Cybersecurity Python Project

File and Directory Monitoring Tool

Student Name: Mohammed Abdollah Hasan AL_odaini **Student ID**: 2023050626
Department: Cybersecurity **Submission Date**: 16 September 2025

Abstract

This project presents a tool designed to monitor and track file and directory operations. Its main objective is to detect any changes in the file system, such as creation, deletion, modification, or renaming, and log these events for analysis. The tool uses SHA256 hashing to verify file integrity and detect unauthorized changes. It also records the original and updated names of renamed files.

The monitoring logic is implemented using Object-Oriented Programming (OOP) principles, which allow for modular and scalable design. The results are displayed in a simplified and user-friendly format, making it easier to understand and analyze file system activity.

Introduction

In today's rapidly evolving digital world, monitoring file system activity has become a critical task in cybersecurity. The ability to detect unauthorized changes, access attempts, and suspicious behavior within directories and files is essential for protecting sensitive data and maintaining system integrity.

This project aims to address this challenge by developing a tool that monitors file and directory operations—such as creation, deletion, modification, and renaming—and logs these events for further analysis. The tool also tracks changes in file names and content, helping identify potential threats or tampering.

By implementing this solution using Object-Oriented Programming (OOP), the tool ensures modularity, scalability, and ease of maintenance. It provides a fast and efficient way for system administrators and security analysts to observe file system behavior, detect anomalies, and respond to incidents in real time.

Methodology

This tool was developed using Python and focuses on monitoring file and directory operations with high accuracy and responsiveness. The system is modular and scalable, built around the following core components:

1. Directory Monitor Class

This is the heart of the system. It handles: * Scanning the target directory. * Comparing current and previous states. * Detecting changes such as creation, deletion, modification, and renaming. * Logging events to a text file.

2. Hashing for Integrity

To detect file modifications, the tool uses SHA256 hashing. Each file’ s hash is calculated and stored, allowing the system to detect content changes even if the filename remains the same.

3. State Management

The previous state of the directory is stored in a JSON file (`monitor_states.json`). This includes: * File names. * Directory structure. * Hash values.

The tool compares this saved state with the current scan to detect: * New files or folders. * Deleted items. * Modified content. * Renamed files or directories.

4. Logging

All detected operations are recorded in `log.txt`, with timestamps and operation types. Each entry is formatted for easy parsing and later visualization.

5. Menu-Driven Interface

The user interacts with the tool through a terminal-based menu. Options include: * Start monitoring. * View log summary. * Visualize results. * Reset state. * Exit.

This interface ensures ease of use and guides the user through all available features.

Threading Implementation

To improve performance and responsiveness, the tool uses Python's `threading` module. Monitoring operations run in a background thread, allowing the interface to remain active and responsive. This ensures: * Real-time scanning without freezing the UI. * Efficient handling of large directories. * Smooth user experience.

Visualization Module

The tool includes a visualization module using `matplotlib`, offering: * Horizontal bar charts. * Combined charts (both views in one figure).

Each chart displays the frequency of operations (create, delete, modify, rename) and is saved automatically with a timestamped filename. Colors are mapped using `viridis` for a clean, professional look.

Libraries Used

- `os`: File system access.
- `json`: State storage.
- `hashlib`: SHA256 hashing.
- `re`: Regular expressions.
- `time`: Timestamps and delays.
- `threading`: Background execution.
- `matplotlib`: Data visualization.

Algorithm or Code Design

The tool's design is based on the `DirectoryMonitor` class, which encapsulates all core monitoring functionalities. This class includes several key methods that work together to detect and log changes.

`DirectoryMonitor` Class

This class is responsible for scanning the target directory, comparing current and previous states, detecting changes (creation, deletion, modification, renaming), logging events, and saving and loading state data.

`__init__(self, dir_path, log_file='log.txt')`

This function initializes the monitoring object. It validates the directory path, sets up log and snapshot files, and loads the previous state.

```
class DirectoryMonitor:
    def __init__(self, dir_path, log_file='log.txt'):
        if not os.path.isdir(dir_path):
            raise FileNotFoundError(f" The specified directory does not exist:
{dir_path}")
        self.dir_path = dir_path
        self.log_file = log_file
        self.snapshot_file = os.path.join(dir_path, '.monitor_states.json')
        self.current_state = {'files': {}, 'directories': {}}
        self.previous_state = self.load_state()
        self.file_metadata = {}
        self.monitor_active = False
```

`_calculate_hash(self, file_path)`

This function calculates the SHA256 hash of a file to detect content changes. It logs an error if the file is inaccessible.

```
    def _calculate_hash(self, file_path):
        hasher = hashlib.sha256()
        try:
            with open(file_path, 'rb') as f:
                while True:
                    text_body = f.read(4096)
                    if not text_body:
                        break
                    hasher.update(text_body)
            return hasher.hexdigest()
        except(IOError , PermissionError):
            self._log_event(f" can not access this file
{os.path.basename(file_path)}")
            return  None
```

`_get_current_state(self)`

This function scans the directory and collects metadata for all files and folders, excluding internal files.

```
    def _get_current_state(self):
        state = {'files': {}, 'directories': {}}
        for root, dirs, files in os.walk(self.dir_path):
            for d in dirs:
                dir_path = os.path.join(root, d)
                try:
                    state['directories'][dir_path] = {
                    'modified': os.path.getmtime(dir_path),
                    'basename': os.path.basename(dir_path) }
                except FileNotFoundError:
                    continue
            for file in files:
                file_path = os.path.join(root, file)
                if file_path == self.snapshot_file or file_path ==
self.log_file:
                    continue
                file_hash = self._calculate_hash(file_path)
                if file_hash is not None:
                    try:
                        state['files'][file_path] = {
                            'hash': file_hash,
                            'size': os.path.getsize(file_path),
                            'basename': os.path.basename(file_path),
                            'modified_time': os.path.getmtime(file_path)
                        }
                    except FileNotFoundError:
                        continue
        return state
```

`load_state(self)`

This function loads the previous state from the snapshot file.

```
    def load_state(self):
        if os.path.exists(self.snapshot_file):
            try:
                with open(self.snapshot_file, 'r' , encoding='utf-8') as f:
                    return json.load(f)
            except JSONDecodeError:
                self._log_event(" cannot  load this file ")
        return {'files': {}, 'directories': {}}
```

`save_state(self, state)`

This function saves the current directory state to a JSON file for future comparison.

```
    def save_state(self , state):
       try:
            with open(self.snapshot_file, 'w' , encoding='utf-8') as f:
                json.dump(state, f, indent=4)
       except IOError:
           self._log_event("cannot save his file")
```

`_log_event(self, message)`

This function logs a message with a timestamp to both the console and the log file. It handles write errors gracefully.

```
    def _log_event(self, message):
        timestamp = time.strftime('%Y-%m-%d %H:%M:%S')
        log_message = f"[{timestamp}] {message}"
        print(log_message)
        try:
            with open(self.log_file, 'a', encoding='utf-8') as f:
                f.write(log_message + "\n")
        except IOError:
            print(f"[{timestamp}]  cannot write in log.txt file")
```

`_find_renamed_items(self, old_file_data, new_files_state, item_type)`

This function detects renamed files or directories by comparing metadata such as hash, size, and modification time.

```
    def _find_renamed_items(self, old_file_data, new_files_state ,item_type):
        for new_path, new_data in new_files_state.items():
            if item_type == 'files':
                if (new_data['hash'] == old_file_data['hash'] and
                        new_data['size'] == old_file_data['size'] and
                        new_data['modified_time'] ==
old_file_data['modified_time'] and
                        new_path not in self.current_state['files']):
                        return new_path
            elif item_type == 'directories':
                if (new_data['modified'] == old_file_data['modified'] and
                        new_path not in self.current_state['directories']):
                        return new_path
        return None
```

`monitor_changes(self)`

This function compares current and previous states to detect created, deleted, modified, and renamed items. It logs all changes.

```
    def monitor_changes(self):
        print("=" * 80)
        self._log_event(f"Monitoring: {self.dir_path}")
        print("=" * 80)
        current_state = self._get_current_state()

        old_files = self.previous_state.get('files', {})
        new_files = current_state.get('files', {})
        old_dirs = self.previous_state.get('directories', {})
        new_dirs = current_state.get('directories', {})

        renamed_files = {}
        for old_path, old_data in old_files.items():
            new_path = self._find_renamed_items(old_data, new_files ,'files' )
            if new_path and old_data['basename'] != os.path.basename(new_path):
                renamed_files[old_path] = new_path
                self._log_event(f"File RENAMED: {old_path} -> {new_path}")


        for file_path in set(old_files) - set(new_files) - set(renamed_files):
            self._log_event(f"File DELETED: {file_path}")

        for file_path, new_data in new_files.items():
            if file_path not in old_files and file_path not in
renamed_files.values():
                self._log_event(f"File CREATED: {file_path} (Size:
{new_data['size']} bytes)")
            elif file_path in old_files and old_files[file_path]['hash'] !=
new_data['hash']:
                self._log_event(f"File MODIFIED: {file_path}")


        renamed_directories = {}
        for old_path, old_data in old_dirs.items():
            new_path = self._find_renamed_items(old_data,new_dirs,
'directories')
            if new_path and old_data['basename'] != os.path.basename(new_path):
                renamed_directories[old_path] = new_path
                self._log_event(f"Directory RENAMED: {old_path} -> {new_path}")

        for old_path in old_dirs:
            if old_path not in new_dirs and old_path not in
renamed_directories:
                self._log_event(f"Directory DELETED: {old_path}")

        for new_path in new_dirs:
            if new_path not in old_dirs and new_path not in
renamed_directories.values():
                self._log_event(f"Directory CREATED: {new_path}")

        self.previous_state = current_state
        self.save_state(current_state)
```

`start_monitoring(self, interval=5)`

This function starts the monitoring loop with a defined interval. It runs in a separate thread to maintain UI responsiveness.

```
    def start_monitoring(self, interval=5):
        if not os.path.isdir(self.dir_path):
            self._log_event(f"ERROR: Directory not found: {self.dir_path}")
            return

        self._log_event("first scan complete. Monitoring started...")
        self.monitor_active = True
        print("\n--- Starting monitoring (Press Ctrl+C to return to menu) ---")
        try:
            while self.monitor_active:
                self.monitor_changes(),
                time.sleep(interval)
        except KeyboardInterrupt:
            self._log_event("Monitoring stopped by user")
            self._log_event("Monitor stopped")
            self.monitor_active = False
```

`stop_monitor(self)`

This function gracefully stops the monitoring loop.

```
    def stop_monitor(self):
        self.monitor_active = False
        self._log_event("Monitor stopped")
```

Helper Functions

`show_menu_items(pattern_f, pattern_d, name_f, log_file='log.txt')`

This function displays a submenu for viewing specific types of log entries (created, deleted, renamed, modified).

```
def show_menu_items(pattern_f, pattern_d, name_f, log_file='log.txt'):

    while True:
        print("\n" + "=" * 80)
        print(f"=======[ View: {name_f.upper()} ]=======")
        print("=" * 80)
        print(f'[1] View {name_f} files')
        if pattern_d:
            print(f'[2] View {name_f} directories')
        print('[0] return to Main Menu')
        print("=" * 80)

        try:
            choice = input("Enter your choice: ")
            if choice == '1':
                display_logs(pattern_f, log_file)
            elif choice == '2' and pattern_d:
                display_logs(pattern_d, log_file)
            elif choice == '0':
                break
            else:
                print("invalid option, please try again.")
        except ValueError:
            print(" Please enter a valid number.")
        except Exception as e:
            print(f" unexpected error occurred: {e}")
```

`display_logs(pattern, log_file)`

This function displays log entries that match a given regular expression pattern.

```
def display_logs(pattern, log_file):
    print("\n====== Log Results ======")
    found = False
    try:
        with open(log_file, 'r', encoding='utf-8') as file:
            for line in file:
                if pattern.search(line):
                    print(line.strip())
                    found = True
    except FileNotFoundError:
        print("Log file not found.")
        input("\n============== Press Enter to continue ==================")
        return

    if not found:
        print("No matching log entries were found.")
    input("\n-- Press Enter to continue --")
```

`count_log_events(log_file='log.txt')`

This function parses the log file and counts the number of operations using regular expressions. It returns a dictionary of counts.

```
def count_log_events(log_file='log.txt'):
    patterns = {
        'File CREATED': r'File CREATED:',
        'File DELETED': r'File DELETED:',
        'File MODIFIED': r'File MODIFIED:',
        'File RENAMED': r'File RENAMED:',
        'Directory CREATED': r'Directory CREATED:',
        'Directory DELETED': r'Directory DELETED:',
        'Directory RENAMED': r'Directory RENAMED:'
    }

    counts = Counter()
    try:
        with open(log_file, 'r', encoding='utf-8') as f:
            for line in f:
                for label, pattern in patterns.items():
                    if re.search(pattern, line):
                        counts[label] += 1
    except FileNotFoundError:
        print("Log file not found.")
    return counts
```

`visualize_log_counts_horizontal(counts)`

This function generates a horizontal bar chart summarizing the frequency of operations. It saves the chart as a PNG image.

```
def visualize_log_counts_horizontal(counts):
    try:
        labels = list(counts.keys())
        values = list(counts.values())
        y = np.arange(len(labels))

        norm = plt.Normalize(min(values), max(values))
        colors = plt.cm.viridis(norm(values))

        fig, ax = plt.subplots(figsize=(12, 6))
        bars = ax.barh(y, values, color=colors, edgecolor='black', linewidth=1)

        ax.set_yticks(y)
        ax.set_yticklabels(labels, fontsize=12)
        ax.set_title('Summary of File and Directory Operations', fontsize=16,
weight='bold')
        ax.set_xlabel('Count', fontsize=14)
        ax.set_ylabel('Operation Type', fontsize=14)
        ax.grid(axis='x', linestyle='--', alpha=0.3)

        for bar in bars:
            width = bar.get_width()
            ax.text(width - (width * 0.05), bar.get_y() + bar.get_height()/2,
                    f'{int(width)}', ha='right', va='center', color='white',
fontsize=11, weight='bold')

        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)

        timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
        filename = f"log_chart_horizontal_{timestamp}.png"
        plt.tight_layout()
        plt.savefig(filename, dpi=300)
        print(f" Horizontal chart saved successfully as: {filename}")
        plt.show()

    except Exception as e:
        print(f" Error generating horizontal chart: {e}")
```

Control Flow and Main Execution Logic

The `main()` function serves as the entry point of the program. It prompts the user to enter the path of the directory to be monitored. Once the path is provided, the program initializes an instance of the `DirectoryMonitor` class, which is responsible for scanning the directory, detecting changes, and logging events.

Upon initialization, the program attempts to load the previous state from a JSON file named `monitor_states.json`. This file contains metadata about files and directories from the last scan, allowing the program to compare and detect changes such as creation, deletion, modification, or renaming.

After loading the state, the `main_menu()` function is called. This function displays a terminal-based menu that allows the user to: * `1` Start monitoring in a separate thread. * `-1` Stop monitoring. * `2`, `3`, `4`, `5` View created, deleted, renamed, and modified items. * `6` Visualize log summary. * `0` Exit the program safely.

It uses `try-except` blocks to handle invalid inputs and unexpected errors gracefully.

Threading Integration

To ensure the interface remains responsive while monitoring is active, the program uses Python's `threading` module. When the user selects "Start Monitoring," the `start_monitoring()` method is executed in a background thread. This allows real-time monitoring without blocking the menu interface, making the tool suitable for continuous observation.

Summary of Execution Flow

1. **User Input:** Directory path via terminal.
2. **Initialization:** `DirectoryMonitor` object created.
3. **State Loading:** Previous state loaded from `monitor_states.json`.
4. **Monitoring:** Changes detected and logged.
5. **Menu Interaction:** User navigates options to view or visualize logs.
6. **Threading:** Monitoring runs in the background for real-time updates.
7. **Error Handling:** All critical operations wrapped in `try-except`.

Input and Output Handling

The tool handles input and output in multiple ways to ensure ease of use and provide comprehensive information to the user.

Terminal Input

The user enters the full directory path via the terminal. After submission, the program verifies the validity of the path. If the path is incorrect, an error message is displayed in the terminal. Input is handled through a command-line prompt, and the user selects from options (0-5) to interact with the tool.

File Input

The program reads the current state of the directory from a file named `monitor_states.json`. This file contains metadata such as file hashes, sizes, and modification timestamps, allowing the tool to detect changes.

Terminal Output

The program displays real-time messages in the terminal, including: * Monitoring status. * Detected changes. * Error messages. * Operation summaries.

Each operation type (created, deleted, renamed, modified) is clearly shown to the user.

File Output

All detected operations are saved to a log file named `log.txt`. The log includes timestamps and operation types, making it easy to analyze changes later. The state file `monitor_states.json` is also updated after each scan to reflect the latest directory structure.

"""

Error Handling

The tool incorporates robust error handling mechanisms to ensure its stability and reliability. The following common error types are handled:

`FileNotFoundError`

This error occurs when the user enters a directory path that does not exist or is inaccessible. The program displays a clear error message and safely stops execution. This error may also occur if `log.txt` or `monitor_states.json` is missing.

`PermissionError` and `IOError`

These errors occur when the program cannot access a file due to permission restrictions or file corruption. The tool logs the issue and continues execution without

crashing.

`JSONDecodeError`

This error occurs when the `monitor_states.json` file is present but contains invalid JSON. The program logs the error and initializes an empty state to continue monitoring safely.

`ValueError`

This error occurs when the user enters an invalid option in the menu (e.g., letters instead of numbers). The program prompts the user to enter a valid number and does not exit.

`General Exception`

This catches any unexpected errors during runtime in `main()` or `main_menu()`. The program displays a generic error message and continues safely.

Object-Oriented Programming (OOP) Design

This monitoring tool is built using the principles of Object-Oriented Programming (OOP), which provides a structured and scalable approach to software development. The program is centered around a single class named `DirectoryMonitor`, which encapsulates all the core logic and functionality required for monitoring file system changes.

By using OOP, the tool benefits from: * **Encapsulation:** All attributes and methods are grouped within the class, making the code organized and modular. * **Reusability:** The class can be reused in other projects or extended with additional features. * **Maintainability:** The structure allows for easy updates and debugging. * **Modeling:** The class models real-world behavior by representing a directory as an object with state and behavior.

`DirectoryMonitor` Class

This class is responsible for: * Scanning the target directory. * Comparing current and previous states. * Detecting changes (created, deleted, modified, renamed). * Logging events. * Saving and loading state data.

Key Attributes

Attribute	Description
`dir_path`	The full path of the directory to be monitored
`log_file`	The name of the log file where events are recorded (`log.txt`)
`snapshot_file`	The path to the JSON file storing the previous state (`monitor_states.json`)
`current_state`	Dictionary containing the current state of files and directories
`previous_state`	Dictionary containing the last saved state for comparison

Key Methods

Method	Purpose
`__init__()`	Initializes the object and loads previous state
`_calculate_hash()`	Computes SHA256 hash of a file to detect content changes
`_get_current_state()`	Scans the directory and builds the current state
`load_state()`	Loads the previous state from the snapshot file
`save_state()`	Saves the current state to the snapshot file
`_log_event()`	Logs events with timestamps to the log file
`_find_renamed_items()`	Detects renamed files or directories based on metadata comparison
`monitor_changes()`	Compares states and logs detected changes
`start_monitoring()`	Starts the monitoring loop with a defined interval
`stop_monitor()`	Stops the monitoring loop gracefully

This class-based design ensures that the tool remains organized, extensible, and easy to maintain. It also allows for future enhancements such as GUI integration, real-time alerts, or multi-directory support without disrupting the core logic.

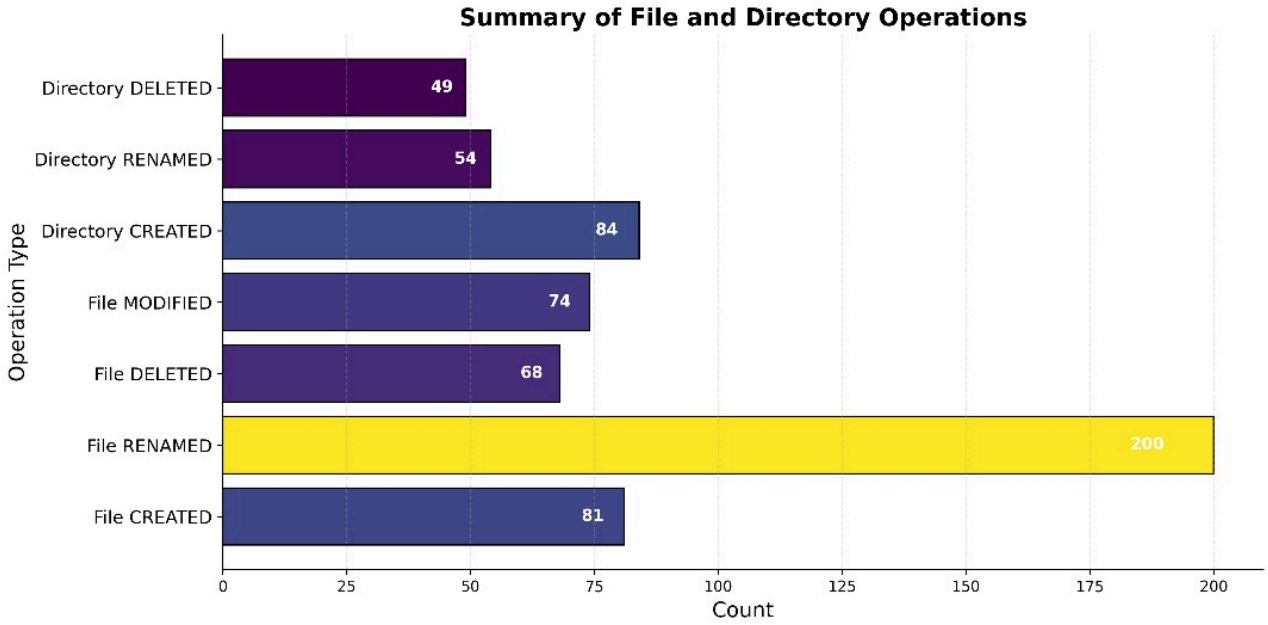
Network and Web Features (if applicable)

This tool currently does not include any network or web features. However, it can be extended in the future to support remote monitoring or integration with other systems over the network, as discussed in the "Future Enhancements and Recommendations" section.

Output Visualization

The tool includes a visualization module using `matplotlib` to provide a visual summary of file and directory operations. A horizontal bar chart is generated,

illustrating the frequency of different operations (creation, deletion, modification, renaming) and is automatically saved as a PNG image.



Future Enhancements and Recommendations

To improve the tool’ s usability, scalability, and integration with modern systems, the following enhancements are recommended:

Graphical User Interface (GUI)

The tool can be upgraded from a command-line interface to a graphical user interface, making it more user-friendly and accessible. This can be achieved using frameworks such as Tkinter or PyQt. A GUI would allow users to interact with the tool through buttons, menus, and visual panels, reducing the need for manual command-line input.

Network Integration

The tool can be extended to support remote monitoring using technologies like Sockets or RPC (Remote Procedure Call). This would enable centralized monitoring across multiple devices, allowing administrators to track changes from a single dashboard.

SIEM Integration (Security Information and Event Management)

To support enterprise-level security, the tool can be integrated with SIEM platforms. This would allow: * Real-time alerts. * Centralized log collection. * Automated incident response.

Such integration would make the tool suitable for professional cybersecurity environments.

Behavioral Analysis

Future versions could include anomaly detection by analyzing behavioral patterns in file operations. For example: * Sudden spikes in deletions. * Frequent renaming of sensitive files. * Unusual access times.

This would help identify suspicious activity and potential threats.

Cross-Platform Support

The tool can be adapted to run on multiple operating systems, including: * Windows. * macOS. * Linux.

This ensures broader usability and deployment flexibility.

Packaging and Distribution

To simplify installation and usage, the tool can be packaged as a standalone application or installer. This would make it easier to distribute and deploy in various environments.

Full Recursive Scanning

The tool can be enhanced to support deep recursive scanning of directories, with configurable depth and filters. This would allow comprehensive monitoring of complex directory structures.

State File Optimization and Encryption

The `monitor_states.json` file can be optimized for faster read/write operations and better performance during large-scale scans. Encryption can also be added for increased security.

Conclusion

This project demonstrates the design and implementation of a file and directory monitoring tool using Python. It reflects a strong understanding of programming fundamentals, error handling, and system architecture.

The tool was built with a focus on: * **Modularity** through object-oriented design. * **Reliability** via robust error handling. * **Scalability** for future enhancements. * **Responsiveness** using threading. * **Clarity** through terminal and file-based outputs.

The `DirectoryMonitor` class serves as the core of the system, encapsulating all logic related to scanning, comparing, logging, and saving state. The program handles exceptions gracefully and provides clear feedback to the user, making it suitable for real-world use.

This project has strengthened my skills in structured programming, system design, and practical cybersecurity applications. It also opened the door to future improvements such as GUI integration, network support, and SIEM compatibility.

References

Python Official Documentation >> https://docs.python.org
Real Python Tutorials >> https://realpython.com
Stack Overflow >> https://stackoverflow.com
GitHub – Code Examples and BesPractices>>https://github