requires elevated permissions to access the network interface for packet capturing.
3. **Interface Compatibility**:
    - Interface names vary by operating system. The user must correctly specify the network interface.
4. **Packet Filtering**:
    - The script does not apply advanced filtering (e.g., protocols or specific ports). It captures all IP-based traffic.
5. **File Handling**:
    - Writing to the CSV may fail if the file is in use or the script lacks write permissions.

Potential Improvements

1. **Advanced Filtering**:
    - Add support for protocol-based or port-based filtering to focus on specific types of traffic.
2. **Improved CSV Management**:
    - Include headers in the CSV file if it doesn't exist or separate logs by timestamp for better organization.
3. **User Configuration**:
    - Allow users to configure parameters (e.g., interface name, capture duration, CSV file name) via command-line arguments.
4. **Real-Time Visualization**:
    - Integrate a dashboard for real-time visualization of network traffic statistics.
5. **Error Handling**:
    - Add more robust error handling for edge cases, such as invalid interfaces or sudden network interruptions.

6. Mapping the network traffic data with the data extracted from the task manager

Overview

This Python script provides comprehensive monitoring and logging of system processes and their associated network connections, capturing privileges, resource usage, and process lifecycle information. It integrates advanced features like process privilege simulation, network traffic mapping, and runtime monitoring, saving the

results into a structured CSV file. It uses `psutil`, `ctypes`, and `wmi` libraries to collect and process data.

Features

1. **Process Information Collection:**
   - Extracts process details including PID, name, executable path, CPU and memory usage, privileges, runtime, parent process details, and priority class.
   - Uses WMI and ctypes for privilege analysis, enhancing granularity.
2. **Network Traffic Monitoring:**
   - Captures source and destination IPs and ports, connection status, and correlates them with processes.
   - Includes bytes sent and received for each connection.
3. **Lifecycle Monitoring:**
   - Tracks process start and end times to detect lifecycle changes.
   - Uses `psutil` for real-time updates on active processes.
4. **Data Export:**
   - Stores all collected data in a CSV file located on the desktop for easy access and further analysis.
5. **Background Execution:**
   - Runs data collection in a separate thread to avoid blocking the main program execution.

Functions

1. **`get_process_privileges(pid)`**
   - Fetches privileges assigned to a process using Windows APIs.
     Inputs:
   - `pid` (int): Process ID.
     Returns:
   - List of privileges (`Enabled/Disabled`), or "Access Denied" if privileges are restricted.
2. **`get_network_connections()`**
   - Retrieves active network connections and maps them to processes.
     Returns:
   - List of dictionaries with connection details such as source IP/port, destination IP/port, and status.
3. **`collect_combined_info(duration_minutes=1)`**
   - Gathers process and network information for a specified duration.
     Processes:
   - Iterates through all active processes.
   - Maps network connections to processes and fetches detailed information.
   - Monitors process start and end times.
     Returns:

    - Combined process and network information along with lifecycle data.
4. **`save_to_csv(data, active_processes, filename)`**
    - Saves the collected data into a CSV file.
      Inputs:
    - `data` (list): Process and network details.
    - `active_processes` (dict): Process lifecycle data.
    - `filename` (str): Name and location of the output file.
5. **`background_collection(duration_minutes=1)`**
    - Runs data collection in the background for the specified duration.
      Outputs:
    - Saves the data to a CSV file on the desktop.
6. **`start_background_task()`**
    - Initiates the background data collection in a separate thread.

Workflow

1. **Initialization:**
    - Initializes WMI client and sets up constants for Windows privilege management.
2. **Data Collection:**
    - Iterates over running processes and captures details.
    - Maps active network connections to corresponding processes.
    - Tracks process lifecycle and privileges.
3. **Background Execution:**
    - Launches a thread to perform collection tasks without interrupting other operations.
4. **Data Export:**
    - Saves aggregated process and network data into a CSV file named `full_process_network_info.csv` on the desktop.

Dependencies

1. **Python 3.7 or later**
2. **Required libraries:**
    - `psutil`: Process and network monitoring.
    - `ctypes`: Windows privilege simulation.
    - `wmi`: Windows Management Instrumentation access.
    - `csv`, `datetime`, `os`, `threading`: Standard Python libraries for data handling and threading.

Limitations

1. **Windows-Specific:**
    - Uses WMI and ctypes, making it suitable only for Windows environments.
2. **Privilege Handling:**
    - Simulates privileges; real-time privilege changes may not be captured accurately.
3. **Performance Overhead:**
    - Real-time monitoring with frequent data collection intervals may cause higher resource usage.
4. **Error Handling:**
    - Restricted processes or inaccessible connections may result in skipped data.
5. **Static Privilege Mapping:**
    - Does not dynamically fetch new privilege types.

7. Keylogger

Technical Overview

The Keylogger script is a monitoring tool designed to capture and log keyboard activity on a system. Its primary purpose is to record keypresses in real time, associating them with the active application or window. The script ensures efficient tracking of user input for analysis while maintaining flexibility for customization and integration into broader security systems.

Detailed Functionality

Real-Time Key Logging

- Records all keyboard inputs across active applications.
- Logic: Captures every keypress event and matches it with the active application window.
- Code Explanation:
    - The `keyboard` module intercepts keypresses globally.
    - The `pygetwindow` library retrieves the title of the currently active application window.
    - Outputs the captured keypress along with the window's title for contextual logging.

```
import keyboard
import pygetwindow as gw

while True:
    if keyboard.read_event().event_type == "down":
        window_title = gw.getActiveWindow().title
```

```
key = keyboard.read_key()
print(f"{window_title}: {key}")
```

Customizable Key Exclusion

- Allows the exclusion of specific keys or key combinations from being logged.
- Use Case: Ensures privacy by excluding sensitive inputs like passwords or authentication keys.

Application Context Logging

- Logs keypresses along with the active window title to provide context.
- Purpose: Enables detailed analysis of user activity for monitoring or forensic purposes.

Data Storage and Security

- The logged data can be redirected to secure storage (e.g., files, databases) for analysis or auditing.
- Future Integration: Encrypt logs to enhance data security and ensure compliance with privacy standards.

Security and Privacy Benefits

- Comprehensive Monitoring: Provides detailed visibility into keyboard usage.
- User Behavior Analysis: Tracks keypress patterns for productivity or security analysis.
- Customizable Privacy Controls: Supports user-defined exclusions to maintain confidentiality.

Potential Enhancements

- **Anomaly Detection**: Add functionality to identify irregular typing patterns or unauthorized usage.
- **Encryption**: Store logs in an encrypted format to ensure data security.
- **Real-Time Alerts**: Trigger alerts for suspicious input patterns or unauthorized key usage.
- **Cross-Platform Support**: Extend functionality to non-Windows operating systems like macOS or Linux.

8. Camera access

Overview

The Camera Accessibility Monitoring and Control feature provides users with the ability to monitor, manage, and control camera usage on their system effectively. It

ensures privacy and security by identifying processes or browser tabs using the camera and offering options to restrict access at both software and hardware levels.

Key Functionalities

1. Real-Time Process Monitoring:
    - Detects running processes that may have access to the camera.
    - Provides detailed insights such as:
        - Process name (e.g., Chrome, Firefox).
        - Process ID (PID).
        - Memory usage.
        - Total CPU time.
    - Alerts users to processes or browser instances likely to use the camera.
2. Browser Tab Inspection:
    - Fetches and displays all active browser tabs with titles and URLs.
    - Identifies tabs likely to access the camera and provides user options to manage permissions.
3. Camera Permission Management:
    - Browser-Level Control:
        - Uses the Chrome DevTools Protocol to disable camera access for specific tabs.
        - Operates via WebSocket commands, ensuring direct and efficient communication.
    - System-Level Control (Optional):
        - Disables or enables the camera device entirely using Windows WMI.
        - Prevents all processes from accessing the camera until re-enabled.
4. Hardware Accessibility Check:
    - Monitors hardware usage through command-line arguments of processes.
    - Highlights processes explicitly requesting camera access.
5. User Interaction:
    - Interactive prompts to allow or deny camera permissions for specific tabs.
    - Optional system-wide kill switch to disable camera hardware for maximum security.

Security and Privacy Benefits

- Enhanced Awareness:
    - Provides visibility into processes and tabs that access the camera.
    - Reduces risks of unauthorized camera access.
- User Control:
    - Empower users to decide camera permissions on a case-by-case basis.
    - Offers a fallback to completely disable the camera for added safety.
- System-Wide Defense:
    - Acts as a privacy guard by locking down hardware-level access when necessary.

Integration with Other Systems

- Can be extended to include microphone access control for comprehensive monitoring.
- Works alongside system-wide privacy settings, ensuring compatibility with user-defined preferences.

Use Cases

1. **I**ndividual Users:
    - Monitor and manage personal camera usage to avoid unauthorized access.
2. Corporate Environments:
    - Implement organization-wide policies for camera and microphone access.
    - Protect sensitive data during virtual meetings.
3. Parental Controls:
    - Restrict children's access to camera-enabled applications.
4. Cybersecurity Applications:
    - Actively monitor and block potential threats exploiting camera access.

Future Enhancements

- Automatic Monitoring:
    - Real-time notifications for unauthorized camera usage.
- Reporting:
    - Generate detailed logs for review and compliance audits.
- Multi-Platform Support:
    - Extend compatibility to macOS and Linux systems.
- Enhanced Permissions:
    - Manage other hardware permissions (e.g., location, microphone) via a unified interface.

9. Microphone access

Technical Overview

The Microphone Access feature is a security-oriented tool designed to monitor and control microphone usage. Its primary goal is to prevent unauthorized access and ensure that microphone-enabled applications operate transparently. The tool provides both passive monitoring (detection and logging of microphone usage) and active control (kill switch functionality) through real-time tracking, system-level integrations, and administrative overrides.

Detailed Functionality

1. **Active Application Monitoring**
    - The module identifies active applications using the microphone by querying window titles and cross-referencing them against a whitelist of known microphone-using applications.

        - **Applications Monitored:** Zoom, Skype, Teams, Discord, WhatsApp, Instagram, browsers (e.g., Chrome, Brave, Firefox), media software (e.g., VLC, OBS Studio), and system utilities (e.g., Sound Recorder).
        - **Logic:** Matches active window titles with predefined keywords to detect potential microphone usage.
    - **Code Explanation:**
        - Utilizes pygetwindow to list active windows and filter titles containing names of microphone-using apps.
        - Tracks duration using a time-based dictionary (`start_times`) for each application.

```
active_apps = [win.title for win in gw.getWindowsWithTitle("")
if is_microphone_app(win.title)]
for app in active_apps:
    duration = time.time() - start_times.get(app, time.time())
    print(f"{app} has been using the microphone for
{duration:.2f} seconds.")
```

2. **Microphone Status Detection**
    - Directly queries the microphone's operational state using WMI (Windows Management Instrumentation).
    - **Logic:** Searches for connected devices with "microphone" in their name and checks their operational status **(`Status == "OK"`).**
    - **Use Case:** Confirms whether the microphone is physically active, regardless of associated applications.
    - **Technical Implementation:**Leverages the `Win32_PnPEntity` class in WMI to extract hardware status.

```
def get_microphone_status():
    c = wmi.WMI()
    for device in c.Win32_PnPEntity():
        if 'microphone' in str(device.Name).lower() and
device.Status == "OK":
            return True
    return False
```

3. **Event Log Analysis**
    - Analyzes Windows Security Event Logs for microphone access.
    - **Logic:** Uses PowerShell commands to extract microphone-related events from the system logs.
    - **Purpose:** Detects usage even when no known applications or windows are active.

PowerShell Command:

```
Get-WinEvent -LogName Security | Where-Object { $_.Message
-match 'microphone' }
```

4. **Integration with Python:**
    - The script executes the PowerShell command using Python's `subprocess` library and parses the output for relevant data.

```
result = subprocess.run(
    ["powershell", "-Command", "Get-WinEvent -LogName Security
| Where-Object { $_.Message -match 'microphone' }"],
    capture_output=True, text=True
)
if result.stdout:
    print("Microphone access detected in event logs.")
```

5. **Kill Switch for Microphone Access**
    - The kill switch is a GUI-based application developed using Tkinter. It provides an intuitive interface to enable or disable microphone access at the OS level.
    - **Implementation:** Modifies system registry keys to override microphone permissions.
    - **Registry Paths:**
        - `HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\CapabilityAccessManager\ConsentStore\microphone`
        - `HKCU:\Software\Microsoft\Windows\CurrentVersion\CapabilityAccessManager\ConsentStore\microphone`

PowerShell Commands:

```
# Disable microphone access
Set-ItemProperty -Path 'HKLM:\...ConsentStore\microphone'
-Name 'Value' -Value 'Deny'
Set-ItemProperty -Path 'HKCU:\...ConsentStore\microphone'
-Name 'Value' -Value 'Deny'

# Enable microphone access
Set-ItemProperty -Path 'HKLM:\...ConsentStore\microphone'
-Name 'Value' -Value 'Allow'
Set-ItemProperty -Path 'HKCU:\...ConsentStore\microphone'
-Name 'Value' -Value 'Allow'
```

6. **Python Integration:**
    - Uses `subprocess` to execute PowerShell commands securely.
    - Verifies administrative privileges using `ctypes`.

```
def disable_microphone_access():
    script = "Set-ItemProperty -Path 'HKLM:...microphone'
-Name 'Value' -Value 'Deny'"
    subprocess.run(["powershell", "-Command", script],
check=True)
    print("Microphone access disabled.")
```

7. **GUI Interface:**
    - Features buttons to enable or disable microphone access.
    - Displays status messages dynamically based on command execution results.

```
root = tk.Tk()
tk.Button(root, text="Disable Microphone Access",
command=disable_microphone_access).pack()
tk.Button(root, text="Enable Microphone Access",
command=enable_microphone_access).pack()
root.mainloop()
```

8. **Security and Privacy Benefits**

- **Real-Time Monitoring:** Ensures continuous tracking of microphone usage.
- **Unauthorized Access Detection:** Identifies unusual or unauthorized microphone usage patterns through event log analysis.
- **Immediate Control:** Allows users to instantly disable microphone access, preventing misuse.
- **System-Level Enforcement:** Registry-based implementation ensures robust protection, immune to application-level overrides.

Potential Enhancements

- **Anomaly Detection:** Integrate machine learning algorithms to identify suspicious microphone usage patterns.
- **Notification System:** Send real-time alerts to users when unauthorized access is detected.
- **Cross-Platform Functionality:** Expand support to macOS and Linux systems.
- **Data Encryption:** Log microphone usage data securely to prevent tampering.

9. Process Chain Tracking Script

1. Overview

This code is designed to monitor processes on a system and build a hierarchical tree of processes based on their parent-child relationships. It uses the `psutil` library to interact with system processes, `time` to manage sleep intervals, and `threading` to run the process monitoring asynchronously.

The `ProcessChain` class allows the user to specify processes to track and execute actions on them when detected. The processes are tracked in real-time, and a tree structure is built starting from the root processes. This is useful for monitoring the execution chain of processes for debugging, analysis, or security purposes.

2. Components and Structure

Class: `ProcessChain`

This is the central class that manages process monitoring, builds a process tree, and handles actions.

- **Attributes:**
  - `chain`: A dictionary mapping process names to actions.
  - `running`: A boolean flag to control the continuous monitoring of processes.
  - `process_tree`: A dictionary representing the hierarchy of processes.
- **Methods:**
  - `add_to_chain(process_name, action)`: Adds a process name and its corresponding action to the monitoring chain.
  - `monitor_processes()`: Continuously monitors processes, checking if any of the processes in the chain are active. When detected, it executes the associated action and builds the process tree.
  - `build_process_tree(root_process, current_processes)`: Constructs a hierarchical tree of processes starting from a specified root process.
  - `display_process_tree()`: Displays the tracked process tree with indentation for hierarchical clarity.
  - `stop()`: Stops the continuous monitoring of processes.

Example Action:

A sample action is defined as `action_for_process`, which simply prints the detected process name and PID, but this could be replaced with more complex actions such as logging, sending alerts, etc.

3. Workflow

1. **Process Detection:**
    - The `monitor_processes` method uses `psutil.process_iter()` to retrieve all running processes with their `pid`, `ppid`, and `name` attributes. It then checks if any of the processes match the names in the monitoring chain. If a match is found, the action for that process is triggered.
2. **Building the Process Tree:**
    - Once a process is detected, the `build_process_tree` method is invoked to construct a tree of processes. This method starts with the root process (the detected process) and recursively adds its child processes (those with matching `ppid`).
3. **Displaying the Tree:**
    - After monitoring and detecting processes, the `display_process_tree` method is called to print the process hierarchy. It uses recursion to display parent and child processes with appropriate indentation.
4. **Stopping the Monitoring:**
    - The `stop` method stops the continuous process monitoring, which is useful in cases where the user wants to halt monitoring manually (e.g., through a `KeyboardInterrupt`).

4. Execution Flow

Example of Adding Processes:

The code adds processes associated with Microsoft Office (`WINWORD.EXE`, `EXCEL.EXE`, etc.) to the chain for monitoring.

```
microsoft_processes = ["WINWORD.EXE", "EXCEL.EXE",
"POWERPNT.EXE", "OUTLOOK.EXE", "MSACCESS.EXE", "ONENOTE.EXE"]
for process_name in microsoft_processes:
    process_chain.add_to_chain(process_name,
action_for_process)
```

This code adds the specified Microsoft Office processes to the monitoring chain, with `action_for_process` as the action to be executed when any of them is detected.

Monitoring in a Separate Thread:

The monitoring is performed in a separate thread, enabling the main program to stay responsive or perform other tasks concurrently.

```
monitor_thread =
Thread(target=process_chain.monitor_processes)
monitor_thread.start()
```

Stopping the Monitoring:

The process monitoring runs indefinitely until interrupted. Upon receiving a `KeyboardInterrupt`, the monitoring is stopped, and the process tree is displayed.

```
try:
    while True:
        time.sleep(1)  # Keep the main thread alive
except KeyboardInterrupt:
    process_chain.stop()
    monitor_thread.join()
    process_chain.display_process_tree()
```

5. Example Output

The output of the script is twofold:

Real-time Tracking: As each process in the chain is detected, the associated action is executed, and the process chain is updated.

```
Process 'WINWORD.EXE' detected (PID: 1234). Tracking its
chain.
Action: Tracking chain for process 'WINWORD.EXE' (PID: 1234)
```

Process Tree: After the monitoring is stopped, the hierarchical process tree is displayed.

```
Process Tree:
Process 'python.exe' (PID: 5678)
    Process 'chrome.exe' (PID: 1234)
        Process 'WINWORD.EXE' (PID: 4321)
    Process 'mspmsnsv.exe' (PID: 8765)
```

6. Use Cases

- **Security Monitoring:** Track malicious or unauthorized processes by adding their names to the chain and taking actions like logging, sending alerts, or terminating the processes.
- **Performance Analysis:** Analyze how processes interact and monitor resource consumption, especially when one process spawns several children.
- **Debugging and Auditing:** Help developers understand the relationship between processes, especially when debugging complex applications that launch multiple subprocesses.

7. Improvements and Considerations

- **Error Handling:** The code currently lacks comprehensive error handling for situations where a process might terminate unexpectedly or if there are issues accessing certain processes due to permissions.
- **Optimizing for Large Systems:** On systems with many processes, checking every process every 2 seconds might be inefficient. The system could be optimized by implementing a more efficient approach for detecting changes in the process list.
- **Action Extensions:** The actions associated with processes could be expanded to perform more meaningful tasks, such as logging to a file, notifying the user, or even triggering specific scripts based on the detected process.

10.Mapping Network Traffic with Threat Analysis

Overview

This Python script provides comprehensive monitoring and analysis of network traffic to identify malicious activities. It integrates advanced features like threat detection using blocklists, Google Safe Browsing API, and custom reputation checks. The results are logged into a CSV file for further investigation. It uses libraries like Scapy, pybloom-live, and standard Python libraries for packet capture, filtering, and logging.

1.Features

Blocklist Integration:

- Downloads and processes blocklists from multiple trusted sources (e.g., FireHOL, AbuseIPDB).
- Implements a Bloom Filter for fast IP lookups and memory efficiency.

Real-Time Packet Capture:

- Captures live packets and extracts source and destination IPs using Scapy.

  Threat Analysis with Chaining Methods:

- Bloom Filter Check: Quickly verifies if IPs are in the blocklist.
- Google Safe Browsing API: Analyzes IPs for malicious URLs.
- Custom Reputation Check: Flags IPs based on hardcoded reputation rules.

  Results Logging:

- Logs IP reputation, threat analysis flags, and timestamps to a CSV file located on the desktop.
- Provides real-time console output for detected malicious IPs.

  Background Execution:

- Runs packet capture, IP extraction, threat analysis, and logging in separate threads for non-blocking performance.

  Blocklist Updates:

- Periodically downloads and refreshes blocklists every 24 hours to ensure the latest data is used.

  2.Functions

  download_blocklists()

- Fetches blocklists from predefined URLs and processes them.
- Adds valid IPs to the Bloom Filter for fast lookups.

  process_blocklist(data)

- Parses the blocklist data to extract valid IPs.

  packet_capture(packet_queue)

- Captures network packets using Scapy and places them into a queue.

  ip_extraction(packet_queue, ip_queue)

- Extracts source and destination IPs from captured packets and forwards them for analysis.

  threat_analysis(ip_queue, analysis_queue)

- Chains multiple threat detection methods to classify IPs as Malicious or Clean:
    - Bloom Filter Check for quick validation.
    - Google Safe Browsing API for advanced threat detection.
    - Custom Reputation Check for additional filtering.

result_logging(analysis_queue)

- Logs analyzed results to a CSV file on the desktop, including IP, reputation, analysis flags, and timestamp.

periodic_update_blocklist()

- Refreshes the blocklist periodically to ensure up-to-date threat data.

listen_for_stop()

- Monitors for user input to gracefully stop the program.

3.Workflow

Initialization:

- Prompts the user for a CSV filename.
- Sets up paths, blocklists, and thread queues.

Blocklist Handling:

- Downloads blocklists and processes them into a Bloom Filter.

Packet Capture and Analysis:

- Captures live packets and extracts IPs.
- Chains multiple threat detection methods to classify IPs.

Logging:

- Writes results to a CSV file in real-time.

Background Execution:

- Runs all major tasks in separate threads for efficient and non-blocking operation.

User Interaction:

- Listens for a stop signal ('S' key) or automatically stops after a set duration (1 minute).

4.Dependencies

- Python 3.7 or later
- Required Libraries:
    - Scapy: For packet capture and processing.
    - pybloom-live: For efficient IP filtering.
    - Requests: For downloading blocklists and interacting with APIs.
    - Csv, Time, Threading, Queue: Standard libraries for data handling and concurrency.
    - Msvcrt: For detecting key presses in Windows.

5.Limitations

Platform-Specific:

- Designed for Windows due to the use of `msvcrt` for key press detection.

Google Safe Browsing API:

- Requires a valid API key. Usage may be limited by API quotas.

Blocklist Reliability:

- Depends on the accuracy and freshness of downloaded blocklists.

Performance Overhead:

- Real-time packet capture and analysis may affect system performance.

Static IP Reputation Check:

- Reputation rules are hardcoded and may require manual updates.

6.Output

- **CSV File**: Saved on the desktop with columns for IP, reputation, method flags, and timestamp.
- **Console Logs**: Displays detected malicious IPs in real-time.

7.Usage

- Install the required libraries:
  pip install scapy pybloom-live requests
- Enter a name for the CSV output file when prompted.
- Monitor the console for real-time updates.
- Stop the script by pressing the 'S' key or wait for the set duration (default: 1 minute).