Table of Contents

Getting Started with SPECTRE

Welcome to the **SPECTRE Getting Started Guide**! This guide is designed to help developers set up, understand, and start working with SPECTRE, a cutting-edge memory forensics tool focused on analysing RAM images from Windows-based systems. Whether you're a seasoned professional in memory forensics or a new developer exploring this domain, this guide will provide a structured approach to kickstart your journey.

Overview of SPECTRE

SPECTRE is a powerful Python-based system developed to streamline memory forensics workflows. It offers advanced capabilities, including scenario simulation, anomaly detection, visualization of memory deltas, and compatibility with the JSON intermediate format for efficient processing. Its modular architecture ensures scalability and adaptability in handling complex and evolving forensic datasets.

System Requirements

To successfully run SPECTRE, the following environment setup and dependencies are required:

1. **Python Version:** SPECTRE is developed using **Python 3.7.3**. Ensure that this version (or higher) is installed on your system.
2. **Operating System:** SPECTRE has been tested and implemented on a **Windows 10** environment.
3. **Supported Data Formats:** SPECTRE currently supports the analysis of **RAM images specific to Windows-based systems**.

Required Python Libraries

SPECTRE relies on several additional Python libraries, which are not included by default in Python 3.7.3. The table below outlines these dependencies, their functionality, and installation commands:

Library	Description	Installation Command
matplotlib	Comprehensive library for static, animated, and interactive visualizations in Python.	pip3 install matplotlib
Faker	Generates fake data, such as names, addresses, emails, and phone numbers.	pip3 install faker
requests	Simplifies sending HTTP requests to access web resources.	pip3 install requests
dns.resolver	A module from dnspython for querying DNS records.	pip3 install dnspython
networkx	Facilitates creating, analyzing, and visualizing complex networks or graphs.	pip3 install network
PrettyTable	Provides an easy way to render ASCII tables.	pip3 install PrettyTable
colorama	Enables colored terminal text and styles.	pip3 install colorama
tabulate	Allows pretty-printing of tabular data in plain text, Markdown, and other formats.	pip3 install tabulate
Pillow	A fork of the Python Imaging Library (PIL) for image processing.	pip3 install pillow
scipy	A library for scientific computing and technical computing.	pip3 install scipy
Numpy	A powerful library for numerical computing.	pip3 install numpy

Installation Instructions

1. **Install Python:** Download and install Python 3.7.3 from the https://www.python.org/downloads/. Ensure you add Python to your system's PATH during installation.
2. **Install Dependencies:** Use the following command to install all required libraries:

```
$ pip3 install -r requirements.txt
```

3. **Download SPECTRE:** Clone the SPECTRE repository or download the source code from the official project page.
4. **Volatility 3 Framework:** Clone the Volatility 3 source code from https://github.com/volatilityfoundation/volatility3
5. **Execution:** Follow rest of the getting started guide to get idea of the system and available example programs demonstrating the behavior of system in great details.

Additional Notes

- **Documentation:**
  Refer to the accompanying documentation for detailed instructions on SPECTRE's features and modules.
- **Support:**
  For any issues during setup or usage, consult the FAQ section or reach out to the support team.
- **Future Compatibility:** As the project evolves, ensure regular updates and adapt dependencies as necessary.

With this guide, you're ready to begin exploring and contributing to SPECTRE. Happy coding!

Memory Module

The **Memory Module** provides a streamlined approach to analyzing system memory data through its core MemoryDump class, which serves as the backbone for further data processing. This class is responsible for organizing, and enabling efficient handling of memory-related information, setting the stage for advanced visualizations and forensic analyses.

Key functionalities include:

1. **Data Structuring**: The MemoryDump class organizes raw memory data into accessible formats for efficient processing.
2. **Visualization Ready**: Facilitates generation of statistical insights, such as memory process distributions and protocol usage, for clear, actionable understanding.
3. **Anomaly Detection**: Supports identifying malicious activities, unsafe extensions, or blacklisted entities through integrated analysis.

Designed for ease of use and scalability, the Memory Module empowers users with the tools needed to interpret and secure their system's memory footprint effectively.

MemoryDump

The MemoryDump class facilitates handling and analyzing memory dump data from SPECTRE intermediate format. It includes methods for loading, manipulating, and visualizing data from memory dumps.

Core Methods

1. **Loading Data**:
    - loadFiles(processes_file, connections_file, users_file, modules_file, dlls_file, registries_file): Loads data from individual JSON files and extracts IPs from connections.
    - loadDirectory(directory): Loads data from a directory with predefined filenames like pstree.json and netstat.json.
    - load_json(file_path): Reads a JSON file and returns the parsed content.
2. **Managing Data**:
    - read_dump(file_path): Reads and initializes data from a consolidated JSON dump file.
    - save_json(output_path): Saves the current data into a JSON file.
3. **Data Accessors**:

    - getProcesses(), getConnections(), getUsers(), getModules(), getIPAddresses(), getRegistries(), getCommands(): Retrieve respective data categories.
4. **Data Adders**:
    - Methods like addProcesses(newProcesses) and addConnections(newConnections) extend existing data with new entries.
5. **Utility Methods**:
    - _extract_commands(processes): Extracts commands recursively from processes.
    - plot_statistics(): Visualizes data statistics as a horizontal bar chart.

Data Categories

- **Processes**: Process tree data with details like PID, command, and children.
- **Connections**: Network connection details, including IPs and protocols.
- **Users**: User credential data (e.g., hashes).
- **Modules**: Loaded modules with paths and PIDs.
- **Registries**: Registry keys, names, and data.
- **Commands**: Extracted commands from processes.
- **IPs**: Foreign IPs extracted from connections.

Visualization

- plot_statistics() displays counts of data categories (Processes, Connections, etc.) using a bar chart.

MemoryAnalysis

The MemoryAnalysis class provides methods to visualize and analyze memory dump statistics, focusing on processes, connections, and malicious activity. It includes two main methods:

1. plot_detailed_statistics(memoryDump): Creates comprehensive visualizations for various memory dump statistics.
2. plot_additional_statistics(memoryDump, whitelist_ips, blacklist_ips, vt_key): Focuses on forensic anomaly detection and visualization.

Key Functionalities

1. plot_detailed_statistics(memoryDump)

- **Purpose**: Generate visual insights into processes and connections from a memory dump.
- **Visualizations**:
    - **Processes**:
        - Parent vs. Child distribution (Pie Chart).
        - Running vs. Closed status (Pie Chart).
    - **Connections**:
        - Internal vs. External connections (Pie Chart).
        - Connection states (Horizontal Bar Chart).
        - Protocol types (Pie Chart).
    - **Top Entities**:
        - Top 3 processes with most DLLs (Pie Chart).
        - Top 3 processes with most threads (Pie Chart).
        - Top 3 processes with most connections (Pie Chart).
        - Protocol distribution for the process with the most connections (Horizontal Bar Chart).
- **Key Insights**:
    - Detailed analysis of process relationships and states.
    - Network activity by protocols and connection types.
    - Identification of top resource-intensive processes.

2. plot_additional_statistics(memoryDump, whitelist_ips, blacklist_ips, vt_key)

- **Purpose**: Enhance forensic analysis with a focus on malicious patterns and anomalous activities.
- **Visualizations**:
    - **Unsafe Extensions**:
        - Chart summarizing processes with unsafe extensions.
    - **Malicious Activity**:
        - Detection of malicious rundll32 parent processes.
        - Detection of credential dumping methods (e.g., ProcDump).
    - **Network Connections**:
        - Geographic distribution of connection origins.
        - Connections with blacklisted IPs.
        - Top 3 foreign IPs with the most connections (Horizontal Bar Chart).
    - **Scatter Plot**:
        - Processes marked as malicious with supporting indicators.
- **Key Insights**:
    - Identification of potentially compromised processes and IPs.
    - Assessment of network anomalies using blacklists, whitelists, and VirusTotal integration.

How to Use

1. **Setup**:
    - Provide a memoryDump object containing processes, connections, and modules.
    - For plot_additional_statistics, include paths to whitelist/blacklist files and a VirusTotal API key (vt_key).
2. **Visualization**:
    - Call plot_detailed_statistics(memoryDump) for a statistical overview.
    - Use plot_additional_statistics(memoryDump, whitelist_ips, blacklist_ips, vt_key) for anomaly detection.
3. **Interpret Results**:
    - Review pie and bar charts to identify trends and anomalies.

- Use scatter plots for a concise view of malicious indicators.

Outputs

Both methods display charts and summaries for interactive analysis, with the second method emphasizing forensic anomalies and malicious behavior.

Memory Module Demonstration Program

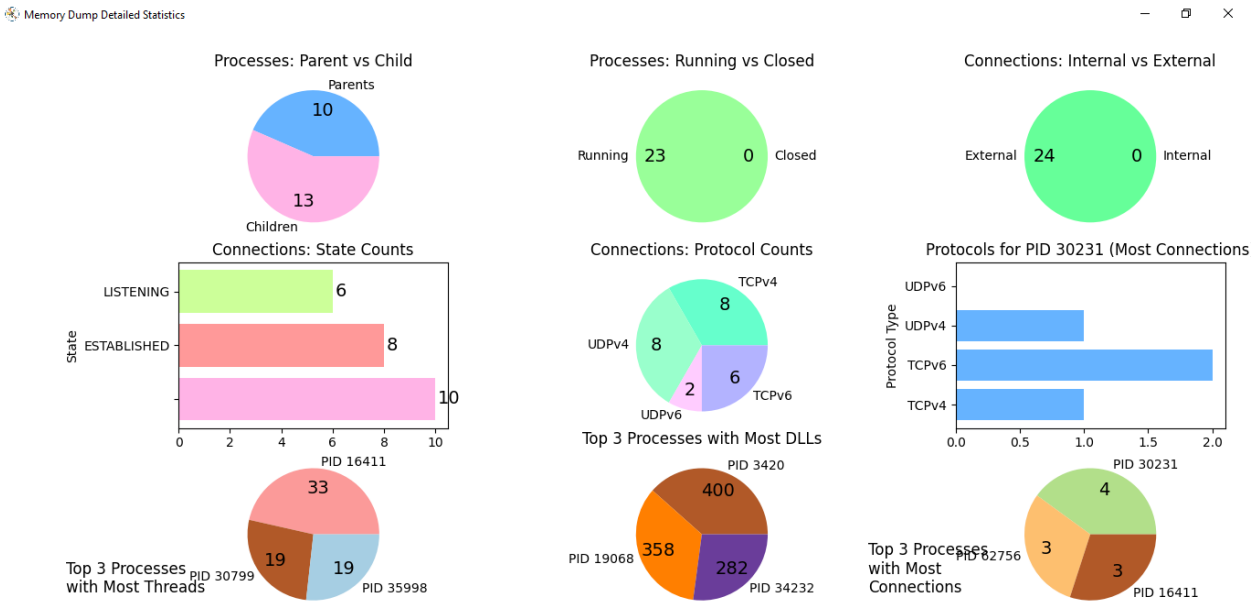
The program **memory_module_demo.py** analyzes memory dump data and generates statistical plots based on various JSON files. Using the MemoryDump class, it processes information about processes, connections, users, modules, DLLs, and registries. It accepts input via command-line arguments, allowing users to specify the source files or a directory for processing. Additionally, it supports blacklisted and whitelisted IP addresses, as well as an API key for VirusTotal.

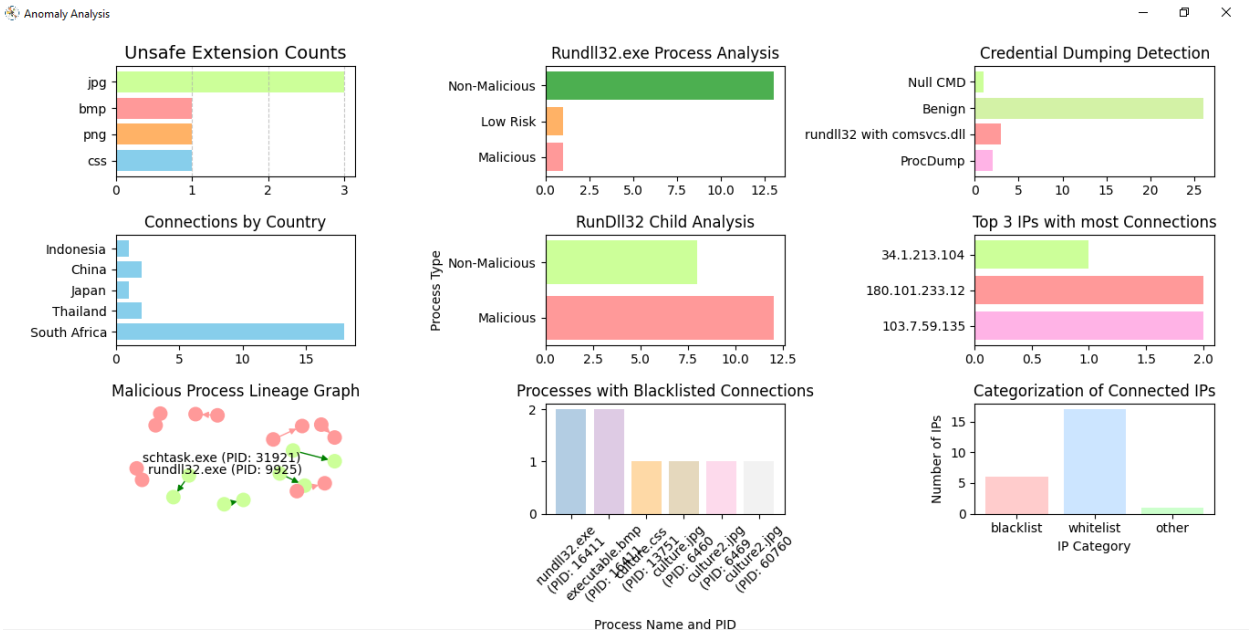
Key components:

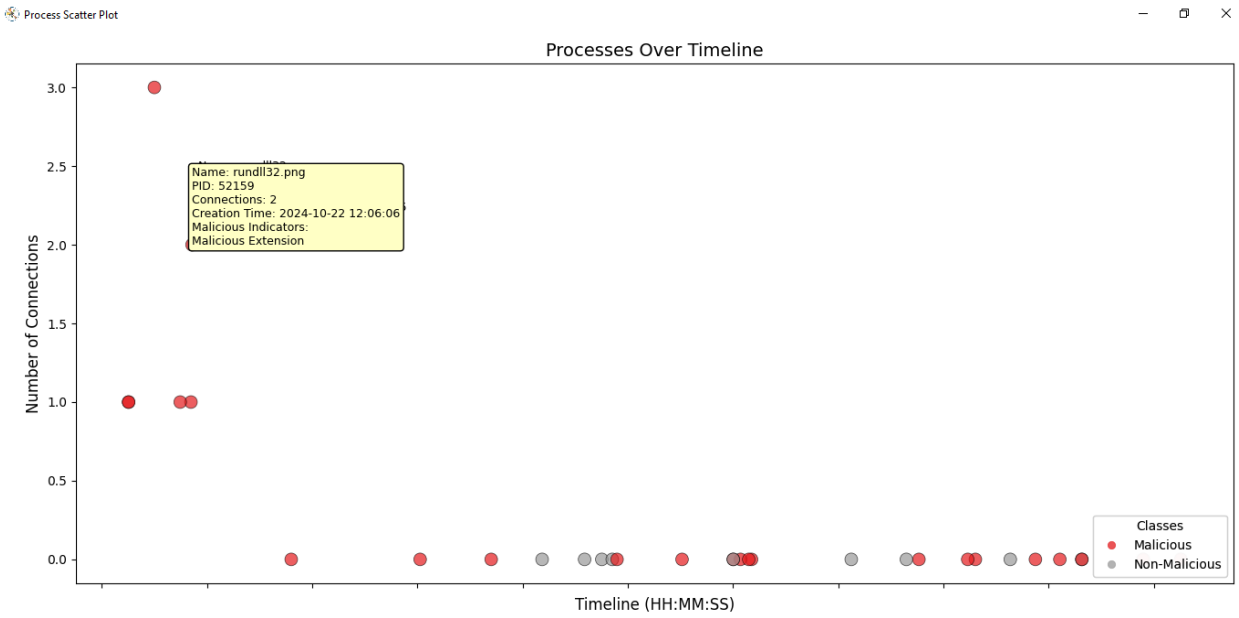
1. **Argument Parsing**: Collects file paths for memory data and configuration (IP lists, VirusTotal API key).
2. **Data Loading**: The MemoryDump class loads data from the directory or individual files.
3. **Statistical Plotting**: Uses the MemoryAnalysis class to generate detailed and additional memory usage statistics.
4. **Memory Usage Tracking**: Tracks memory allocation using tracemalloc and prints current and peak memory usage for benchmark purposes.

The program ends by showing memory statistics and stopping memory tracing. Example run is shown below.

```
$ python ./memory_module_demo.py \
  -input ../sample_outputs/sample2 \
  --whitelist-ips ../emulation/ip_lists/file_google.txt \
  --blacklist-ips ../emulation/ip_lists/compromised_ip_full.txt \
  -k Virus Total key goes here
```


Memory Dump Detailed Statistics
Processes: Parent vs Child
Parents
10
13
Children
Processes: Running vs Closed
Running
23
0
Closed
Connections: Internal vs External
External
24
0
Internal
Connections: State Counts
LISTENING
ESTABLISHED
State
6
8
10
0
2
4
6
8
10
Connections: Protocol Counts
TCPv4
8
UDPv4
8
2
6
TCPv6
UDPv6
Protocols for PID 30231 (Most Connections
UDPv6
UDPv4
TCPv6
TCPv4
Protocol Type
0.0
0.5
1.0
1.5
2.0
PID 16411
Top 3 Processes with Most DLLs
PID 3420
400
PID 19068
358
282
PID 34232
PID 30231
4
3
3
PID 16411
Top 3 Processes with Most Threads
33
19
19
PID 30799
PID 35998
Top 3 Processes with Most Connections


Anomaly Analysis
Unsafe Extension Counts
jpg
bmp
png
css
0
1
2
3
Rundll32.exe Process Analysis
Non-Malicious
Low Risk
Malicious
0.0
2.5
5.0
7.5
10.0
12.5
Credential Dumping Detection
Null CMD
Benign
rundll32 with comsvcs.dll
ProcDump
0
5
10
15
20
25
Connections by Country
Indonesia
China
Japan
Thailand
South Africa
0
5
10
15
RunDll32 Child Analysis
Process Type
Non-Malicious
Malicious
0.0
2.5
5.0
7.5
10.0
12.5
Top 3 IPs with most Connections
34.1.213.104
180.101.233.12
103.7.59.135
0.0
0.5
1.0
1.5
2.0
Malicious Process Lineage Graph
schtask.exe (PID: 31921)
rundll32.exe (PID: 9925)
Processes with Blacklisted Connections
2
1
0
rundll32.exe (PID: 16411
executable.bmp (PID: 16411
culture.css (PID: 13751
culture.jpg (PID: 6460
culture2.jpg (PID: 6469
culture2.jpg (PID: 60760
Process Name and PID
Categorization of Connected IPs
Number of IPs
15
10
5
0
blacklist
whitelist
other
IP Category



Emulation Module

This guide provides instructions on how to install the necessary libraries and use the classes in the Emulation Module to simulate benign and malicious activities, such as process generation, network connections, credential dumping attacks, and more. Additionally, it explains how to generate fake process trees, modules, registry keys, and user data for testing and research purposes in digital forensics and cybersecurity.

Classes for Emulation Module

The Emulation Module includes several classes designed to simulate various security scenarios and generate fake data for process trees, network activity, registry keys, modules, user accounts, and more. Below is a detailed description of each class and its primary functionality.

ProcessTreeEmulator

The ProcessTreeEmulator class generates fake processes, simulates hierarchical process trees, and creates fake command-line arguments and paths.

Key Methods:

- **generate_random_arguments()**: Generates random command-line arguments.
- **generate_fake_process()**: Generates fake process data.
- **distribute_processes_across_depths()**: Distributes processes across multiple tree depths.
- **generate_process_tree()**: Creates a complete process tree with multiple processes.
- **createTree()**: Generates a simple hierarchical process tree.

Example Usage:

```
# Generate a fake process tree
process_tree = ProcessTreeEmulator.generate_process_tree(total_processes=10,
max_depth=3, max_children=2)
```

TestDataEmulator

The TestDataEmulator class leverages the ProcessTreeEmulator to generate hierarchical process trees, generate commands, and save data to JSON files.

Key Methods:

- **create_process_tree()**: Generates a process tree with specific constraints.
- **generate_commands_with_ips()**: Generates commands with benign and malicious IP addresses.
- **write_json_output()**: Saves the generated process tree as a JSON file.

Process Emulation Demonstration Program

The program pstree_emulation.py demonstrates the usage of the TestDataEmulator.create_process_tree method for generating emulated process trees with extensive flexibility and customization through command-line options. The generated process trees can be serialized to JSON and optionally displayed in a tabular format, making it a versatile tool for testing and research in process activity emulation.

Command-Line Options

The program supports several command-line options to customize the emulated process tree generation:

- **-n / --num_processes**: Specifies the total number of processes to generate including child processes). *Default*: 10
- **-d / --depth**: Sets the maximum depth of the process tree, defining how many hierarchical levels are created. *Default*: 1
- **-m / --max_children**: Determines the maximum number of child processes allowed for each parent. *Default*: 10
- **-e / --extensions**: Provides a comma-separated list of file extensions for process image names. These are added to the default extensions (.exe, .bat, .bin). *Default*: None (only default extensions are used)
- **--parent_ids**: Defines the list of parent process IDs for root processes. *Default*: [0]
- **--exclude_pids**: Specifies a list of PIDs to exclude from the generated process tree. *Default*: Empty list
- **--skip_plot**: If set, skips displaying the process table in the console. *Default*: Not set (table is displayed)

Key Features and Flexibility

1. **Extensible Process Tree**: Users can control the size and complexity of the process tree by adjusting --num_processes, --depth, and --max_children.
2. **Custom File Extensions**: The --extensions option enables adding additional file extensions to emulate specific process image types.
3. **Parent and Exclusion Control**: The --parent_ids and --exclude_pids options allow precise control over the roots and exclusion of specific process IDs.
4. **Serialized JSON Output**: The generated process tree is saved as a JSON file (fake_volatility_pstree.json) for further analysis or integration with other tools.

5. **Optional Visualization**: Process data can be displayed in a tabular format using the print_process_table function, offering a clear and human-readable view of the emulated tree. This visualization can be skipped using the --skip_plot flag.

Usage Example

```
$ python pstree_emulation.py --num_processes 10 --depth 3 \
    --max_children 5 \
    --extensions "dll,sys" \
    --parent_ids 1000 \
    --exclude_pids 2000 3000
Generated Volatility pstree JSON data and saved to pstree.json.

EMULATED PROCESSES
```

Serial No	Process Name	Process ID	Parent Process ID
1	event-1360.dll	3946	1000
2	TV-3868.exe	45962	3946
3	win-2954.exe	51419	45962
4	somebody-4725.bat	16209	45962
5	research-5106.bin	56605	45962
6	modern-1767.sys	38536	45962
7	visit-4768.dll	59299	3946
8	health-1762.bin	31210	3946
9	quality-5916.sys	55341	3946
10	behind-6605.bat	11094	3946

This program is a powerful demonstration of the TestDataEmulator.create_process_tree method, offering significant customization to emulate realistic and diverse process activity scenarios.

ConnectionGenerator

This class generates network connections in JSON format, simulating both benign and malicious connections based on the provided parameters like IP addresses, process information, and the benign-to-malicious ratio.

Key Methods:

- **load_ips(filename)**: Loads a list of IP addresses from a file.
- **parse_ratio(ratio_arg)**: Parses the ratio of benign to malicious connections.
- **load_process_tree(filename)**: Loads a process tree from a JSON file.
- **generate_single_connection(newProcess=None)**: Generates a single network connection entry.
- **generate_connections()**: Generates multiple network connections.
- **generate_connections_for_process(process)**: Generates network connections for a specific process.
- **generate_connections_for_all_processes()**: Generates connections for all processes.

Connection Emulation Demonstration Program: netstat_emulation.py

The netstat_emulation.py program demonstrates the functionality of EmulationModule.ConnectionGenerator by generating emulated network connection records in JSON format. The program provides comprehensive command-line options, enabling flexible and detailed control over the creation of connection records, including benign and malicious connections.

Command-Line Options

- **--n** *(required)*: Specifies the number of connections to generate.
- **--benign_list** *(required)*: Path to a file containing a list of benign IP addresses.
- **--malicious_list** *(required)*: Path to a file containing a list of malicious IP addresses.
- **--ratio** *(optional)*: Sets the ratio of benign to malicious connections. *Default*: 9:1

- **--exclude_pids** *(optional)*: A list of process IDs (PIDs) to exclude from connection generation. *Default*: Empty list
- **--pstree_file** *(required)*: Path to a JSON file containing the process tree data. This provides the processes from which connections are generated.

Key Features and Flexibility

1. **IP Source Management**:
    - Loads separate lists of benign and malicious IPs from user-provided files.
    - Supports dynamic adjustment of connection behavior based on IP roles.
2. **Customizable Ratios**:
    - Allows specification of benign-to-malicious connection ratios using the --ratio argument, ensuring controlled simulation of diverse network activity.
3. **Process-Driven Connections**:
    - Utilizes a process tree file (--pstree_file) to derive process data, creating connections that emulate realistic process behaviors.
    - Option to exclude specific PIDs from participating in connection generation.
4. **Flexible Configuration**:
    - Supports generation of any number of connections (--n) with full control over the distribution of benign and malicious activity.
5. **Serialized Output**:
    - Outputs the generated connections to a JSON file (connections.json) for further analysis or integration into security tools.

Usage Example

```
$ python netstat_emulation.py \
  --n 100 \
  --benign_list benign_ips.txt \
  --malicious_list malicious_ips.txt \
  --ratio 8:2 \
  --exclude_pids 101 202 \
  --pstree_file process_tree.json
```

Generates 100 connections with:

- IPs sourced from benign_ips.txt and malicious_ips.txt.
- A benign-to-malicious connection ratio of 8:2.
- Excludes processes with PIDs 101 and 202 from connection generation.
- Process data loaded from process_tree.json.

Output

- **Connections JSON File**: The generated connections are saved in connections.json, containing detailed records of emulated network activity, including information about associated processes and connection roles.

This program provides a robust demonstration of the ConnectionGenerator's capabilities, showcasing its ability to create realistic and customizable emulated network connections.

RunDLL32MaliciousParent

This class generates fake rundll32.exe commands with either benign or malicious parent processes. It helps simulate suspicious process lineage, a technique often used to detect malicious behavior in a system.

Key Methods:

- **parse_ratio(ratio_arg)**: Parses the ratio of benign to malicious commands.
- **generate_command_chain(is_benign, depth)**: Generates a command chain with a specified depth, starting from either a benign or suspicious parent process.
- **generate_commands(benign_ratio, malicious_ratio, count, depth)**: Generates a set of commands based on the benign-to-malicious ratio.

Demonstration Program

The rundll32_malicious_parent_emulation.py program demonstrates the functionality of RunDLL32MaliciousParent by generating command-line executions of rundll32.exe, simulating both benign and malicious parent processes. This program offers detailed

customization through command-line arguments and demonstrates hierarchical process chains with color-coded output.

Command-Line Options

- **--ratio** *(optional)*: Specifies the ratio of benign to malicious commands in the format benign:malicious. *Default*: 9:1 (90% benign, 10% malicious).
- **--count** *(optional)*: The total number of commands to generate. *Default*: 10.
- **--depth** *(optional)*: Depth of the process chain in each command. *Minimum*: 2. *Default*: 2.

Key Features and Flexibility

1. **Benign and Malicious Command Generation**:
    - Generates rundll32.exe commands with plausible arguments and hierarchical parent-child process chains.
    - Simulates realistic scenarios for benign and malicious behaviors.
2. **Customizable Ratios**:
    - Enables control over the proportion of benign versus malicious commands using the --ratio argument.
3. **Process Chain Depth**:
    - Allows users to define the depth of the parent-child process hierarchy in generated commands, with a minimum depth of 2.
4. **Color-Coded Output**:
    - **Benign Commands**: Displayed in **green** for easy identification.
    - **Malicious Commands**: Displayed in **red** for clear distinction.
5. **JSON Output for Processes**:
    - Converts generated process chains into structured process tree data using ProcessTreeEmulator.
    - Saves the output as rundll32_parent_pstree.json for further use or analysis.

Usage Example

```
python rundll32_malicious_parent_emulation.py \
    --ratio 8:2 \
    --count 20 \
    --depth 3
```

- **Ratio**: 80% benign, 20% malicious commands.
- **Count**: Generates 20 commands.
- **Depth**: Each command includes a process chain of 3 levels.

Output

1. **Command Display**:
    - Commands are displayed in the terminal with color-coded differentiation:
        - Benign commands in **green**.
        - Malicious commands in **red**.

```
$ python rundll32_malicious_parent_emulation.py \
    --ratio 8:2 \
    --count 20 \
    --depth 3
Benign Commands:
join.exe -> evening.exe /one/produce.dll, paper -> rundll32.exe /director/glass.dll, create
major.exe -> several.exe /group/budget.dll, plant -> rundll32.exe /region/great.dll, factor
remember.exe -> eat.exe /leg/trade.dll, police -> rundll32.exe /boy/summer.dll, message
natural.exe -> here.exe /our/security.dll, others -> rundll32.exe /both/summer.dll, suffer
many.exe -> young.exe /pay/week.dll, hope -> rundll32.exe /let/account.dll, range
training.exe -> image.exe /company/require.dll, plan -> rundll32.exe /into/option.dll, yes
foreign.exe -> us.exe /control/why.dll, perhaps -> rundll32.exe /federal/summer.dll, growth
onto.exe -> long.exe /authority/seven.dll, mind -> rundll32.exe /party/morning.dll, something
exist.exe -> to.exe /music/rest.dll, also -> rundll32.exe /fire/site.dll, care
or.exe -> front.exe /television/seem.dll, bed -> rundll32.exe /head/hospital.dll, audience
anyone.exe -> market.exe /wish/me.dll, state -> rundll32.exe /least/how.dll, fine
recognize.exe -> any.exe /each/star.dll, book -> rundll32.exe /general/civil.dll, check
body.exe -> take.exe /imagine/analysis.dll, language -> rundll32.exe /local/administration.dll, into
side.exe -> particular.exe /current/why.dll, sometimes -> rundll32.exe /month/husband.dll, knowledge
cold.exe -> none.exe /federal/entire.dll, carry -> rundll32.exe /they/lot.dll, north
ahead.exe -> me.exe /hand/test.dll, both -> rundll32.exe /foot/act.dll, country
phone.exe -> peace.exe /often/difference.dll, fall -> rundll32.exe /last/able.dll, common
particular.exe -> perform.exe /beyond/establish.dll, responsibility -> rundll32.exe /finally/again.dll, middle
this.exe -> send.exe /who/past.dll, type -> rundll32.exe /him/anything.dll, decade

Malicious Commands:
winword.exe -> arrive.exe /effort/data.dll, various -> rundll32.exe /above/current.dll, often
Processes generated in file rundll32_parent_pstree.json
```

2. **Generated Process Tree File**:
    - Hierarchical process tree structures are saved in rundll32_parent_pstree.json.
    - Example JSON structure:

```
[
  {
    "image_file_name": "parent1.exe",
```

```
        "pid": 1234,
        "ppid": 0,
        "children": [
          {
            "image_file_name": "rundll32.exe",
            "pid": 1235,
            "ppid": 1234,
            "children": []
          }
        ]
    }
]
```


This program provides a comprehensive demonstration of the RunDLL32MaliciousParent module, allowing users to simulate complex and realistic scenarios of rundll32.exe execution with both benign and malicious characteristics. The customizable options for command ratio, count, and depth offer significant flexibility for testing and analysis in various emulation contexts.

RunDllGenerator

The RunDllGenerator class is designed to generate and categorize simulated rundll32.exe command-line activities. It creates a mix of benign**,** low-risk, and malicious commands based on a configurable ratio, making it useful for testing and research in digital forensics, malware analysis, and cybersecurity scenarios.

Key Methods:

- **generate_benign_command()**: Generates a benign rundll32.exe command with plausible arguments.

rundll32.exe fakefile.dll,FunctionName --log=C:\path\to\log.log

- **generate_low_risk_command()**: Generates a low-risk rundll32.exe command with minimal arguments.
- **generate_malicious_command()**: Generates a suspicious or potentially malicious rundll32.exe command.
- **generate_commands()**: Generates a dictionary with categorized commands (benign, low-risk, malicious) based on the specified ratio.

This class is ideal for simulating various rundll32.exe command-line activities in testing environments, helping analyze potential attack scenarios.

Demonstration Program: rundll_process_emulation.py

The **rundll_process_emulation.py** script demonstrates the functionality of the RunDllGenerator class and integrates with other modules in the **TestDataEmulator** package to emulate command-line commands and process trees with varying risk levels. The program is particularly useful for generating categorized rundll32 commands and creating JSON representations of process trees and network connections.

Features and Capabilities

1. **Generate Categorized rundll32 Commands**:
    - Divides generated commands into categories: benign, low_risk, and malicious.
    - Supports customizable ratios for these categories (e.g., "8:1:1" for 80% benign, 10% low-risk, and 10% malicious).
2. **Verbose Mode**:
    - Provides detailed output of generated commands for inspection.
3. **Process Tree Emulation**:
    - Creates process trees for benign and low-risk commands.
    - Builds detailed process hierarchies for malicious rundll32 commands.
4. **Network Connections Emulation**:
    - Generates JSON data for network connections tied to processes using benign and malicious IPs.

5. **Memory Usage Monitoring**:
    - Tracks and reports memory usage during execution using the tracemalloc library.

Command-Line Options

- **-r / --ratio**:
    - Defines the ratio of benign:low_risk:malicious commands (default: "8:1:1").
- **-v / --verbose**:
    - Enables detailed printing of generated commands by category.
- **--benign_list**:
    - Path to a file containing benign IP addresses.
- **--malicious_list**:
    - Path to a file containing malicious IP addresses.

Workflow

1. **Command Generation**:
    - Uses RunDllGenerator to generate categorized rundll32 commands based on the specified ratio.
2. **Process Tree Creation**:
    - Builds process trees for benign and low-risk commands.
    - Constructs malicious process trees with rundll32 hierarchies or standalone malicious processes.
3. **Network Connections Generation**:
    - Creates network connections tied to malicious processes using a ConnectionGenerator.
4. **File Outputs**:
    - **pstree.json**: JSON representation of the generated process trees.
    - **netstat.json**: JSON data for network connections tied to malicious processes.
5. **Memory Monitoring**:

- Tracks peak memory usage during execution for performance evaluation.

Example Run

```
$ python rundll_process_emulation.py \
  --benign_list ./ip_lists/file_google.txt \
  --malicious_list ./ip_lists/compromised_ip_full.txt \
  -r "7:2:1" \
  -v

Generated Commands by Category:
Benign Commands (7):
  rundll32.exe season.dll,government --log=/husband/alone/two.log
  rundll32.exe ability.dll,treatment --log=/key/even/significant.log
  rundll32.exe near.dll,experience --log=/position/film/smile.log
  rundll32.exe trial.dll,floor --log=/walk/first/half.log
  rundll32.exe create.dll,task --log=/at/despite/system.log
  rundll32.exe others.dll,buy --log=/girl/physical/positive.log
  rundll32.exe half.dll,few --log=/large/large/marriage.log
Low_risk Commands (2):
  rundll32.exe
  rundll32.exe
Malicious Commands (1):
  rundll32.exe->pattern.css
Processes generated in file pstree.json
Generated connections in 'netstat.json'.
Current memory usage: 0.078823 MB
Peak memory usage: 47.970995 MB
```

The rundll_process_emulation.py script is a comprehensive tool for generating categorized rundll32 commands and emulating associated process trees and network activity. It integrates various modules in the TestDataEmulator package to simulate real-world benign and malicious activities, with outputs tailored for testing and analysis.

CredentialDumpGenerator

This class generates commands for credential dumping, simulating both benign and malicious credential dump actions, which are common in advanced persistent threat (APT) attacks.

Key Methods:

- **generate_commands()**: Generates a list of benign and malicious commands.
- **generate_benign_command()**: Generates a benign command for dumping credentials.
- **generate_malicious_command()**: Generates a malicious credential dumping command.

Demonstration Program: credential_dump_emulation.py

The credential_dump_emulation.py script demonstrates the functionality of the CredentialDumpGenerator class in the TestDataEmulator package. The program generates a mix of benign and malicious credential dumping commands, creates associated process trees, and outputs them in a structured JSON format for further analysis.

Features and Capabilities

1. **Credential Dump Command Generation**:
    - Produces a specified number of credential dumping commands divided into benign and malicious categories.
    - Allows customization of the ratio between benign and malicious commands.
2. **Process Tree Generation**:
    - Builds hierarchical process trees for the generated commands.
    - Ensures realistic simulation of both benign and malicious behaviors.
3. **JSON Output**:
    - Outputs the process trees to a JSON file (credentials_pstree.json) for easy inspection and analysis.
4. **Visual Feedback**:
    - Displays generated commands on the terminal with color-coded categories:
        - **Green**: Benign commands.
        - **Red**: Malicious commands.

Command-Line Options

- **--count**:
  - Total number of commands to generate (default: 10).
- **--ratio**:
  - Specifies the ratio of benign-to-malicious commands in the format benign:malicious (default: "9:1").

Workflow

1. **Command Generation**:
   - Utilizes CredentialDumpGenerator to generate commands categorized as benign or malicious based on the specified ratio.
2. **Display Commands**:
   - Prints the generated commands on the console:
     - **Benign Commands**: Shown in green for quick identification.
     - **Malicious Commands**: Highlighted in red for differentiation.
3. **Process Tree Construction**:
   - Creates realistic process trees for each generated command using the ProcessTreeEmulator module.
4. **Output File**:
   - Saves the constructed process trees to a JSON file named credentials_pstree.json.

Example Run

```
$ python credential_dump_emulation.py --count 15 --ratio 8:2
Benign Commands:
bill.exe -param1 main model -config 9191
point.exe -param1 let -config 2017
my.exe -param1 assume crime rest -config 7726
compare.exe -param1 yard middle around -config 6265
clearly.exe -param1 thank push can -config 9739
reduce.exe -param1 nature trial myself -config 5483
structure.exe -param1 financial west -config 1510
charge.exe -param1 machine sometimes -config 5927
daughter.exe -param1 true only -config 8384
shake.exe -param1 politics -config 6436

Malicious Commands:
procdump.exe -ma lsass.exe -o \sort\age\sense.avi\lsass_dump.dmp
rundll32.exe \second\per\to.png\comsvcs.dll MiniDump 5997 lsass.dmp full
rundll32.exe \oil\process\care.css\comsvcs.dll MiniDump 1374 lsass.dmp full
procdump.exe -ma lsass.exe -o \poor\model\see.webm\lsass_dump.dmp
procdump.exe -ma lsass.exe -o \often\special\camera.gif\lsass_dump.dmp
Processes generated in file credentials_pstree.json
```

LDRModulesEmulator

This class emulates the loading of dynamic link library (DLL) modules for processes, generating entries with fake process IDs and DLL paths, simulating real-world scenarios for digital forensics testing.

Key Methods:

- **generate_module_entry()**: Generates a single module entry with random data.
- **generate_modules()**: Generates multiple module entries for all processes.
- **save_to_file()**: Saves the generated modules to a JSON file.

Example Usage:

```
# Create an LDRModulesEmulator instance with a mapping of PIDs to processes
pid_process_map = {1234: {"ImageFileName": "chrome.exe"}, 5678:
{"ImageFileName": "firefox.exe"}}
ldr_emulator = LDRModulesEmulator(pid_process_map)

# Generate and save module entries
modules = ldr_emulator.generate_modules()
ldr_emulator.save_to_file("ldr_modules.json", modules)
```

KeysEmulator

This class emulates Windows registry keys, generating random keys, paths, and values to simulate registry data for testing purposes.

Key Methods:

- **generate_entry()**: Generates a single registry entry with random values.
- **unique_name()**: Generates a unique name for each registry entry.
- **generate_data()**: Generates a list of registry entries.
- **save_to_file()**: Saves the generated registry data to a JSON file.

Example Usage:

```
# Generate and save registry key entries
key_emulator = KeysEmulator(num_entries=10)
key_emulator.generate_data()
key_emulator.save_to_file("registry_keys.json")
```

UserEmulator

This class emulates user accounts on a system, generating random user details, such as names and unique hashes, for cybersecurity testing.

Key Methods:

- **generate_unique_hash()**: Generates a unique hash for each user.
- **generate_entry()**: Generates a single user entry with random data.
- **generate_data()**: Generates a list of user entries.
- **save_to_file()**: Saves the generated user data to a JSON file.

Example Usage:

```
# Generate and save user entries
user_emulator = UserEmulator(num_users=5)
user_emulator.generate_data()
user_emulator.save_to_file("users.json")
```

Test Program Overview: users_registries_modules_emulation.py

The users_registries_modules_emulation.py script demonstrates the usage of the following classes from the **TestDataEmulator** package:

1. **LDRModulesEmulator**:
2. **KeysEmulator**:
3. **UserEmulator**:

The script processes input data, generates emulated data for these three categories, and exports the results into JSON files.

Features and Capabilities

1. Process Module Simulation (LDRModulesEmulator):

- Reads a process list from a JSON file.
- Generates a list of loaded DLL modules for each process.
- Exports the simulated modules to a JSON file (ldrmodules.json).

2. Registry Key Activity Simulation (KeysEmulator):

- Generates registry key modification events.
- Supports a configurable count of registry entries to emulate.
- Saves the simulated registry data to a JSON file (printkey.json).

3. User Data Simulation (UserEmulator):

- Simulates user data, including credentials and hashed values.
- Exports the generated user data to a JSON file (hashdump.json).

4. Memory Profiling:

- Tracks and displays memory usage during the script's execution using Python's tracemalloc library.

Command-Line Options

- **-c or --count**:
  - Specifies the number of entries to generate for the registry keys and user data.
- **-p or --processes**:
  - Path to a JSON file containing the process list, required for module generation.

Example Run

```
user@DESKTOP-OSJU4T7 MINGW64 /d/Westminster/Project/spectre/emulation
$ python users_registries_modules_emulation.py -c 5 -p ./pstree.json
ldrmodules.json has been generated.
printkey.json has been generated.
hashdump.json has been generated.
Current memory usage: 0.185411 MB
Peak memory usage: 0.243343 MB
```

IPCliEmulator

The IPCliEmulator class provides tools for emulating command-line behavior by generating realistic commands containing IP addresses and arguments. It is especially useful for testing scenarios involving benign and malicious IPs.

Key Features

1. **IP Address Loading**: Load a list of IP addresses from a file for command generation.
2. **Benign-to-Malicious Ratio Parsing**: Allows you to define and parse the ratio of benign to malicious IPs.
3. **Command Generation**: Dynamically generates commands with IPs and realistic command-line options, simulating benign or malicious behavior.

Demonstration Program: ip_commands_emulation.py

The ip_commands_emulation.py script demonstrates the functionality of the IPCliEmulator class from the TestDataEmulator package. This script generates a mix of command-line commands involving benign and malicious IPs, organizes them into process trees, and exports the data in a structured JSON format.

Features and Capabilities

1. **IP Command Generation**:
    - Reads benign and malicious IP addresses from specified files.
    - Generates command-line commands associated with these IPs.
    - Supports customization of the ratio between benign and malicious commands.
2. **Process Tree Simulation**:
    - Constructs realistic process trees for the generated commands.
    - Simulates relationships between parent and child processes.
3. **JSON Output**:

    - Outputs the process tree structure to a JSON file (ip_commands_pstree.json) for further analysis.
4. **Visual Feedback**:
    - Displays generated commands in the terminal with color-coded categories:
        - **Green**: Benign commands.
        - **Red**: Malicious commands.

Command-Line Options

- **--benign**:
    - Path to the file containing a list of benign IP addresses.
- **--malicious**:
    - Path to the file containing a list of malicious IP addresses.
- **--ratio**:
    - Ratio of benign-to-malicious commands in the format benign:malicious (default: "90:10").
- **--count**:
    - Total number of commands to generate (default: 10).

Workflow

1. **Load IPs**:
    - Reads benign and malicious IPs from the provided files using IPCliEmulator.load_ips.
2. **Parse Ratio**:
    - Converts the benign-to-malicious ratio into numerical weights using IPCliEmulator.parse_ratio.
3. **Generate Commands**:
    - Creates commands for both benign and malicious IPs using TestDataEmulator.generate_commands_with_ips.
4. **Display Commands**:
    - Prints generated commands to the console:
        - **Green** for benign commands.

- **Red** for malicious commands.

5. **Construct Process Tree**:
   - Builds a process tree for the commands using ProcessTreeEmulator.create_process_tree.
6. **Export JSON**:
   - Saves the process tree to a JSON file (ip_commands_pstree.json).

Example Run

```
$ python ip_commands_emulation.py \
  --benign ./ip_lists/file_google.txt \
  --malicious ./ip_lists/compromised_ip_full.txt \
  --ratio 80:20 \
  --count 15
Benign Ips:
whatever.exe -ip 34.1.215.167 -user david71
try.exe 34.1.208.66 -port 34462 -config /hold/nothing.conf --timeout 16
from.exe 34.1.221.176 mendoza.org
address.exe -i 34.1.213.232 -user evelyn25 --timeout 31 -port 3112
already.exe -ip 34.1.218.232 -config /suffer/lawyer.conf
deal.exe 34.1.222.8 -config /official/could.conf -user curtis07
art.exe -ip 34.1.219.12 -user johnfoley
model.exe -i 34.1.219.2 -port 52211
individual.exe 34.1.208.41 -config /figure/floor.conf --verbose
hear.exe --ipaddress 34.1.211.131 johnson.com -config /laugh/whatever.conf --verbose

Malicious Malicious:
too.exe --ipaddress 84.113.121.103 lynch.com -user danielyang
cause.exe -i 47.120.35.51 --timeout 16 --verbose -config /couple/food.conf
sea.exe 110.137.192.51 --timeout 15 -port 13490
resource.exe --ipaddress 2600:3c03::f03c:95ff:fea3:c2f --verbose
remember.exe 1.85.46.201 -port 25406
Processes generated in file ip_commands_pstree.json
```

The ip_commands_emulation.py script provides a comprehensive framework for generating, categorizing, and visualizing command-line commands associated with benign and malicious IP addresses. It simplifies the process of testing and analyzing command-line behaviors in various scenarios, making it a valuable tool for security researchers and analysts.

The Emulation Module provides a set of powerful tools to simulate malicious and benign behaviors, including process creation, network activity, credential dumping, registry key generation, user account emulation, and more. These classes help generate realistic data for security testing, research, and analysis.

To get started, ensure all necessary dependencies are installed, then use the provided classes to simulate and analyze different attack techniques and system behaviors.

Delta Module

The Delta Module is designed for analyzing and identifying differences between two memory dumps. It provides tools to compare and evaluate changes across various system entities, including connections, processes, registries, modules, and users. The module is essential for tasks like forensic analysis, system auditing, and tracking changes in system states.

MemoryDiff

The **MemoryDiff** class is the primary class within the Delta Module. It is responsible for performing detailed comparisons between two memory dumps and generating structured reports about detected changes. The comparisons cover:

- **Connections**: Network-related changes.
- **Processes**: Differences in process trees.
- **Registries**: Updates in registry keys.
- **Modules**: Variations in loaded modules.
- **Users**: Alterations in user-related data.

Features of MemoryDiff

- **Comparison of Dumps**:
    - The class can compare two memory dumps, analyzing changes in all supported categories.
- **Delta Report Generation:**
    - Provides a detailed report in JSON format.
    - Optionally outputs an HTML report for visualization.

- **Detailed Entity Diffing:**
    - Fine-grained comparison for each category (e.g., field-level analysis for connections or registries).
- **Summary Statistics:**
    - Tracks the number of added, removed, updated, and consistent entries for each entity.

Methods in MemoryDiff

compare_dumps(dump1, dump2)

- Primary method for comparing two memory dumps.
- Calls specific diffing methods for each category.
- Populates the internal state with detected changes.

generate_delta_report()

- Compiles and outputs the results of the comparison into a structured report.
- Outputs the report to delta_report.html and returns the raw JSON structure.

Diffing Methods

- **diffConnections(old_connections, new_connections)**: Compares connection data, identifying added, removed, updated, and consistent connections.
- **diffProcesses(tree1, tree2)**: Evaluates differences in process trees, including new, removed, and updated nodes.
- **diffRegistries(old_registries, new_registries)**: Analyzes changes in registry key values and metadata.
- **diffModules(old_modules, new_modules)**: Tracks changes in loaded modules for processes.
- **diffUsers(old_users, new_users)**: Detects additions, removals, and updates in user information.

Example: Using MemoryDiff

Initialization

```
from DeltaModule import MemoryDiff


# Create an instance of MemoryDiff
memory_diff = MemoryDiff()
```

Comparing Memory Dumps

```
# Assume dump1 and dump2 are pre-loaded memory dump objects with required data
structures
memory_diff.compare_dumps(dump1, dump2)
```

Generating a Delta Report

```
# Generate a detailed report and save it to an HTML file
report = memory_diff.generate_delta_report()

# Access the raw JSON structure of the report
print(report)
```

Sample Workflow

Load Dumps:

- Load memory dump data from JSON or other supported formats.

Initialize MemoryDiff:

- Create an instance of the class to perform comparisons.

Run Comparisons:

- Use the compare_dumps method to analyze changes between two dumps.

Export Results:

- Call generate_delta_report to save and access the results.

Example Output: Delta Report (JSON)

```
{
    "connection_updates": {
        "added": [ ... ],
        "removed": [ ... ],
        "updated": [ ... ],
        "consistent": [ ... ]
    },
    "process_updates": [
        {
            "pid": 1234,
            "relationship": "Parent",
            "change_type": "Updated",
            "old_value": "Handles: 50",
            "new_value": "Handles: 75"
        },
        ...
    ],
    "registry_updates": {
        "added": [ ... ],
        "removed": [ ... ],
        "updated": [ ... ],
        "consistent": [ ... ]
    },
    "modules_updates": {
        "added": [ ... ],
        "removed": [ ... ],
        "updated": [ ... ],
        "consistent": [ ... ]
    },
    "user_updates": {
        "added": { ... },
        "removed": { ... },
        "updated": { ... },
```

```
        "consistent": { ... }
    }
}
```

The **MemoryDiff** class in the Delta Module provides a robust framework for analyzing differences between memory dumps. With its comprehensive diffing methods and reporting capabilities, it serves as an essential tool for system analysis and forensic investigations.

DeltaAnalysis

The DeltaAnalysis class provides an easy-to-use interface for visualizing changes and comparisons between memory dumps. It leverages the data from a memoryDiff object to create insightful plots for various updates and comparisons. Below is a step-by-step guide to get started with using this class:

Prerequisites

Make sure the following prerequisites are satisfied:

- You have the matplotlib library installed (pip3 install matplotlib).
- You have a MemoryDiff object containing the required data. The MemoryDiff object should have structured data for process, connection, module, registry, and user updates.

Instantiating the DeltaAnalysis Class

```
from delta_analysis import DeltaAnalysis


# Assume `memoryDiff` is an object with all the required data.
delta_analysis = DeltaAnalysis(memoryDiff)
```

The memoryDiff object should include fields like:

- process_updates, connection_updates, etc.
- Attributes for added, removed, updated, and consistent counts.

Visualizing Updates

The DeltaAnalysis class provides multiple plotting methods, each designed to visualize specific aspects of the memory differences. Below are the methods and how to use them:

Process Updates

Visualize the changes in processes between memory dumps:

```
delta_analysis.plotProcessUpdates()
```

Output: A bar chart showing counts of:

- New processes
- Removed processes
- Updated processes
- Consistent processes

Connection Updates

Visualize the changes in network connections between memory dumps:

```
delta_analysis.plotConnectionUpdates()
```

Output: A bar chart showing counts of:

- New connections
- Removed connections
- Updated connections

- Consistent connections

Modules Updates

Visualize changes in loaded modules:

delta_analysis.plotModulesUpdates()

Output: A bar chart displaying:

- New modules
- Removed modules
- Updated modules
- Consistent modules

Registry Updates

Visualize changes in the registry entries:

delta_analysis.plotRegistriesUpdates()

Output: A bar chart displaying:

- New registry entries
- Removed registry entries
- Updated registry entries
- Consistent registry entries

User Updates

Visualize changes in user accounts:

delta_analysis.plotUserUpdates()

Output: A bar chart displaying:

- New users
- Removed users
- Updated users
- Consistent users

Combined Updates

Visualize a comprehensive overview of updates for processes, connections, modules, registry keys, and users:

delta_analysis.plotCombinedUpdates()

Output: A grid of subplots providing a detailed comparison for:

- Processes
- Connections
- Modules
- Registry entries
- User accounts

Each subplot will highlight the added, removed, updated, and consistent counts.

Example Workflow

Below is a sample workflow that uses the DeltaAnalysis class:

```
# Initialize the DeltaAnalysis object
delta_analysis = DeltaAnalysis(memoryDiff)


# Plot process updates
delta_analysis.plotProcessUpdates()


# Plot connection updates
delta_analysis.plotConnectionUpdates()
```

```
# Plot combined updates
delta_analysis.plotCombinedUpdates()

# Plot specific updates
delta_analysis.plotModulesUpdates()
delta_analysis.plotRegistriesUpdates()
delta_analysis.plotUserUpdates()
```

Notes

- Each plotting method will display the chart using Matplotlib. You may need to close the plot window to proceed with the next visualization.
- The memoryDiff object must provide the necessary data attributes for the class to function correctly.

By following this guide, you can easily explore and visualize memory dump differences using the DeltaAnalysis class.

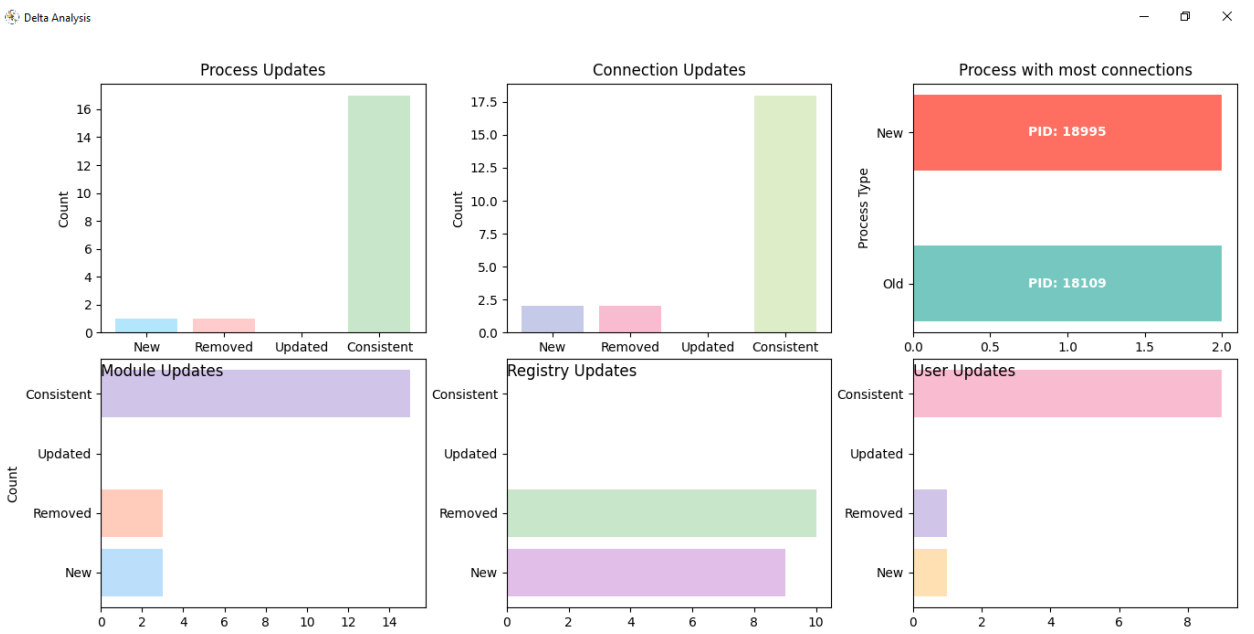
Delta Demonstration Program

The delta_demo.py program performs a delta analysis between two memory dumps to identify and visualize changes. It uses command-line arguments to specify directories for the memory dumps and leverages Memory and Delta modules to load, compare, and analyze the dumps. Key features include:

1. Memory Dump Loading: Parses JSON files from the provided directories.
2. Delta Analysis: Identifies changes in processes, connections, modules, and registries.
3. Visualization: Generates plots for combined updates and other metrics.
4. Memory Tracking: Reports peak memory usage during execution.

Example Run

```
$ python ./delta_demo.py -d1 ../emulation/10 -d2 ../emulation/10/delta/
Peak memory usage: 7.347563 MB
```



The program outputs visualizations and reports memory usage for efficient debugging and system change tracking.

Visualization Module

The VisualizationModule provides specialized function for creating visual representations of various data types, helping analysts identify patterns and anomalies effectively. This guide details how to use the displayExtensions function and mentions that other visualization tasks, such as memory, delta, and timeline analyses, are handled in their respective modules.

Function: displayExtensions

The displayExtensions function visualizes file extension usage as a horizontal bar chart, which is particularly useful for analyzing unsafe or suspicious file types.

Parameters

- **extension_counts** *(defaultdict)*: A dictionary-like object where:
  - Keys are file extensions (e.g., jpg, png).
  - Values are the counts of those extensions.

  Example:

  defaultdict(<class 'int'>, {'bmp': 4, 'jpg': 2, 'png': 3})

- **axis** *(matplotlib.axes._axes.Axes, optional)*: A matplotlib axis object. If provided, the plot will render on the given axis. If not, a new figure is created.

Using displayExtensions

1. Standalone Usage

To generate a new plot with file extension counts:

```
from collections import defaultdict
from VisualizationModule import displayExtensions

# Example data
extension_counts = defaultdict(int, {'exe': 10, 'dll': 8, 'bat': 5, 'js': 3})

# Generate the bar chart
displayExtensions(extension_counts)
```

This produces a horizontal bar chart with extensions on the y-axis and counts on the x-axis.

2. Embedding in Existing Plots

To include the chart within an existing subplot:

```
import matplotlib.pyplot as plt
from collections import defaultdict
from VisualizationModule import displayExtensions
```

```
# Example data
extension_counts = defaultdict(int, {'exe': 10, 'dll': 8, 'bat': 5, 'js': 3})

# Create a figure and axis
fig, ax = plt.subplots()

# Generate the plot on the existing axis
displayExtensions(extension_counts, axis=ax)

# Show the figure
plt.show()
```

This integrates the visualization into a broader dashboard or report.

Key Features

- **Dynamic Sizing:** Automatically adjusts to fit the data and display area.
- **Customizable Appearance:** Modify the color palette, axis labels, or grid styles.
- **Axis Integration:** Seamlessly integrates into subplots for dashboards.

MemoryAnalysis classes uses displayExtensions to plot process extensions in a subplot.

ScatterPlotter

The ScatterPlotter class is designed to generate scatter plots that visually represent process activity over a timeline. Each process is displayed as a point, with additional context such as the number of connections and malicious indicators provided interactively.

Initialization

To begin using the ScatterPlotter class, initialize it with a list of process data. Each process in the list should be a dictionary with the following keys:

- **name**: (str) The name of the process.
- **PID**: (int) The Process ID.
- **connections**: (int) The number of connections the process has.
- **is_malicious**: (bool) Whether the process is malicious (True) or benign (False).
- **creation_time**: (datetime) The time the process was created.
- **malicious_indicators**: (list) A list of strings describing malicious behaviors (optional for benign processes).

Example:

```
from datetime import datetime
process_data = [
    {
        "name": "malware.exe",
        "PID": 5678,
        "connections": 25,
        "is_malicious": True,
        "creation_time": datetime(2024, 12, 28, 14, 45),
        "malicious_indicators": ["Suspicious Parent", "Abnormal Network Activity"]
    },
]
scatter_plotter = ScatterPlotter(process_data)
scatter_plotter.plot()
```

Features of the Scatter Plot

1. **Visual Representation**
    - X-axis: Timeline (creation time of the process).
    - Y-axis: Number of connections associated with each process.
    - Color-coded points:
        - **Red**: Malicious processes.
        - **Grey**: Non-malicious processes.

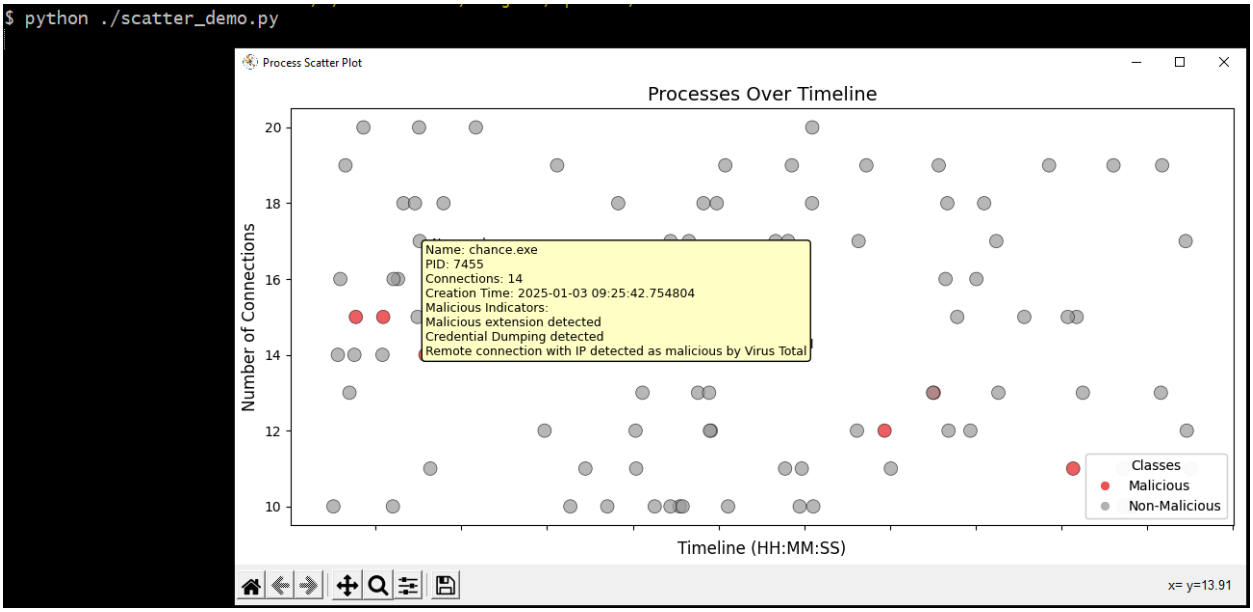
2. **Hover Interactivity**
    - Hovering over a point displays detailed information about the process:
        - Name.
        - PID.
        - Number of connections.
        - Creation time.
        - Malicious indicators (for malicious processes).
3. **Dynamic Tick Interval**
    - The timeline dynamically adjusts tick intervals based on the range of creation times, ensuring clarity.
4. **Scatter Point Size**
    - Points are uniformly sized for simplicity but can be customized.

The hover functionality displays annotations and comment boxes with information about processes. This behavior is automatically enabled when the mouse pointer moves over the scatter plot.

ScatterPlotter demonstration program: scatter_demo.py

This demonstration program uses the Faker library to create realistic-looking process names and it randomly assigns malicious or benign status to processes with weighted probabilities.

Example Run:



Timeline Analysis Module

Timeline Analysis module provides classes to visualize memory changes across multiple snapshots. It also provides a TimelinePlotter class which plots processes and connections from a single dump over the timeline.

TimelinePlotter

The TimelinePlotter class is designed to plot timelines for process creations and network connections, visualizing them from JSON files. It can either read from provided JSON files or generate random data for demonstration purposes. The class offers the following key functionalities:

1. **Loading Data**: It loads process creation and network connection data from specified JSON files.
2. **Parsing Process Data**: It recursively extracts process creation timestamps, including those from child processes.
3. **Plotting Timelines**: It plots timelines for both process creations and network connections on the same graph.
4. **Aggregating Data**: It aggregates the events by minute and plots the counts over time for both processes and network connections.

5. **Top Processes by Connections**: It generates stacked bar charts showing the top processes with the most connections, organized by protocol (TCPv4, TCPv6, UDPv4, UDPv6).
6. **Combined Timeline Plot**: It creates a combined plot with the connections/process timeline along with the top processes per timeslot.

Demo Program: process_connection_plot.py

The demo program demonstrates how to use the TimelinePlotter class to plot various visualizations. Here's the breakdown:

Program Features:

1. **Command-line Arguments**: The program accepts command-line arguments for specifying the paths to process tree (pstree-file) and network connections (connections-file) JSON files.
2. **Memory Usage Tracing**: The program starts by tracing memory allocation using tracemalloc and prints the peak memory usage after execution.
3. **Plotting**: It calls the plotting methods of TimelinePlotter to visualize:
    - Timelines for process creations and network connections.
    - Top processes with the most network connections.
    - Combined timeline with connections and top processes for each timeslot.

Demo Steps:

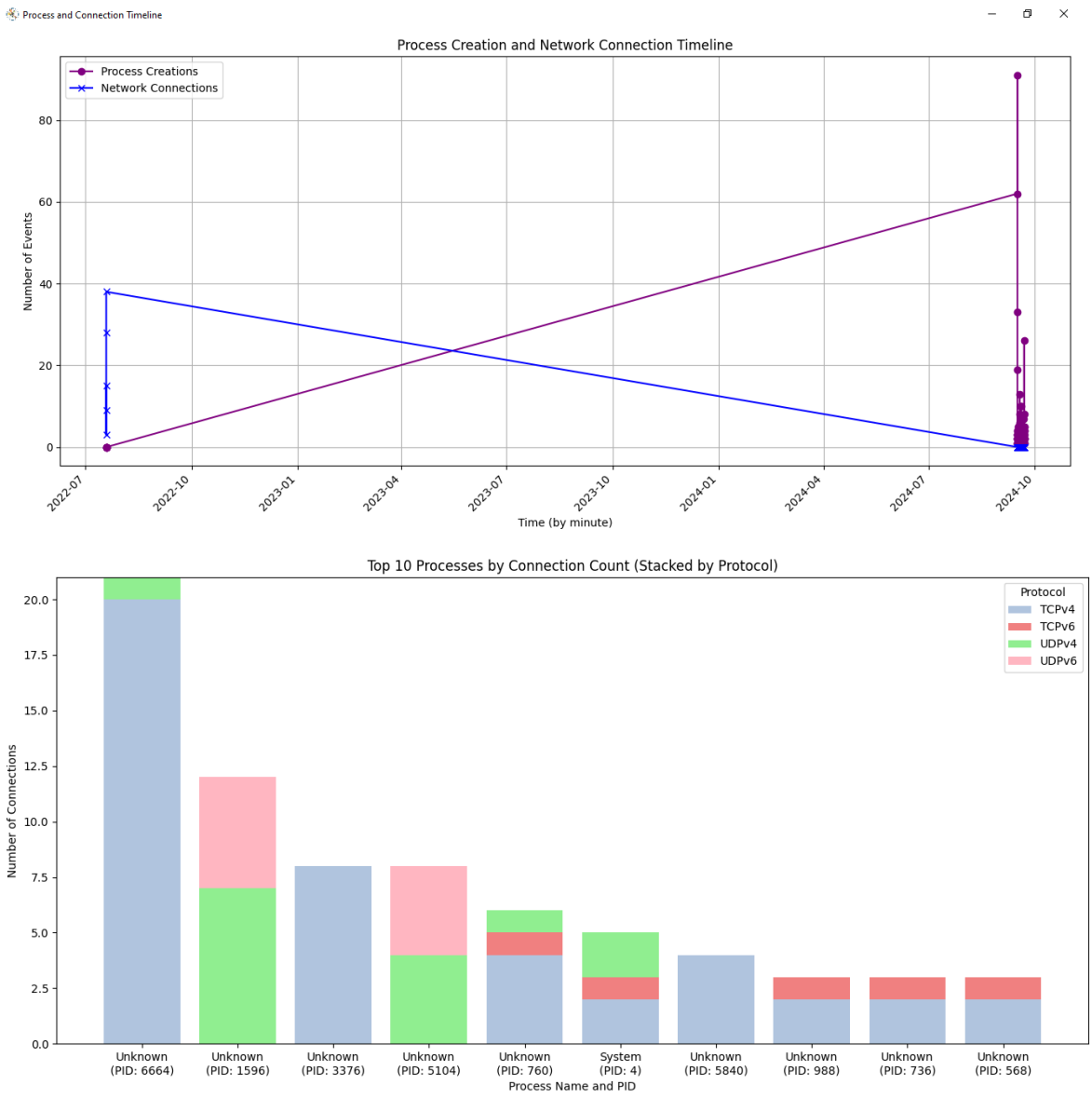
1. **Usage**: Run the program with the required arguments as follows:

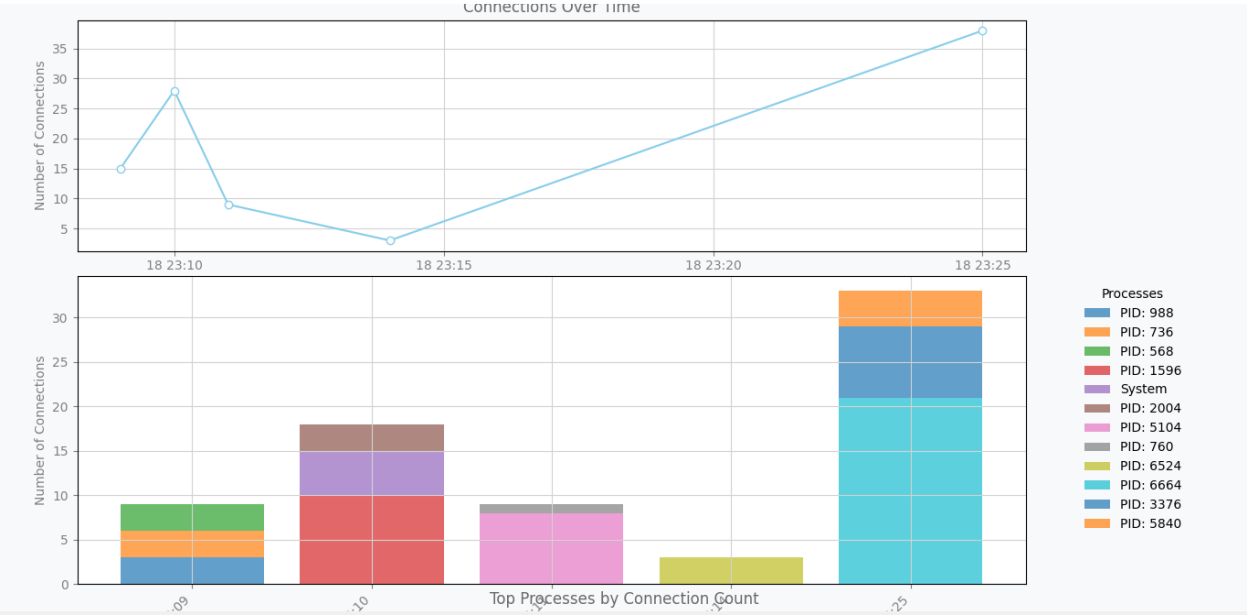
```
$ python process_connection_plot.py --pstree-file <path_to_pstree_file> --connections-file <path_to_connections_file>
```

2. **Outputs**:
    - Timeline graphs for process creations and network connections.
    - Stacked bar charts for the top processes with the most connections.
    - Combined plot of connections and top processes per time slot.

Example Run

```
$ python ./process_connection_plot.py \
      --pstree-file  ../output/pstree.json \
      --connections-file ../output/netstat.json
Peak memory usage: 8.025253 MB
```





ProcessTimeLineAnalysis

ProcessTimeLineAnalysis is designed to analyze and compare consecutive process trees from JSON files, identifying changes over time such as new, removed, updated, or consistent processes. It generates visual plots to highlight the differences.

Key Features of the Class

1. **Initialization:**

   Takes a list of JSON file paths for processing.

```
def __init__(self, file_list: List[str]):
```

2. **Methods**:
   - **read_json_from_file(filename)**: Reads the content of a JSON file.
   - **compute_differences(tree1, tree2)**: Compares two process trees, identifying:
     - New processes
     - Removed processes

        - Updated processes
        - Consistent processes
    - **plot_comparison_results(axis=None)**: Visualizes the differences in four categories (Added, Removed, Updated, Consistent) across multiple timepoints.
    - **compare_files()**:
        - Compares consecutive files in the provided list.
        - Parses JSON content into process tree structures.
        - Computes differences and stores results for plotting.

Demo Program: pstree_timeline_demo.py

The script compares process trees from multiple JSON files provided via command-line arguments and plots the results.

Usage

```
python pstree_timeline_demo.py \
  --file_list file1.json,file2.json,file3.json
```

Workflow

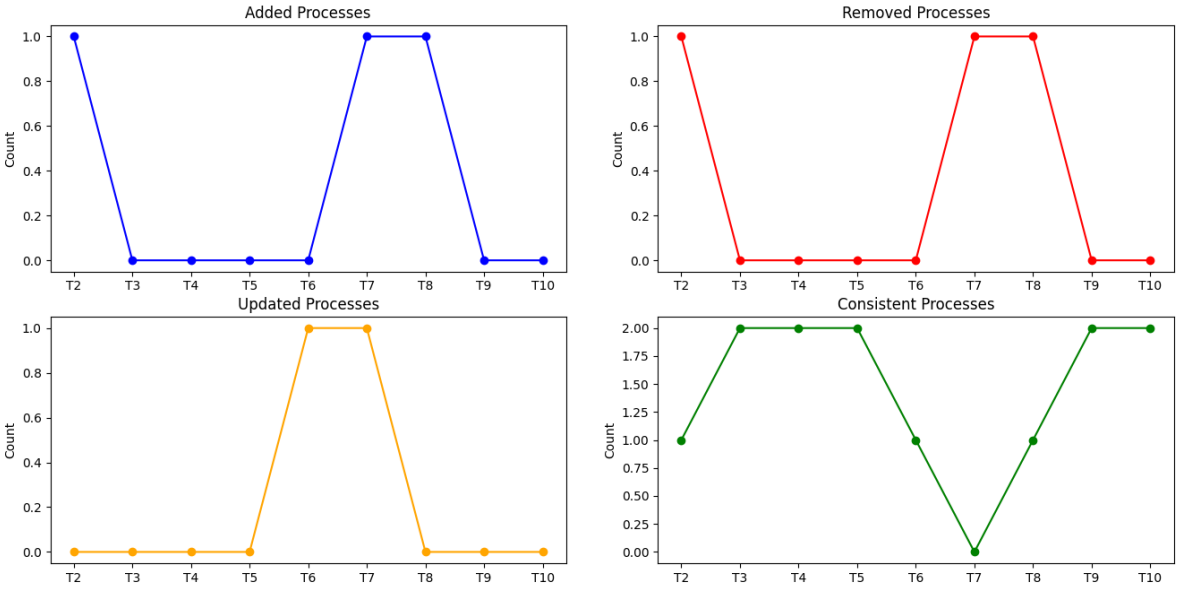
1. **Input**: A comma-separated list of JSON files via --file_list.
2. **Initialization**:

   analysisObject = ProcessTimeLineAnalysis(file_list)

3. **Comparison**: Consecutive JSON files are compared using compare_files():
    - Differences (added, removed, updated, consistent) are calculated.
    - Results are visualized using plot_comparison_results.
4. **Output**: Visual plots highlighting changes in process trees over time.

```
python ./pstree_timeline_demo.py -files
./pstree_test_data/pstree_1.json,./pstree_test_data/pstree_2.json,./pstree_test_data/pstree_3.json,./pst
ree_test_data/pstree_4.json,./pstree_test_data/pstree_5.json,./pstree_test_data/pstree_6.json,./pstree
```

```
_test_data/pstree_7.json,./pstree_test_data/pstree_8.json,./pstree_test_data/pstree_9.json,./pstree_test_data/pstree_10.json
```



ModulesTimeLineAnalysis

ModulesTimeLineAnalysis identifies and visualizes changes in module load records over time by comparing consecutive JSON files. It provides insights into added, removed, updated, and consistent modules and generates detailed plots for comparison.

Key Features of the Class

1. **Initialization**:
   Initializes with a list of JSON file paths for analysis.

```
def __init__(self, file_list: List[str]):
```

2. **Methods**:
   - **compute_differences(old_modules, new_modules)**:
     - Compares two lists of modules and identifies:
       - **Added**: Modules in the new list but not in the old list.
       - **Removed**: Modules in the old list but missing in the new list.

        - **Updated**: Modules with changes in key fields.
        - **Consistent**: Modules unchanged between lists.
    - **plot_comparison_results()**:
        - Visualizes differences across comparisons in four subplots:
            - Added, Removed, Updated, and Consistent modules.
    - **compare_files()**:
        - Reads and compares consecutive JSON files in the input list.
        - Stores computed differences.
        - Generates comparative plots.

Demo Program: ldrmodules_timeline_demo.py

The script compares module load records from multiple JSON files and optionally generates random JSON files for testing.

Usage

1. **Run with JSON files for comparison**:

```
$ python ldrmodules_timeline_demo.py \
  --file_list file1.json,file2.json,file3.json
```

2. **Generate random JSON files for testing**:

```
$ python ldrmodules_timeline_demo.py --generate
```

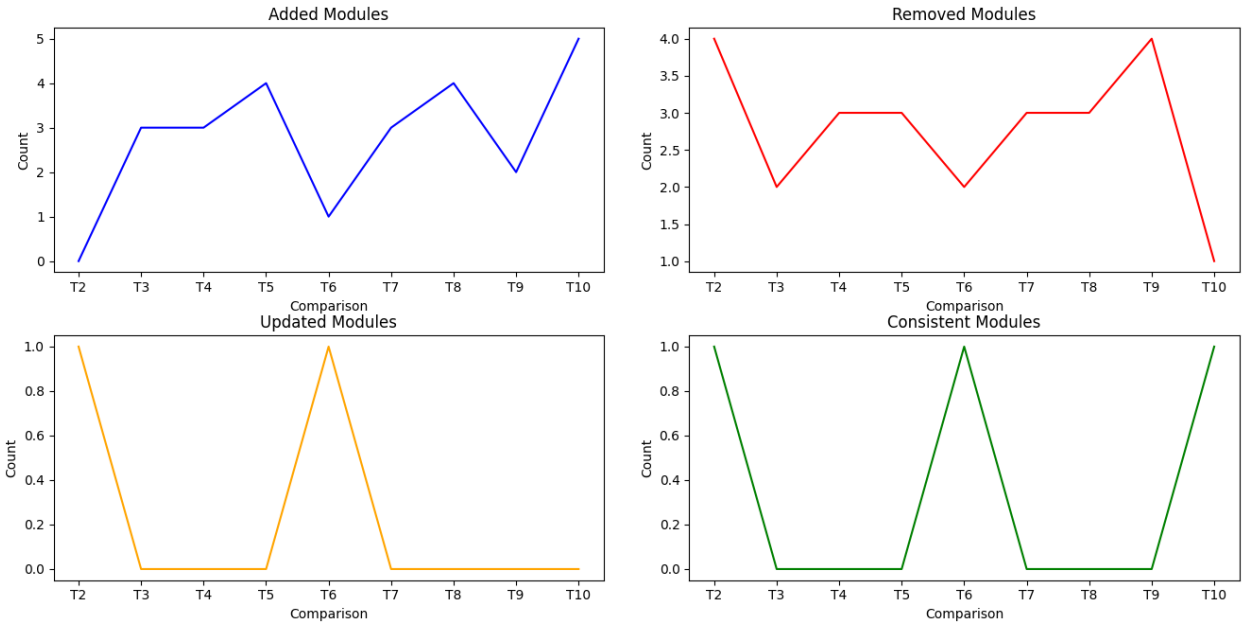
Workflow

1. **Input**:
    - List of JSON files (--file_list) or option to generate random files (--generate).
2. **Analysis**:

- For each pair of consecutive files:
  - Computes differences using compute_differences().
  - Records added, removed, updated, and consistent modules.
- Plots the results with plot_comparison_results().

3. **Output**:
   - Comparative plots showing module changes over time.

```
python ./ldrmodules_timeline_demo.py -files
./ldrmodules_test_data/ldrmodules_1.json,./ldrmodules_test_data/ldrmodules_2.json,.
/ldrmodules_test_data/ldrmodules_3.json,./ldrmodules_test_data/ldrmodules_4.json,./
ldrmodules_test_data/ldrmodules_5.json,./ldrmodules_test_data/ldrmodules_6.json,./l
drmodules_test_data/ldrmodules_7.json,./ldrmodules_test_data/ldrmodules_8.json,./l
drmodules_test_data/ldrmodules_9.json,./ldrmodules_test_data/ldrmodules_10.json
```



ConnectionTimeLineAnalysis

ConnectionTimeLineAnalysis analyzes and visualizes changes in network connections over time by comparing JSON snapshots of network activity. The class identifies added, removed, updated, and consistent connections and produces comparative plots for analysis.

Key Features of the Class

1. **Initialization**:
    - Accepts a list of JSON file paths for analysis.

```
def __init__(self, file_list: List[str]):
```

2. **Methods**:
    - **compute_connection_differences(old_connections, new_connections)**:
        - Compares two lists of network connections and identifies:
            - **Added**: Connections in the new list but missing in the old list.
            - **Removed**: Connections in the old list but absent in the new list.
            - **Updated**: Connections with changes in monitored fields.
            - **Consistent**: Connections unchanged between snapshots.
        - **Monitored Fields**: Includes fields such as Created, ForeignAddr, LocalPort, Proto, and State.
    - **plot_comparison_results()**:
        - Visualizes the analysis results in four subplots:
            - Added, Removed, Updated, and Consistent connections.
    - **compare_connection_files()**:
        - Reads and compares consecutive JSON files in the input list.
        - Computes differences and stores them for visualization.
        - Generates plots summarizing changes across all comparisons.

Demo Program: netstat_timeline_demo.py

The program compares network connection snapshots from multiple JSON files and optionally generates random JSON files for testing purposes.

Usage

1. **Run with JSON files for comparison**:

```
$ python netstat_timeline_demo.py \
        --file_list snapshot1.json,snapshot2.json,snapshot3.json
```

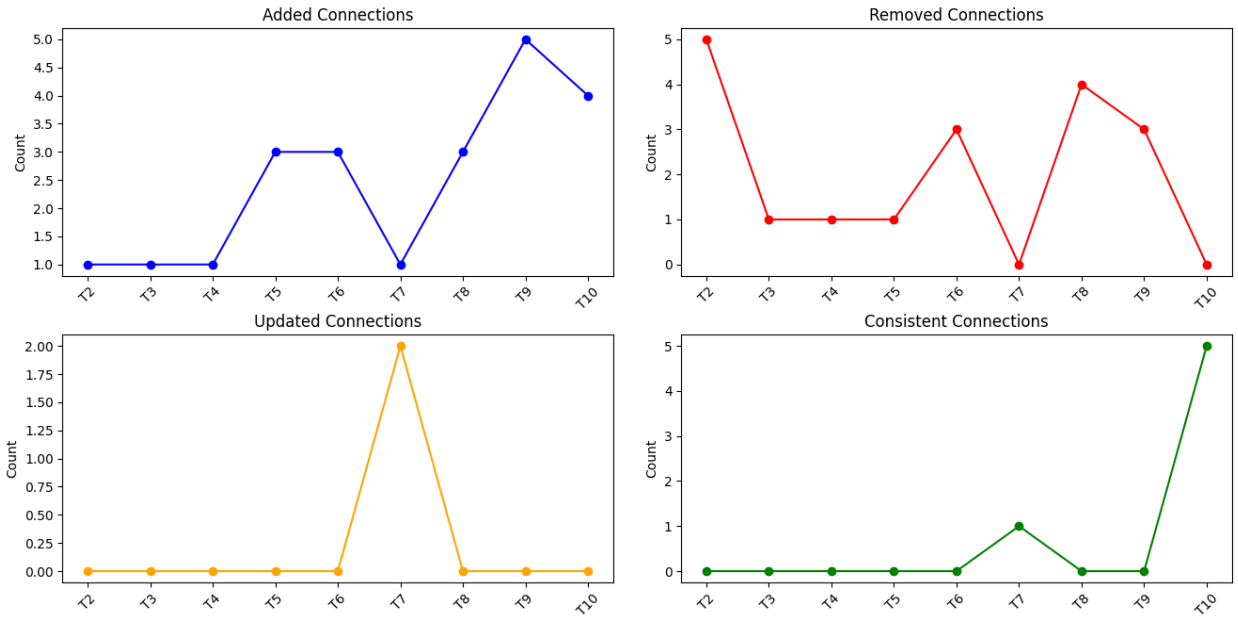
2. **Generate random JSON files for testing**:

```
$ python netstat_timeline_demo.py --generate
```

Workflow

1. **Input**:
   - List of JSON files (--file_list) or an option to generate random files (--generate).
2. **Analysis**:
   - For each pair of consecutive JSON snapshots:
     - Compares network connections using compute_connection_differences().
     - Categorizes connections into added, removed, updated, and consistent.
   - Records and visualizes the results.
3. **Output**:
   - Comparative plots showing changes in network connections over time.

```
$ python ./netstat_timeline_demo.py -files
./netstat_test_data/netstat_1.json,./netstat_test_data/netstat_2.json,./netstat_test_dat
a/netstat_3.json,./netstat_test_data/netstat_4.json,./netstat_test_data/netstat_5.json,./
netstat_test_data/netstat_6.json,./netstat_test_data/netstat_7.json,./netstat_test_data/
netstat_8.json,./netstat_test_data/netstat_9.json,./netstat_test_data/netstat_10.json
```



RegistryTimelineAnalysis

RegistryTimelineAnalysis analyzes changes in Windows Registry snapshots over time by comparing JSON files. It identifies added, removed, updated, and consistent registry entries, and visualizes the results in a comprehensive plot.

Key Features of the Class

1. **Initialization**:
    - Accepts a list of JSON file paths.

```
def __init__(self, file_list: List[str]):
```

2. **Methods**:
    - **parse_json(json_string)**:
        - Converts a JSON string into a list of dictionaries representing registry entries.
    - **read_json_from_file(filename)**:
        - Reads JSON data from a file.
    - **compute_differences(old_entries, new_entries)**:
        - Compares two lists of registry entries and categorizes them:

        - **Added**: New entries in the second snapshot.
        - **Removed**: Entries in the first snapshot but missing in the second.
        - **Updated**: Entries that have changed fields.
        - **Consistent**: Unchanged entries.
    - **Monitored Fields**: Fields such as Data, Key, Last Write Time, Type, and Name.
- **plot_comparison_results()**:
    - Visualizes the analysis results in four subplots:
        - Added, Removed, Updated, and Consistent registry keys.
- **compare_files()**:
    - Compares consecutive JSON files in the list.
    - Computes differences and stores them for visualization.
    - Generates plots summarizing changes across all comparisons.

Demo Program: printkey_time_demo.py

This program analyzes registry changes over time by comparing multiple JSON snapshots or generates random JSON files for testing.

Usage

1. **Run with JSON files for comparison**:

```
$ python printkey_time_demo.py \
    --file_list snapshot1.json,snapshot2.json,snapshot3.json
```

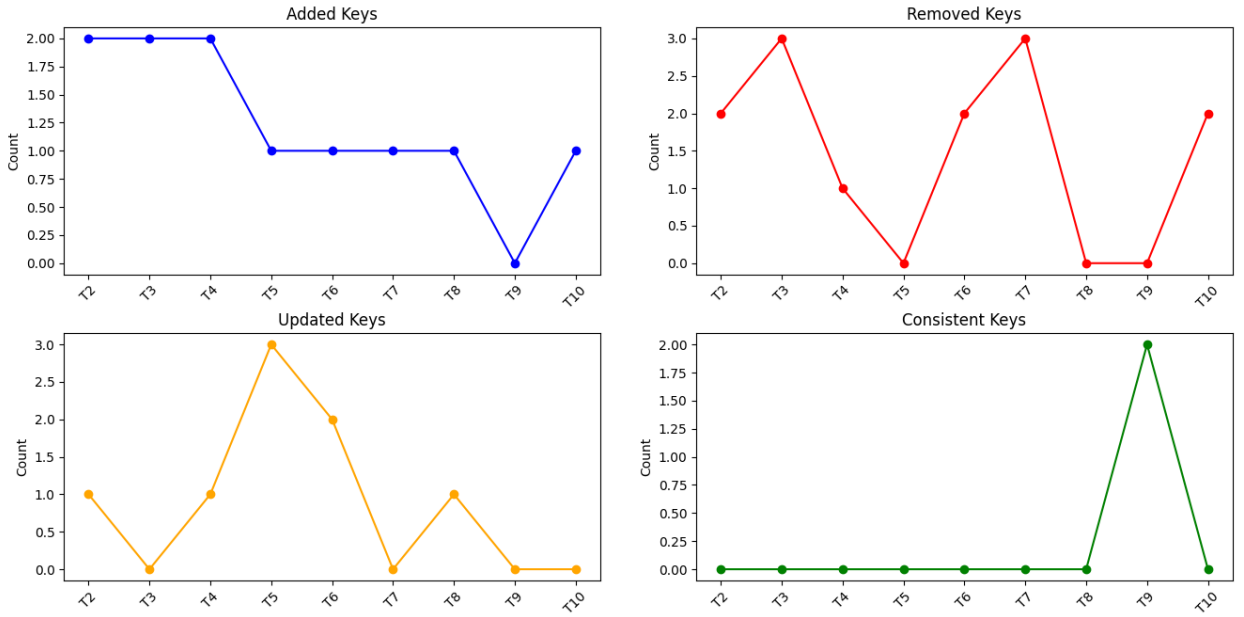
2. **Generate random JSON files for testing**:

```
$ python printkey_time_demo.py --generate
```

Workflow

1. **Input**:
    - List of JSON files (--file_list) or an option to generate random files (--generate).
2. **Analysis**:
    - For each pair of consecutive JSON snapshots:
        - Compares registry entries using compute_differences().
        - Categorizes entries into added, removed, updated, and consistent.
    - Stores results for visualization.
3. **Visualization**:
    - Creates four subplots showing:
        - Count of added, removed, updated, and consistent keys over time.

```
$ python ./printkey_timeline_demo.py -
files ./printkey_test_data/printkey_1.json,./printkey_test_data/printkey_2.json,./printkey_test_data/printkey_3.json,./printkey_test_data/printkey_4.json,./printkey_test_data/printkey_5.json,./printkey_test_data/printkey_6.json,./printkey_test_data/printkey_7.json,./printkey_test_data/printkey_8.json,./printkey_test_data/printkey_9.json,./printkey_test_data/printkey_10.json
```



Utils and OS Modules

Utils Module

The utils module provides essential classes for handling memory forensics operations and managing the output of analysis. These utilities simplify tasks such as executing Volatility commands and processing JSON results.

VolatilityWrapper

The VolatilityWrapper class is a helper for running Volatility commands on memory dump files.

Initialization

```
from utils import VolatilityWrapper

# Initialize with the memory dump file path and Volatility framework path
vol_wrapper = VolatilityWrapper(memDumpFile="memory_dump.raw",
volatility="volatility3.py")
```

Key Method

- **run_command(command: str) -> str**: Executes a Volatility command and returns the raw output as a string.

  Example:

  ```
  raw_output = vol_wrapper.run_command("windows.pstree")
  print(raw_output)
  ```

OutputHandler

The OutputHandler class processes and saves the JSON output from memory analysis.

Initialization

```
from utils import OutputHandler


# Initialize with the desired output folder path
output_handler = OutputHandler(output_folder="output/")
```

Key Methods

1. **clean_json_output(raw_json: str) -> str**: Cleans raw JSON output, making it more readable.

```
cleaned_json = output_handler.clean_json_output(raw_output)
print(cleaned_json)
```

2. **save_output(command: str, output_data: str)**: Saves cleaned JSON output to a file.

```
output_handler.save_output("pstree", cleaned_json)
```

3. **write_json_output(command: str, json_output: dict)**: Saves a Python dictionary as a formatted JSON file.

```
output_handler.write_json_output("pstree", json.loads(cleaned_json))
```

OSModule

The osmodule provides classes for interacting with different operating systems and extracting forensic data from memory dumps.

OSInterface

OSInterface is an abstract base class (ABC) defining methods for retrieving various data types from memory dumps.

Key Methods

These methods must be implemented in subclasses:

- **extractProcesses()**: Retrieves process information.
- **extractConnections()**: Retrieves network connections.
- **extractModules()**: Retrieves loaded modules.
- **extractLibraries()**: Retrieves libraries (e.g., DLLs).
- **extractRegistries()**: Retrieves registry data.
- **extractUsers()**: Retrieves user information.

WindowsInterface

WindowsInterface is a concrete implementation of OSInterface for Windows systems.

Initialization

```
from utils import VolatilityWrapper, OutputHandler
from osmodule import WindowsInterface

# Create VolatilityWrapper and OutputHandler instances
vol_wrapper = VolatilityWrapper(memDumpFile="memory_dump.raw",
volatility="volatility3.py")
output_handler = OutputHandler(output_folder="output/")

# Initialize WindowsInterface
windows_interface = WindowsInterface(volatility_wrapper=vol_wrapper,
output_handler=output_handler
```

Key Methods

1. **extractProcesses()**

   Extracts processes from the memory dump:

   ```
   processes = windows_interface.extractProcesses()
   ```

```
print(processes)
```

2. **extractConnections()**

   Retrieves network connections and applies a delta analysis:

```
connections = windows_interface.extractConnections()
print(connections)
```

3. **extractModules()**

   Extracts loaded modules:

```
modules = windows_interface.extractModules()
print(modules)
```

4. **extractLibraries()**

   Retrieves a list of DLLs:

```
libraries = windows_interface.extractLibraries()
print(libraries)
```

5. **extractRegistries()**

   Extracts registry data:

```
registries = windows_interface.extractRegistries()
print(registries)
```

6. **extractUsers()**

   Retrieves user information:

```
users = windows_interface.extractUsers()
print(users)
```

Notes

- The OSInterface class can be extended for other operating systems by implementing its abstract methods.
- Each method in WindowsInterface automatically cleans and saves the output JSON for later use.

Example Program for Utils and OS Module

The spectre_demo.py program demonstrates the integration and functionality of several key components for memory forensics, including WindowsInterface, VolatilityWrapper, and OutputHandler. It automates the process of extracting forensic data from a memory dump, analyzing it, and visualizing the results.

Working Steps

1. **Initialization**
   The program initializes the core components:
   - A VolatilityWrapper to execute Volatility commands for analyzing memory dumps.
   - An OutputHandler to process, clean, and save output files as JSON.
2. **Data Extraction**: Using the WindowsInterface class, the script executes various Volatility commands to extract:
   - Process trees
   - Network connections
   - Loaded modules
   - Registry keys
   - User account information
3. **Data Loading**: The extracted JSON data is loaded into a MemoryDump object, which serves as a structured representation of the data for further analysis.
4. **Analysis and Visualization**

- **Statistical Analysis**: The script uses MemoryAnalysis to generate statistical insights into processes, connections, and other data.
- **Timeline Visualization**: With the help of TimelinePlotter, timelines of processes and connections are plotted, showing their creation and activity over time.

5. **Delta Analysis (Optional)**: The script demonstrates how changes between two memory dumps can be compared using MemoryDiff, allowing the identification of added, removed, or modified processes, connections, and user accounts. Differences are visualized using DeltaAnalysis.
6. **Execution**

   The program is designed to be run via the command line, with arguments specifying the memory dump file, output directory, and the path to Volatility.

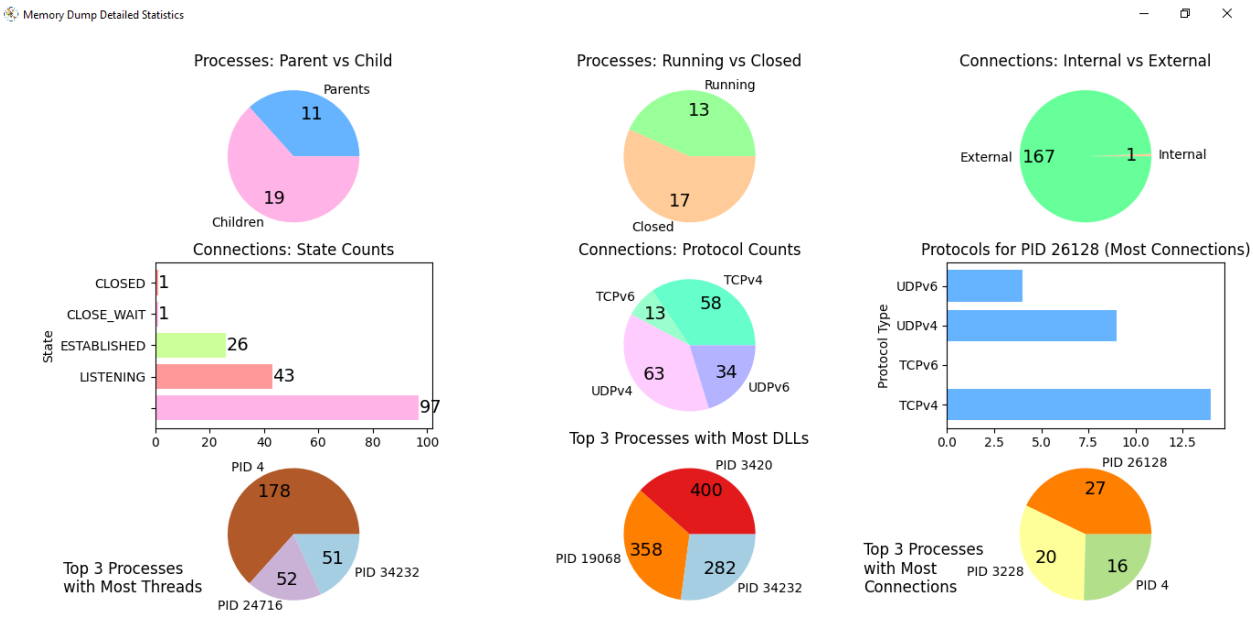
Key Outputs

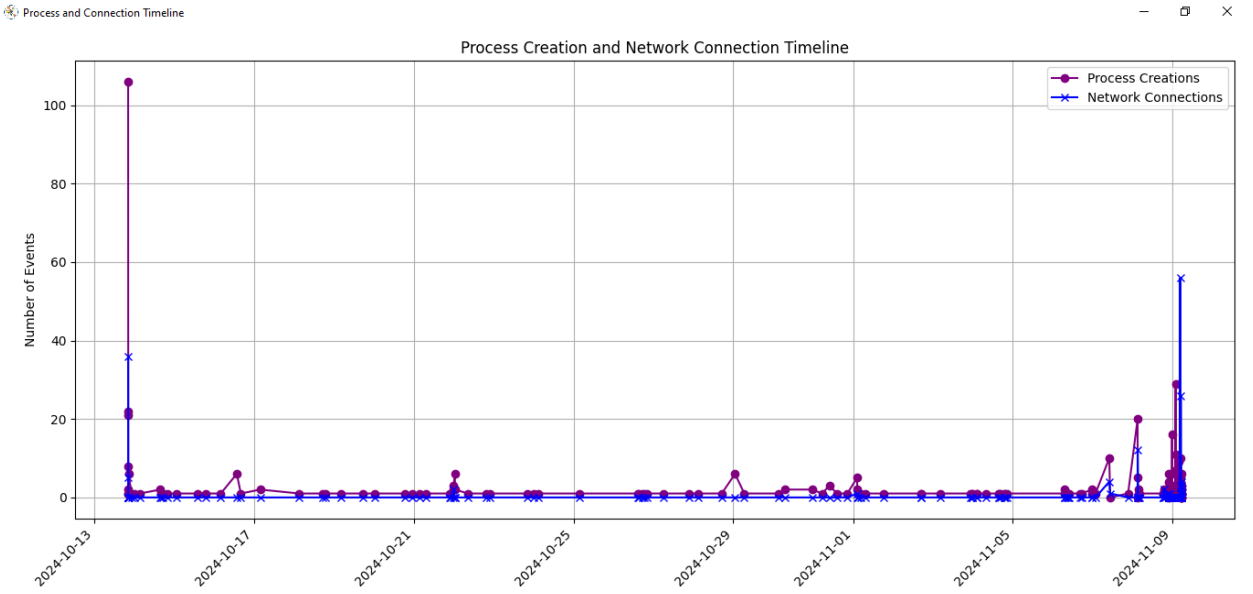
- JSON files containing structured data for processes, connections, and other artifacts. These files can be used as intermediate format of memory and any additional processings.
- Visualizations such as:
  - Detailed statistics
  - Timelines of activity
  - Comparative analysis showing changes between memory dumps.

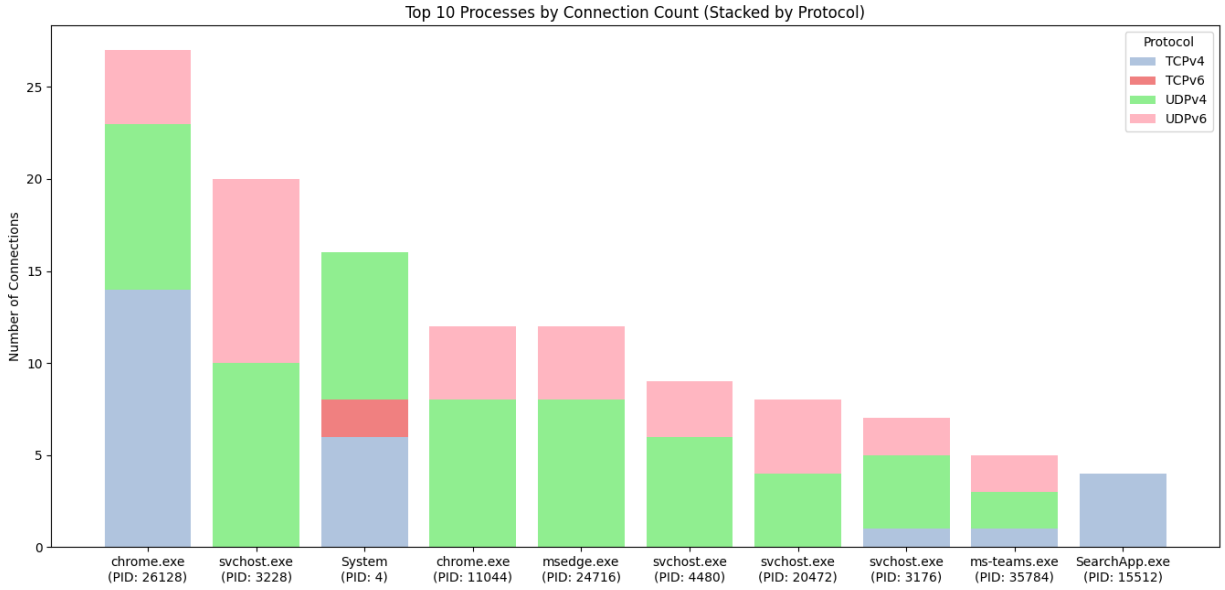
Example Run

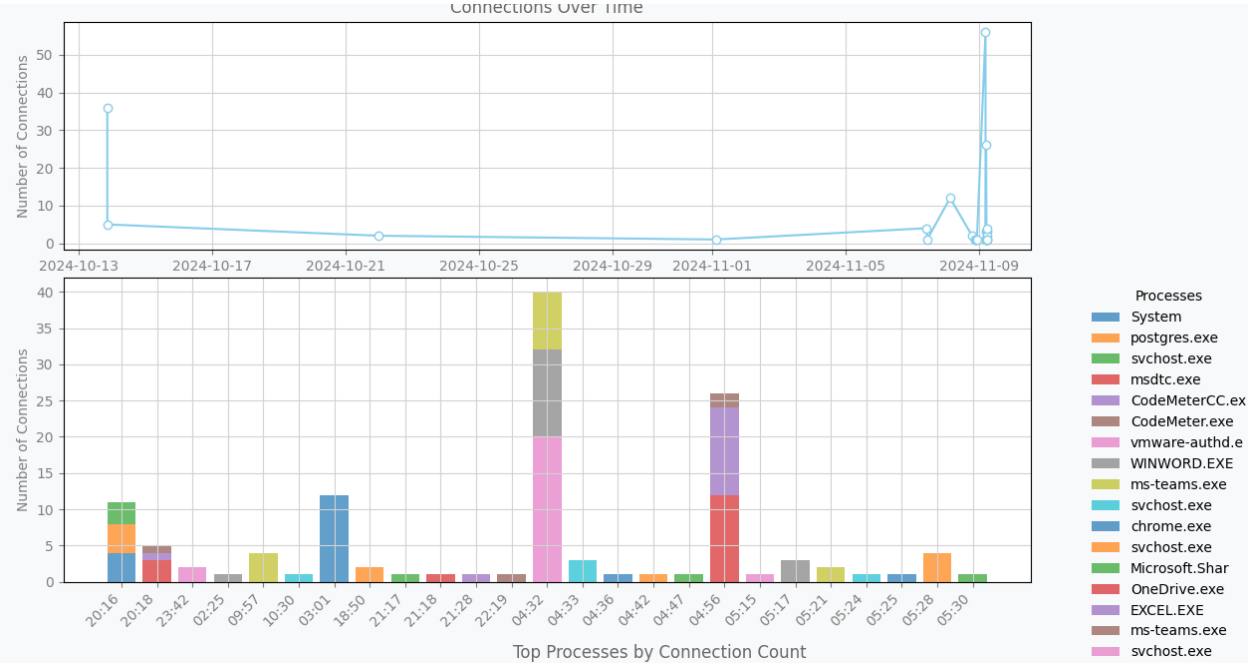
```
$ python -u program.py -f "D:\Westminster\Project\WinPC\memdump2.mem" -o ./output_win11_interface/ -V "D:\Westminster\Proj
ect\volatility\volatility3\vol.py"
Executing Volatility commands...

user@DESKTOP-OSJU4T7 MINGW64 /d/Westminster/Project/spectre
$ ls ./output_win11_interface/
hashdump.json  ldrmodules.json  netscan.json  netstat.json  netstat_original.json  printkey.json  pstree.json
```


Memory Dump Detailed Statistics
Processes: Parent vs Child
Parents
11
19
Children
Processes: Running vs Closed
Running
13
17
Closed
Connections: Internal vs External
External
167
1
Internal
Connections: State Counts
State
CLOSED
CLOSE_WAIT
ESTABLISHED
LISTENING
1
1
26
43
97
0
20
40
60
80
100
Connections: Protocol Counts
TCPv6
TCPv4
13
58
63
34
UDPv4
UDPv6
Protocols for PID 26128 (Most Connections)
Protocol Type
UDPv6
UDPv4
TCPv6
TCPv4
0.0
2.5
5.0
7.5
10.0
12.5
PID 26128
Top 3 Processes with Most Threads
PID 4
178
51
52
PID 34232
PID 24716
Top 3 Processes with Most DLLs
PID 3420
400
PID 19068
358
282
PID 34232
Top 3 Processes with Most Connections
27
20
16
PID 3228
PID 4


Process and Connection Timeline
Process Creation and Network Connection Timeline
Process Creations
Network Connections
Number of Events
0
20
40
60
80
100
2024-10-13
2024-10-17
2024-10-21
2024-10-25
2024-10-29
2024-11-01
2024-11-05
2024-11-09





Use Case

Ideal for forensic analysts and incident responders looking to automate memory dump analysis and generate visual insights quickly.

Anomaly Detection Module

This guide will help you understand how to use of Anomaly Detection Module, which contains classes which helps detect anomalies related to processes, connections and IP addresses.

IPDetectionModule and MaliciousIPDetector

the IPDetectionModule and MaliciousIPDetector classes in the Anomaly Detection Module. These classes are designed to detect malicious IP addresses by leveraging VirusTotal and Spamhaus services, as well as performing WHOIS lookups for detailed information.

1. IPDetectionModule

- **Purpose**: Acts as the main coordinator for malicious IP detection.
- **Features**:
    - Detects malicious IPs using VirusTotal and Spamhaus.
    - Provides an option for WHOIS lookups.
    - Outputs results in a formatted table.
    - Optionally plots a comparison of benign vs malicious IP counts.

2. MaliciousIPDetector

- **Purpose**: Provides utility functions for IP analysis.
- **Key Methods**:
    - check_virustotal: Queries VirusTotal for malicious activity.
    - check_spamhaus: Checks if an IP is blacklisted in Spamhaus.
    - ip_to_domain: Resolves an IP address to a domain name.
    - perform_whois_lookup: Performs WHOIS lookup for a domain.
    - display_results: Displays analysis results in a formatted table.
    - plot_results: Visualizes benign vs malicious IP counts.
    - get_ip_info: Captures the geo location of the ip address.

Prerequisites

1. **API Keys**:
    - Obtain an API key from VirusTotal.
    - (Optional) Obtain an API key for WHOIS lookups (e.g., from ip2whois).
2. **Prepare Input Files**:
    - Create files for compromised and safe IP addresses.

Step-by-Step Guide

Step 1: Prepare Input Files

Safe and unsafe IP addresses are provided by SPECTRE in emulation/ip_lists as file_goole.txt and compromised_ip_full.txt respectively. Custom IP files can be created as per requirements.

Step 2: Use the Example Program

Run the provided example program emulation/ip_detections.py, which utilizes the IPDetectionModule and MaliciousIPDetector.

Command-Line Arguments:

- -i, --ip-addresses: Comma-separated list of IP addresses to check.
- -k, --api-key: API key for VirusTotal.
- --whois-api-key: (Optional) API key for WHOIS lookups.
- --unsafe-ips: File containing compromised IPs.
- --safe-ips: File containing safe IPs.
- -v, --verbose: Enables detailed WHOIS output.

Sample Workflow

1. Detection Process:

- IPDetectionModule.detect_malicious_ips:
    - Checks each IP against VirusTotal and Spamhaus.
    - Classifies IPs as malicious or benign.

- Performs WHOIS lookups for flagged IPs if a WHOIS API key is provided.
- Displays results in a table.

2. Result Visualization:

- Plots a bar chart comparing the counts of benign and malicious IPs.
- Monitor system behaviour for anomalies in process and network activities.

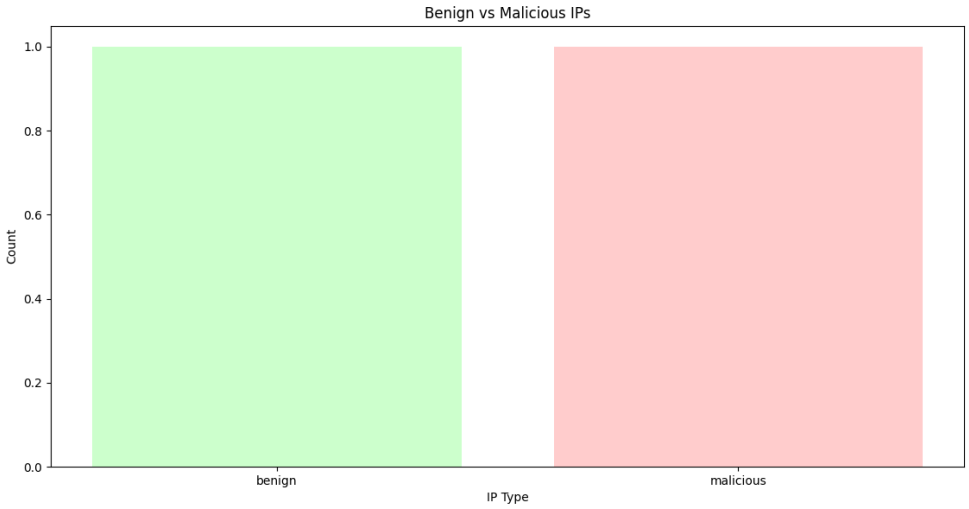
3. Example Program Usage:

```
$ python ./ip_detections.py -i "158.65.89.41,188.241.140.212"  --safe-ips ../emulation/ip_lists/file_google.txt --unsafe-ips ../emulation/ip_lists/compromised_ip_full.txt -k [Virus Total Key goes here] --whois-api-key [WHOIS API Key goes here] -v
+-------------------------------------------------------------------------------------------+
|                              Report for IP: 158.65.89.41                                  |
+------------------------------+------------------------------------------------------------+
| Check                        | Result                                                     |
+------------------------------+------------------------------------------------------------+
| VirusTotal Malicious Activity | 0 out of 94                                               |
| VirusTotal Report Link       | https://www.virustotal.com/api/v3/ip_addresses/158.65.89.41 |
| Spamhaus Blacklisted         | No                                                         |
+------------------------------+------------------------------------------------------------+
VT Result : {'malicious_indicators': 0, 'total_indicators': 94}
+-------------------------------------------------------------------------------------------+
|                              Report for IP: 188.241.140.212                               |
+------------------------------+------------------------------------------------------------+
| Check                        | Result                                                     |
+------------------------------+------------------------------------------------------------+
| VirusTotal Malicious Activity | 1 out of 94                                               |
| VirusTotal Report Link       | https://www.virustotal.com/api/v3/ip_addresses/188.241.140.212 |
| Spamhaus Blacklisted         | No                                                         |
+------------------------------+------------------------------------------------------------+
VT Result : {'malicious_indicators': 1, 'total_indicators': 94}
host-188-241-140-212.dynamic.wirteksrl.it. is not avaiable in ip2whois.com

IP GetLocation Information:
IP Address: 188.241.140.212
Hostname: host-188-241-140-212.dynamic.wirteksrl.it
City: Matelica
Region: The Marches
Country: IT
Location: 43.2579,13.0075
Organization: AS201602 NEWTEC S.R.L.
Postal: 62024
Timezone: Europe/Rome
Malicous IPs as per VirusTotal : {'188.241.140.212'}
Non-malicous IPs as per VirusTotal : {'158.65.89.41'}
{'158.65.89.41': '0:94', '188.241.140.212': '1:94'}
```

GEOLocation Information

Figure 1. Scenario Where WHOIS lookup fails.



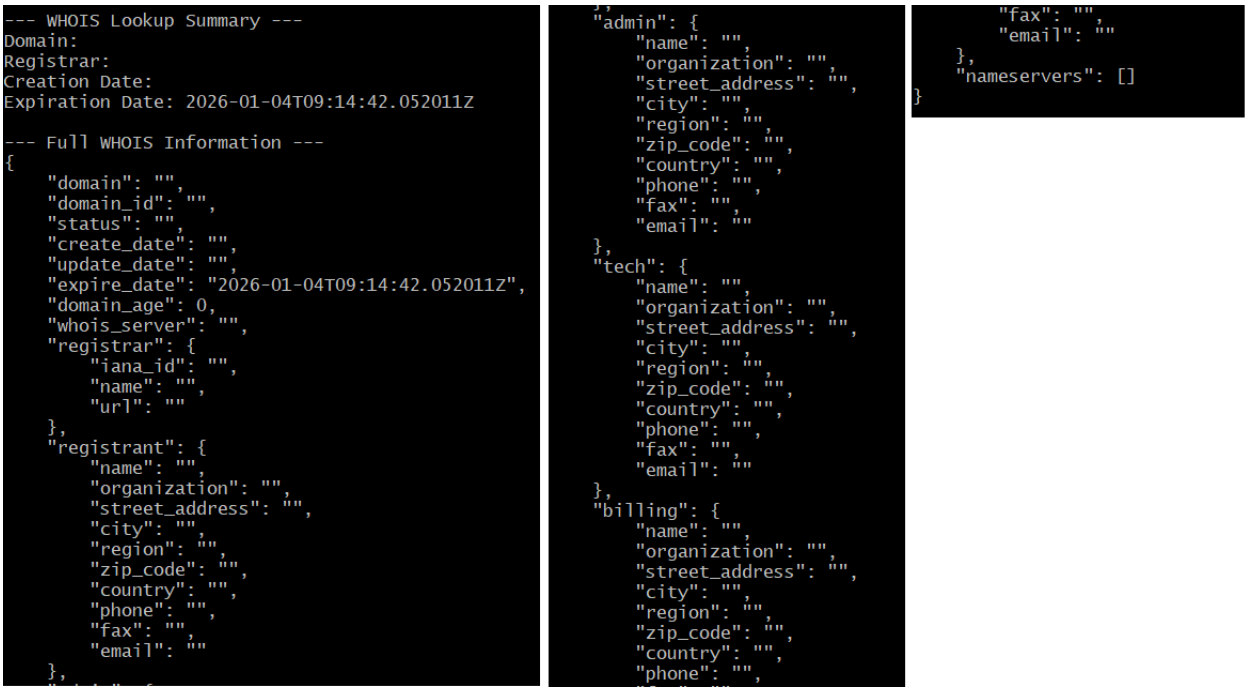


Figure 2. Scenario Where WHOIS lookup is successful.

Code Highlights

- **Initialization**:
  - IPDetectionModule orchestrates the detection process.
  - Relies on utility methods in MaliciousIPDetector.
- **Key Functionalities**:
  - VirusTotal checks identify the number of malicious indicators for each IP.
  - Spamhaus checks determine if an IP is blacklisted.
  - WHOIS lookups provide detailed ownership and registration information. If information isn't available, a message is generated for that as well.
  - GeoLocation is generated using ipinfo.io
- **Visualization**:
  - plot_results generates an intuitive bar chart summarizing the findings.

Use Case Scenarios

1. **Security Analysts**:
   - Quickly identify malicious IPs in a network traffic log.

    - Investigate malicious indicators with WHOIS lookups.
2. **Incident Responders**:
    - Flag compromised IPs during post-breach investigations.
3. **System Administrators**:
    - Monitor safe and unsafe IP lists to maintain secure environments.

IPCategorytDetector

The IPCategorytDetector class analyzes IP connections and process data to categorize IPs into blacklisted, whitelisted, or other, and visualize the results.

Key Features

1. **IP Categorization**: Classifies IPs as blacklisted, whitelisted, or other based on predefined lists.
2. **Command Analysis**: Identifies IPs mentioned in process commands and categorizes them.
3. **Visualization**: Provides graphical insights into the data, such as:
    - IP category distribution.
    - Blacklisted connections by process.

Initialization

```
detector = Detections.IPCategorytDetector(
    blacklist_file="path_to_blacklist.txt",
    whitelist_file="path_to_whitelist.txt",
    pstree_file="path_to_process_tree.json",
    connections_file="path_to_connection_data.json"
)
```

Methods Overview

1. **load_ip_list(filename)**
    - **Purpose**: Loads a list of IPs from a file.

- **Returns**: A set of IP addresses.

2. **load_json_file(filename)**
    - **Purpose**: Loads JSON data from a file.
    - **Returns**: A Python dictionary representing the JSON content.
3. **categorize_ips(axis=None)**
    - **Purpose**: Categorizes foreign IPs in the connections data.
    - **Visual Output**: Bar chart showing counts of blacklisted, whitelisted, and other IPs.
4. **search_cmd_ips(process, blacklist_counts)**
    - **Purpose**: Recursively searches process commands (Cmd field) for IP addresses, categorizing them.
5. **process_cmd_ip_analysis()**
    - **Purpose**: Analyzes IPs in process commands and visualizes the results.
    - **Visual Output**: Bar chart showing CMD IP categorization (blacklist, whitelist, other).
6. **plot_blacklist_connections_by_process(axis=None)**
    - **Purpose**: Visualizes the number of blacklisted connections by process.
    - **Visual Output**: Bar chart with processes on the x-axis and their blacklisted connection counts.

Demonstration Program

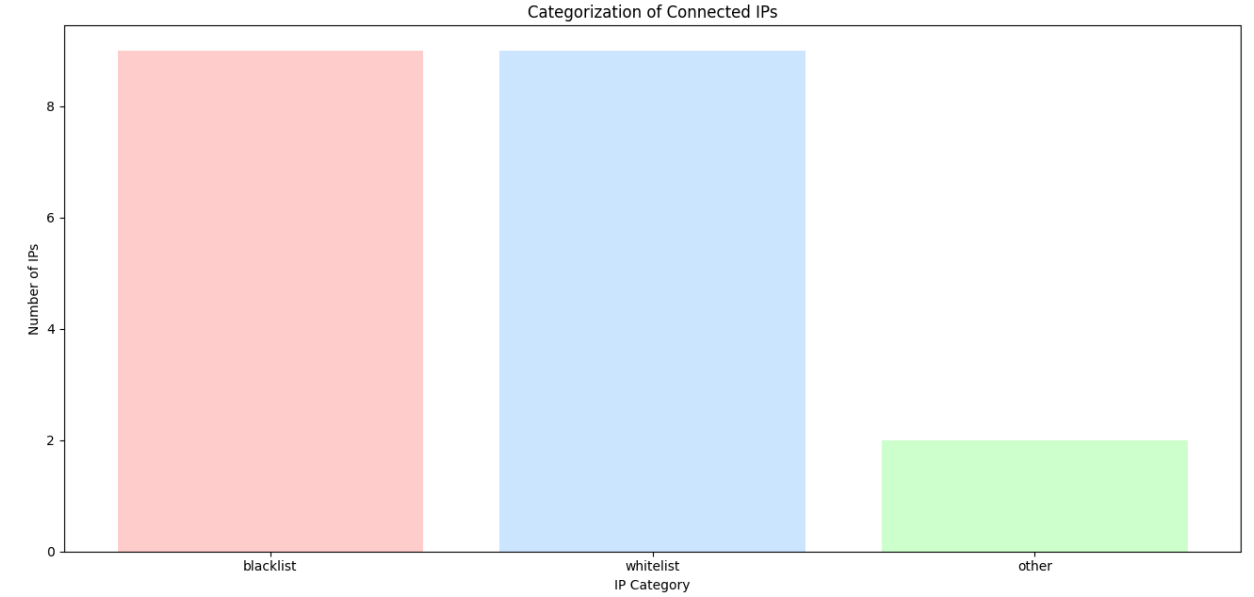
The provided program, ip_listing.py, demonstrates how to use the class:

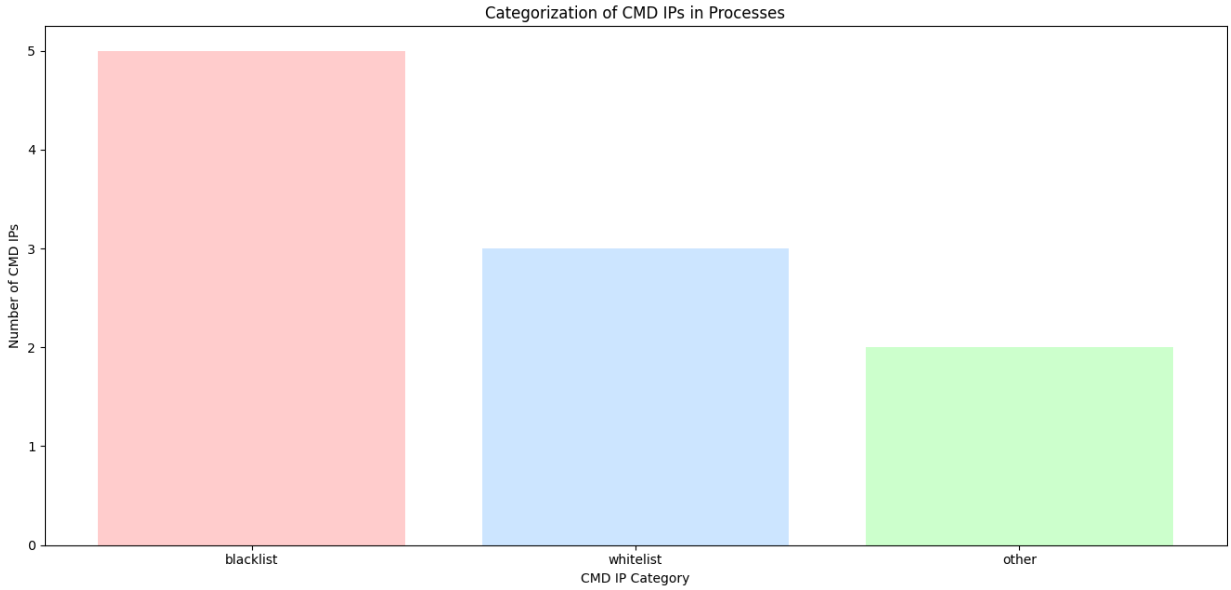
1. Accepts file paths for blacklisted/whitelisted IPs, process tree data, and connection data.
2. Creates an IPCategorytDetector instance.
3. Calls the following methods:
    - categorize_ips(): Categorizes and visualizes IP connections.
    - process_cmd_ip_analysis(): Analyzes and visualizes CMD IP categorization.

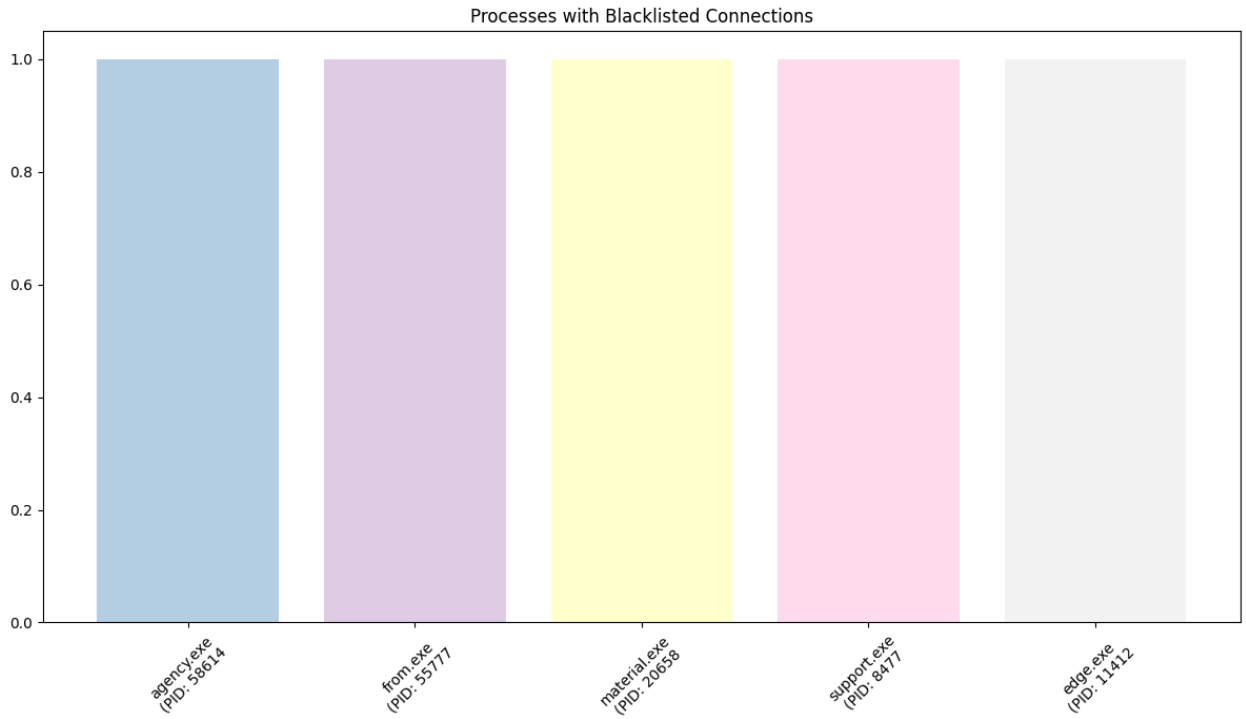
- plot_blacklist_connections_by_process(): Visualizes processes with blacklisted connections.

Execution

```
$ python ./ip_listing.py \
    --pstree-file ../emulation/ip_commands_pstree.json \
    --blacklist-ips ../emulation/ip_lists/file_google.txt \
    --whitelist-ips ../emulation/ip_lists/compromised_ip_full.txt \
    --connections-file ../emulation/10/netstat.json
```







Outputs

1. **Categorization of IPs**:
    - A bar chart showing counts of blacklisted, whitelisted, and other IPs in the connection data.
2. **CMD IP Analysis**:
    - A bar chart categorizing IPs found in process commands.
3. **Blacklisted Connections by Process**:
    - A sorted bar chart showing processes with their counts of blacklisted connections.

MaliciousRunDLLProcess

The MaliciousRunDLLProcess class is a utility designed to analyze and identify potentially malicious rundll32.exe processes based on their behavior and network activity.

Features Overview

1. **Detect Malicious rundll32.exe Processes**

- Scans a list of processes and network connections to identify suspicious rundll32.exe instances.
- Flags processes with:
  - No command-line arguments.
  - Associated network connections.
  - Child processes (potential misuse of rundll32.exe).

2. **Classification of Processes**
   - **Malicious:** Suspicious rundll32.exe processes (e.g., without arguments but with network connections and child processes).
   - **Low Risk:** rundll32.exe processes without arguments, child processes and network connections.
   - **Non-Malicious:** Legitimate rundll32.exe processes.
3. **Visualization**
   - Plots a bar chart summarizing the number of malicious, low-risk, and non-malicious processes.

How to Use the Class

1. **Initialization**

   Create an instance of the MaliciousRunDLLProcess class by providing:
   - process_list: A list of processes, each represented by ProcessTree objects.
   - network_connections: A list of NetworkConnection objects.
   - verbose: (Optional) Enable verbose logging for detailed output.

```
detector = MaliciousRunDLLProcess(process_list, network_connections,
verbose=True)
```

2. **Detect Malicious Processes**: Use the detect_malicious_rundll32 method to analyze the processes:

```
malicious, non_malicious, low_risk = detector.detect_malicious_rundll32()
```

- Returns three lists of categorized processes.

3. **Visualize Results**: Plot a summary of the detection results:

```
detector.plot_results()
```

Running Detection

Use the provided **malicious_rundll32.py** script to test the functionality.

1. **Prepare Input Data**
   - JSON file with process trees (--pstree).
   - JSON file with network connections (--netstat).
2. **Run the Script**: Execute with appropriate arguments:

   python malicious_rundll32.py --pstree processes.json --netstat connections.json -v

   - Add --skip-plot to disable visual output.

Output Details

1. **Detection Results**
   - Prints alerts and information messages for detected processes.
   - Generates a tabular report (if verbose) summarizing suspicious network activity.
2. **Plot Summary**
   - Bar chart showing counts of malicious, low-risk, and non-malicious processes.
3. **Use Cases**
   - Detect malware exploiting rundll32.exe for unauthorized activities.
   - Monitor system behavior for anomalies in process and network activities.

Key Methods

1. detect_malicious_rundll32()

- Core detection logic, recursively analyses processes for suspicious patterns.
- Highlights suspicious rundll32.exe processes based on arguments and network activity.

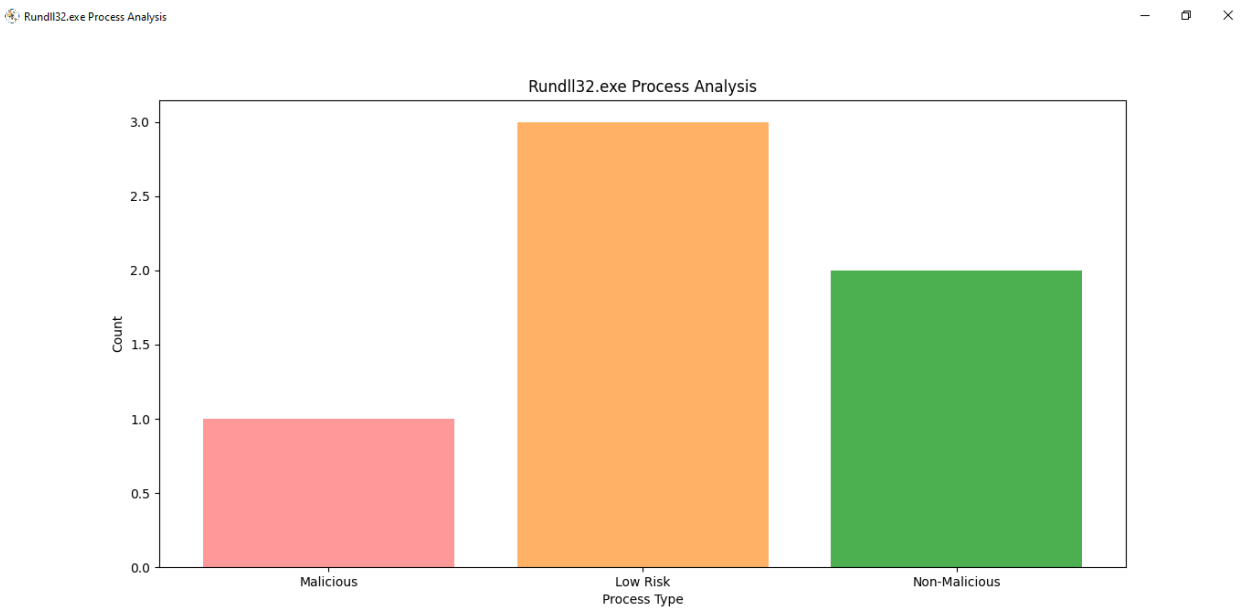
2. plot_results(axis=None)
   - Visualizes results with an optional axis for custom plotting.

Example Execution

```
$ python ./malicious_rundll32.py --pstree ../emulation/pstree.json --netstat ../emulation/netstat.json -v
[INFO] Likely legitimate rundll32.exe process with arguments: PID 52526
[INFO] Likely legitimate rundll32.exe process with arguments: PID 36307
[INFO] Likely legitimate rundll32.exe process with arguments: PID 19545
[Alert] Suspicious rundll32.exe process without arguments and connections: PID 54732
[Verbose Info] Legitimate rundll32.exe process with arguments: PID 19665, Command: rundll32.exe above.dll,keep --log=/thin
k/performance/affect.log
[Verbose Info] Legitimate rundll32.exe process with arguments: PID 44155, Command: rundll32.exe energy.dll,wish --log=/cou
rt/time/true.log

Malicious Process Connections:
```

PID	ImageFileName	Local Address	Local Port	Foreign Address	Foreign Port
54732	rundll32.exe	10.0.2.15	60903	34.1.217.144	27405
54732	rundll32.exe	10.0.2.15	4111	46.101.206.191	26156



CredentialDumpDetector

The CredentialDumpDetector class is designed to identify potential credential dumping activities by analyzing process trees for known malicious patterns.

Features Overview

1. **Detect Credential Dumping Activities**
    - Scans process command-line arguments for patterns associated with:
        - **ProcDump**: Credential dumping with procdump.exe.
        - **rundll32.exe**: Using comsvcs.dll for credential dumping.
2. **Recursive Detection**
    - Traverses processes and their child processes for comprehensive analysis.
3. **Results Visualization**
    - Bar chart summarizing detected credential dumping activities.
    - Tabular display of detected processes with details.

How to Use the Class

1. **Initialization**

   Create an instance of the class with:
    - processes_list: A list of process objects (ProcessTree instances).
    - verbose: (Optional) Enable detailed logs.

```
detector = CredentialDumpDetector(processes_list, verbose=True)
```

2. **Run Detection**: Use the detect_credential_dumping() method to analyze processes:

```
detections = detector.detect_credential_dumping()
```

    - Returns a dictionary where each entry contains the PID, detection method, and command details.
3. **Visualize Results**: Plot the summary of detected activities:

```
CredentialDumpDetector.plot_detection_summary(detections.values())
```

4. **Display Detailed Results**: Use the display_detections() method to print a tabular summary:

```
CredentialDumpDetector.display_detections(detections.values())
```

Example Program

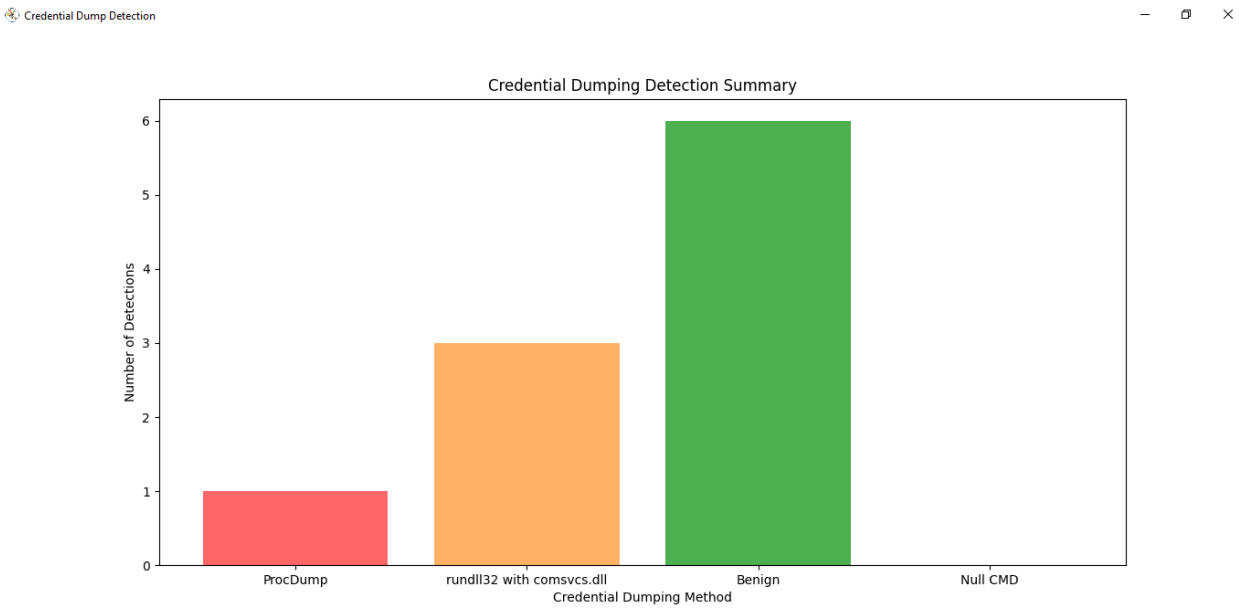
Use the demonstration program **credentials_dumping_detections.py** to test the functionality.

1. **Prepare Input Data**
    - JSON file containing process trees (--pstree). Can be created using emulation module or captured using actual memory dump image.
2. **Run the Script**: Execute with necessary arguments:

```
$ python credentials_dumping_detections.py --pstree pstree.json -v
```

    - Add --skip-plot to disable visualization.

```
$ python ./credentials_dumping_detections.py --pstree ../emulation/credentials_pstree.json -v
[Info] No credential dumping detected for: PID 31316, Command: skin.exe -param1 line out -config 9522
[Info] No credential dumping detected for: PID 42175, Command: number.exe -param1 we Democrat expect -config 466
9
[Info] No credential dumping detected for: PID 40221, Command: fear.exe -param1 art instead agree -config 2987
[Info] No credential dumping detected for: PID 49308, Command: always.exe -param1 audience -config 5383
[Info] No credential dumping detected for: PID 39538, Command: believe.exe -param1 great of -config 9422
[Info] No credential dumping detected for: PID 11942, Command: protect.exe -param1 including -config 6274
[Alert] Detected ProcDump credential dumping: PID 43371, Command: procdump.exe -ma lsass.exe -o \partner\structu
re\last.odp\lsass_dump.dmp
[Alert] Detected rundll32 credential dumping: PID 43740, Command: rundll32.exe \medical\performance\lawyer.ods\c
omsvcs.dll MiniDump 3584 lsass.dmp full
[Alert] Detected rundll32 credential dumping: PID 62876, Command: rundll32.exe \yes\focus\member.mp3\comsvcs.dll
 MiniDump 4353 lsass.dmp full
[Alert] Detected rundll32 credential dumping: PID 8197, Command: rundll32.exe \will\blue\space.mp4\comsvcs.dll M
iniDump 7345 lsass.dmp full
```



Output Details

1. **Detection Results**
    - Alerts for detected credential dumping activities.
    - Verbose logs for benign and other processes.
2. **Visualization**
    - Bar chart summarizing credential dumping methods:
        - **ProcDump**
        - **rundll32 with comsvcs.dll**
        - **Benign**
        - **Null CMD** (processes without command details).

ProcessExtensionAnalyzer

The ProcessExtensionAnalyzer class identifies unsafe file extensions in processes to detect potential security risks.

Features Overview

1. **Safe Extension Verification**

- Maintains a list of known **safe executable extensions** (e.g., .exe, .bat, .ps1).
- Validates file paths against this list.

2. **Recursive Detection**
   - Analyses a process and its children recursively for unsafe extensions.
3. **Result Summarization**
   - Provides detailed findings for processes with unsafe extensions.
   - Counts occurrences of unsafe extensions for visualization.

How to Use the Class

1. **Validate File Extensions**: Use is_safe_extension() to check if a file path has a safe extension:

```
is_safe = ProcessExtensionAnalyzer.is_safe_extension("example.exe")
print(is_safe)  # True if the extension is safe
```

2. **Detect Unsafe Extensions in Processes**: Use detect_unsafe_extensions() to analyze a list of processes:

```
unsafe_entries, extensions_count =
ProcessExtensionAnalyzer.detect_unsafe_extensions(processes_list)
```

   - **details**: Dictionary of processes with unsafe extensions and associated metadata.
   - **extensions_count**: Count of each unsafe extension encountered.
3. **Recursive Analysis of a Single Process**: Use detect_unsafe_extensions_recursive() to analyze a process and its children:

```
details = {}
extensions_count = defaultdict(int)
```

```
ProcessExtensionAnalyzer.detect_unsafe_extensions_recursive(process,
details, extensions_count)
```

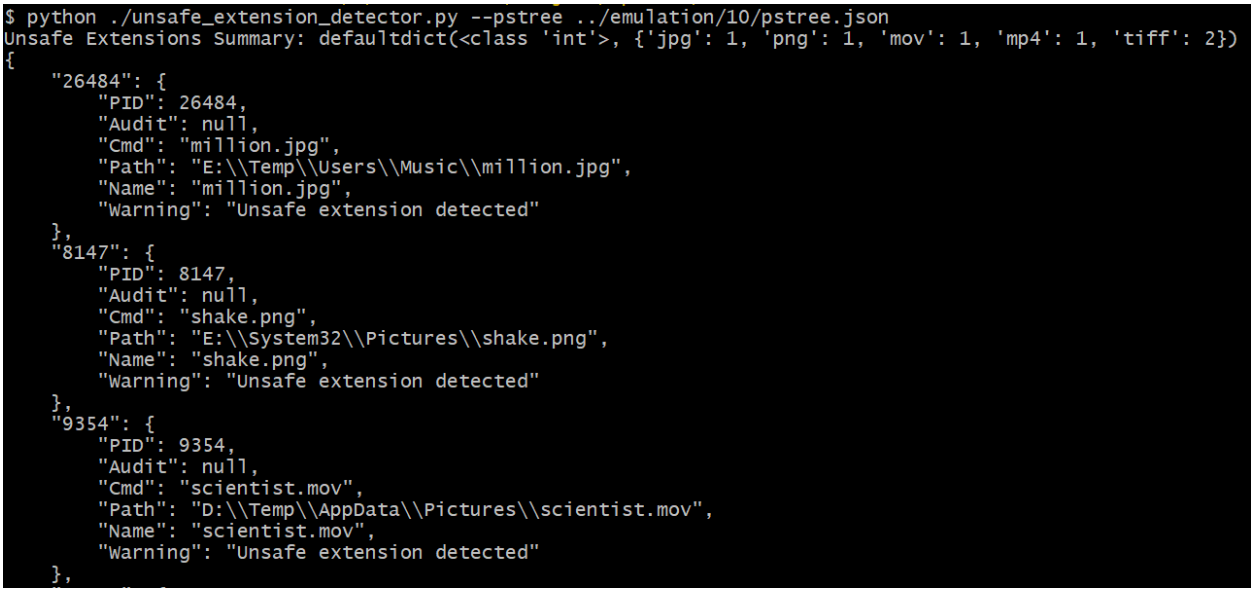
Example: Running Detection

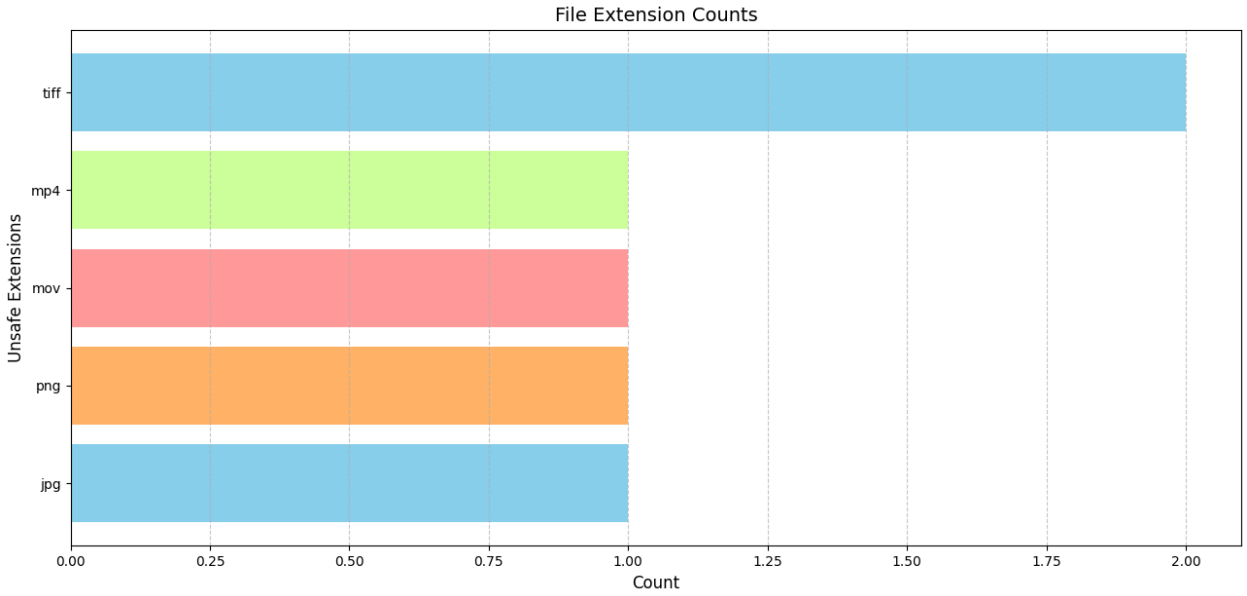
Use the demonstration program **unsafe_extension_detector.py** for testing.

1. **Prepare Input Data**
    - JSON file containing the process tree (--pstree).
2. **Run the Script**: Execute with the JSON file as input:

    python unsafe_extension_detector.py --pstree processes.json

    - Outputs unsafe extensions and triggers visualization.





Output Details

1. **Detection Results**
    - Detailed report of processes with unsafe extensions, including:
        - **PID**, **Command**, **Path**, and **Audit Data**.
    - Warnings for unsafe extensions.
2. **Extension Counts**
    - Summarized counts of unsafe extensions for quick insights.
3. **Visualization**

ConnectionDetector

The ConnectionDetector class provides tools for analyzing, visualizing, and detecting patterns in network connections from a netstat JSON file.

Features Overview

1. **Load and Analyze Network Connections**
    - Parses a JSON file containing connections data.
    - Retrieves geographical data for foreign IPs.
2. **Visualizations**

- **Connection Types**: Bar chart of protocols (e.g., TCPv4, TCPv6).
- **Country Analysis**:
  - Histogram of connection counts by country.

3. **Utility Methods**
   - Retrieves country names for IPs.
   - Handles time parsing and rounding.

How to Use the Class

1. Initialization

Load and parse a connections JSON file:

```
detector = ConnectionDetector("connections.json")
```

2. Visualize Connection Types

Plot a bar chart of protocols (TCP/UDP, v4/v6):

```
detector.plot_connection_types()
```

3. Country Analysis

Display Histogram of Connections by Country:

```
detector.display_countries()
```

Example: Running Detection

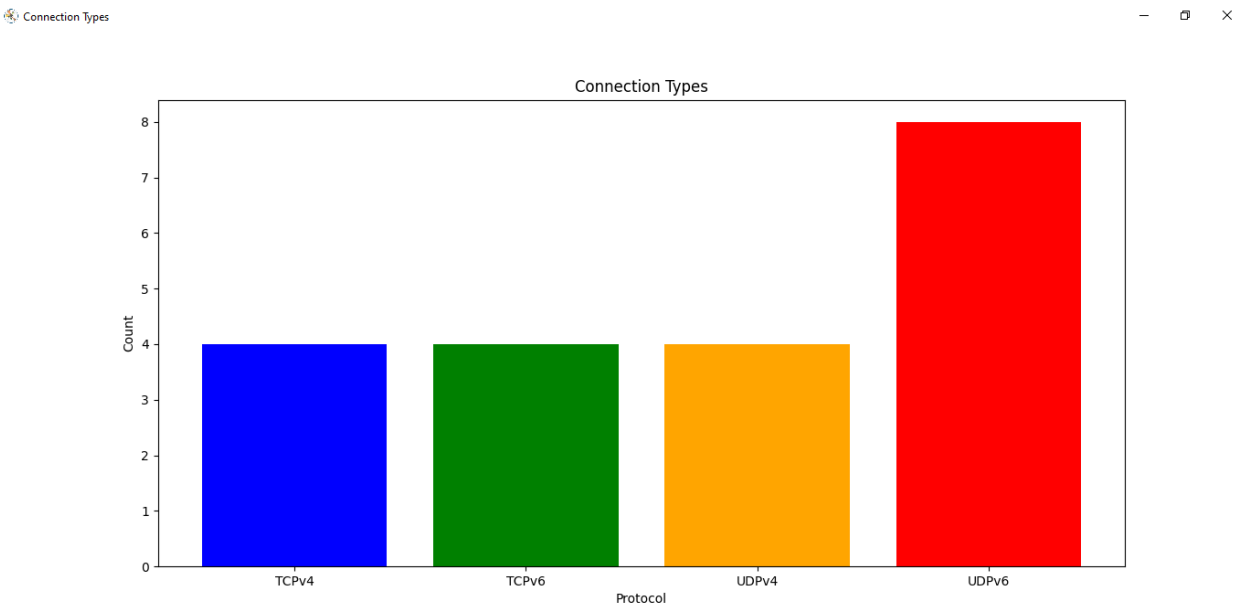
Use the demonstration program **connections_detection.py** to analyze and visualize data.

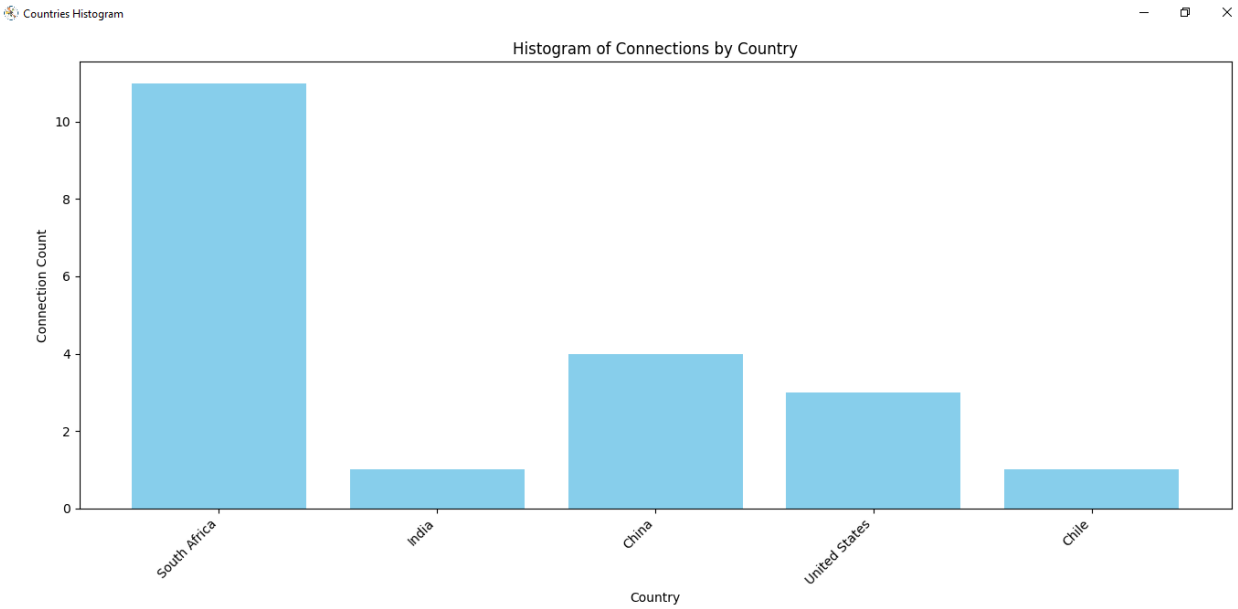
1. Prepare Input Data

- Provide the path to a connections JSON file (--connections-file).

2. Run the Script

Execute the script to generate visualizations:





Output Details

Connection Types

- Bar chart of protocol counts: TCPv4, TCPv6, UDPv4, UDPv6.

Country Histogram

- Histogram of connections by country.

MaliciousRundll32Child

Detect and visualize suspicious process lineages where rundll32.exe is spawned by suspicious parent processes.

Key Features

1. **Detection of Suspicious Lineages:**
    - Monitors process trees for instances of rundll32.exe spawned by known suspicious parent processes.
    - Flags potential threats and maintains categorized lists (malicious and non_malicious).
2. **Visualization:**
    - Directed graph representing parent-child process relationships.
    - Hover-enabled graph visualization for easier exploration.
    - Histogram comparing counts of malicious and non-malicious processes.

Attributes

- SUSPICIOUS_PARENTS:
    - List of process names considered suspicious if they spawn rundll32.exe.
    - Examples: "winword.exe", "excel.exe", "taskeng.exe".
- malicious: List of detected malicious process IDs.
- non_malicious: List of non-malicious process IDs.
- graph: Directed graph (networkx.DiGraph) representing the process lineage.

Methods

1. detect_lineage(process_list):

- **Input:** List of process objects with attributes like image_file_name, pid, and children.
- **Behavior:** Recursively detects parent-child relationships, classifying and categorizing processes as malicious or non-malicious.
- **Flags Suspicious Lineages:** Adds flagged relationships to the graph with distinct edge colors.

2. _add_edge_to_graph(parent, child, color):
    - Adds parent-child relationships as nodes and edges in the graph.
    - Uses different colors (red for malicious, green for benign) for nodes.
3. plot_process_lineage(axis=None):
    - **Output:** A graph with minimal node/edge overlap. Displays suspicious and benign relationships with distinct colors.
    - **Optional Input:** Matplotlib axis for embedding within larger visualizations.
4. plot_process_lineage_with_hover(fig=None, axis=None):
    - **Output:** Interactive graph where node labels appear on hover.
    - **Behavior:** Highlights node labels dynamically based on cursor position.
    - **Optional Input:** Matplotlib figure and axis for integration.
5. plot_process_histogram(axis=None):
    - **Output:** Bar chart comparing counts of malicious vs. non-malicious processes.
    - **Optional Input:** Matplotlib axis for embedded visualization.

Usage Example: CLI Integration

Demonstration script malicious_parents.py provides command-line access for:

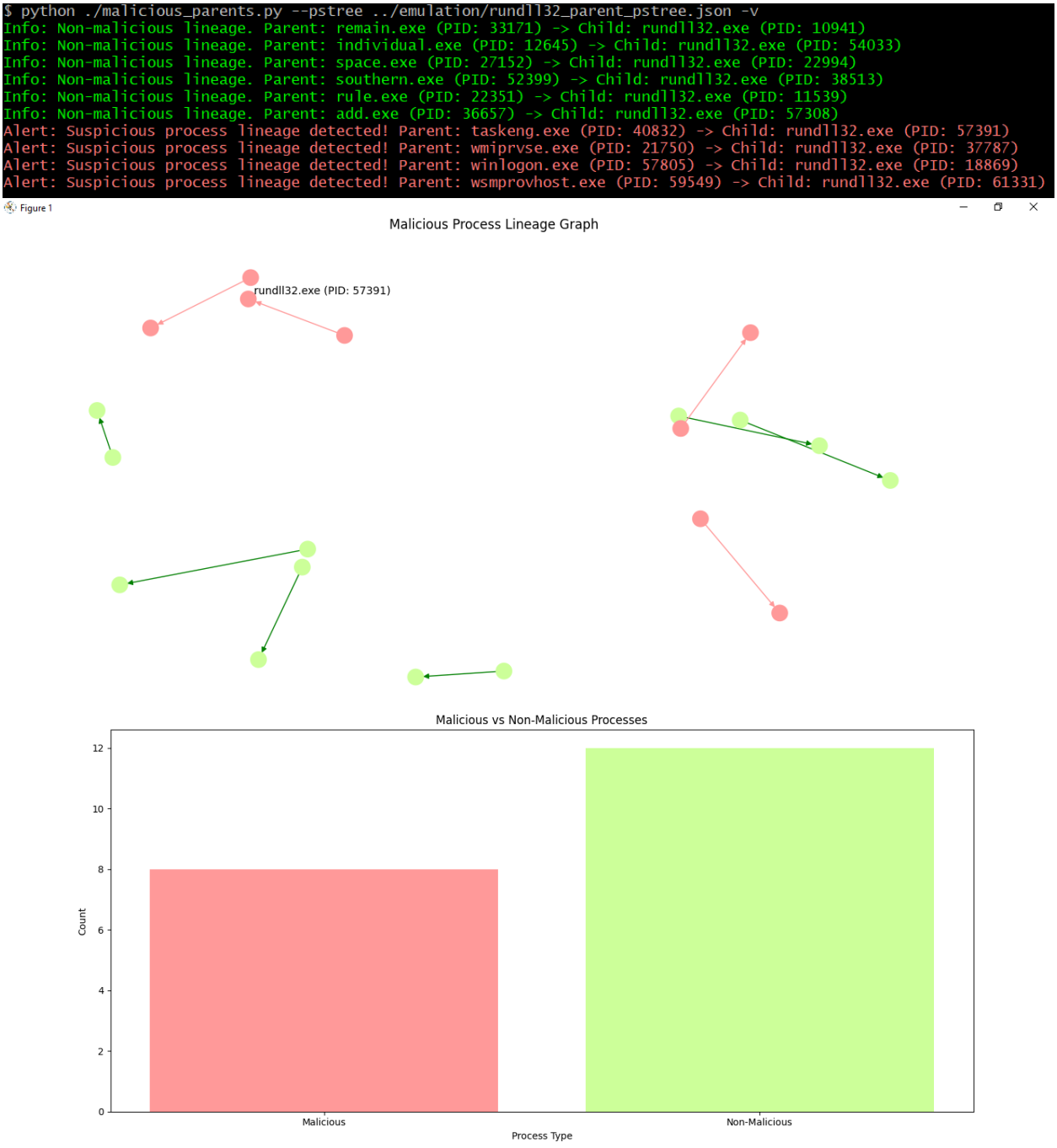
- Parsing a process tree from a JSON file.
- Detecting suspicious process lineages.
- Visualizing process relationships and activity summary.

Command-Line Options

1. --pstree <file>: Specifies the JSON file containing the process tree.
2. --skip-plot: Skips plotting visualizations.
3. --verbose: Enables detailed output during detection.

Sample Execution

```
$ python ./malicious_parents.py --pstree ../emulation/rundll32_parent_pstree.json -v
Info: Non-malicious lineage. Parent: remain.exe (PID: 33171) -> Child: rundll32.exe (PID: 10941)
Info: Non-malicious lineage. Parent: individual.exe (PID: 12645) -> Child: rundll32.exe (PID: 54033)
Info: Non-malicious lineage. Parent: space.exe (PID: 27152) -> Child: rundll32.exe (PID: 22994)
Info: Non-malicious lineage. Parent: southern.exe (PID: 52399) -> Child: rundll32.exe (PID: 38513)
Info: Non-malicious lineage. Parent: rule.exe (PID: 22351) -> Child: rundll32.exe (PID: 11539)
Info: Non-malicious lineage. Parent: add.exe (PID: 36657) -> Child: rundll32.exe (PID: 57308)
Alert: Suspicious process lineage detected! Parent: taskeng.exe (PID: 40832) -> Child: rundll32.exe (PID: 57391)
Alert: Suspicious process lineage detected! Parent: wmiprvse.exe (PID: 21750) -> Child: rundll32.exe (PID: 37787)
Alert: Suspicious process lineage detected! Parent: winlogon.exe (PID: 57805) -> Child: rundll32.exe (PID: 18869)
Alert: Suspicious process lineage detected! Parent: wsmprovhost.exe (PID: 59549) -> Child: rundll32.exe (PID: 61331)
```



Visual Output

1. **Process Lineage Graph:**
    - Nodes represent processes.
    - Edges denote parent-child relationships (red for suspicious, green for benign).
2. **Hover-Enabled Graph:**
    - Displays detailed process information interactively.
3. **Histogram:**
    - Comparison of counts for malicious vs. non-malicious processes.