Report on Python Fundamentals

Log Analyzer

Introduction

The project focuses on developing a Python script to parse and monitor activities recorded in the `auth.log` file of an Ubuntu system. The `auth.log` file is a crucial component in system administration and security, as it records all authentication-related events, including logins, logouts, and authentication failures. Monitoring this log file is essential for detecting unauthorized access attempts, ensuring compliance with security policies, and auditing user activities.

The primary objective of this project is to create a script that can accurately extract and log specific events from the `auth.log` file, such as command executions, user additions and deletions, password changes, `sudo` command usage, failed `sudo` attempts, and any attacks involved.

By doing so, the script aims to enhance the security and accountability of the system by providing detailed insights into user activities and potential security breaches.

This report will cover the following aspects of the project:

- **Methodologies**:
  - A detailed explanation of the steps taken to develop and implement the script, including the use of regular expressions for log parsing and the approach to sorting and writing log entries.
- **Discussion**:
  - An analysis of the findings, highlighting the importance of monitoring specific events in the `auth.log` file and the impact of these events on system security.
- **Conclusion**:
  - A summary of the results achieved through the project, discussing the effectiveness of the script in meeting the project's objectives and identifying any areas for improvement.
- **Recommendations**:
  - Suggestions for enhancing the script and extending its capabilities to provide even more comprehensive security monitoring.

Through this project, the goal is to provide a robust tool for system administrators to monitor and audit system activities effectively, thereby contributing to the overall security and integrity of the system.

Methodologies

The methodology section details the systematic approach taken to develop and implement the Python script for parsing and monitoring activities in the `auth.log` file. This section outlines the tools, techniques, and processes used to achieve the project's objectives.

Tools and Technologies

- **Python**:
  - The primary programming language used for scripting.
- **Regular Expressions (re module)**:
  - Utilized for pattern matching and extracting specific data from log entries.
- **Datetime module**:
  - Used for parsing and manipulating date and time data from log entries.

Script Development

- **Reading the Log File**:
  - The script begins by opening and reading the `auth.log` file line by line to ensure efficient processing and memory management.

```
# Open the auth.log file
with open('/home/kali/PythonProject/simulated_auth.log', 'r') as infile:
    for line in infile:
        entry = parse_line(line)
        if entry:
            log_entries.append(entry)
```

```
with open('/home/kali/PythonProject/simulated_auth.log', 'r') as infile:
```

- This line uses the `with` statement to open the file located at `/home/kali/PythonProject/simulated_auth.log` in read mode (`'r'`).
- The `with` statement ensures that the file is properly closed after its suite finishes, even if an exception is raised.

```
for line in infile:
```

- This loop iterates over each line in the opened file `infile`.

```
entry = parse line(line)
```

- Each line is passed to a function named `parse_line`, which presumably processes the line and returns an entry if the line is relevant or contains useful information.
- The implementation of `parse_line` is not shown in this snippet, but it likely involves extracting meaningful data from each log line.

```
if entry:
    log_entries.append(entry)
```

- If the `parse_line` function returns a non-empty or non-null value (indicating that the line was successfully parsed), the resulting `entry` is appended to the `log_entries` list.
- This list (`log_entries`) is likely defined earlier in the code and serves to collect all the parsed entries from the log file.

Discussion

The discussion section evaluates the findings from parsing the `auth.log` file and highlights the significance of monitoring these logs for system security and administration.

- **Command Usage**: A regular expression was crafted to match entries where users executed commands using `sudo`. This included capturing the timestamp,

user, and the executed command.

```
def parse_line(line):
    # Parse command usage
    match = re.match(command_regex, line)
    if match:
        timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
        try:
            timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
        except ValueError:
            return None
        user = match.group(5)
        command = match.group(6)
        return (timestamp, f"Command Log - Timestamp: {timestamp}, User: {user}, Command: {command}\n")

    # Monitor user authentication changes
    match = re.match(user_regex, line)
    if match:
        timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
        try:
            timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
        except ValueError:
            return None
        event = match.group(4)
        details = match.group(6)
        if event == "useradd":
            return (timestamp, f"User Add - Timestamp: {timestamp}, New user added: {details}\n")
        elif event == "userdel":
            return (timestamp, f"User Delete - Timestamp: {timestamp}, User removed: {details}\n")
        elif event == "passwd":
            return (timestamp, f"Password Change - Timestamp: {timestamp}, Password changed: {details}\n")
        elif event == "sudo":
            return (timestamp, f"ALERT!!!! sudo Command - Timestamp: {timestamp}, User used sudo command: {details}\n")

    # Monitor sudo command usage
    match = re.match(sudo_command_regex, line)
    if match:
        timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
        try:
            timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
        except ValueError:
            return None
        user = match.group(6)
        command = match.group(7)
        return (timestamp, f"sudo Command - Timestamp: {timestamp}, User: {user}, Command: {command}\n")

    # Monitor failed sudo attempts
    match = re.match(sudo_failure_regex, line)
    if match:
        timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
        try:
            timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
        except ValueError:
            return None
        user = match.group(7)
        return (timestamp, f"ALERT! Failed sudo - Timestamp: {timestamp}, User: {user}, Command: sudo command failed\n")

    # Monitor Hydra attack logs
    match = re.match(hydra_attack_regex, line)
    if match:
        timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
        try:
            timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
        except ValueError:
            return None
        status = match.group(5)
        target_user = match.group(6)
        ip_address = match.group(7)
        port = match.group(8)
        if status == "Failed":
            return (timestamp, f"Hydra Attack - Failed login attempt - Timestamp: {timestamp}, User: {target_user}, IP: {ip_address}, Port: {port}\n")
        elif status == "Accepted":
            return (timestamp, f"Hydra Attack - Successful login - Timestamp: {timestamp}, User: {target_user}, IP: {ip_address}, Port: {port}\n")
```

```
def parse_line(line):
```

- This line defines the `parse_line` function which takes a single argument `line`. This argument represents a single line from the log file.

```
match = re.match(command_regex, line)
if match:
    timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
    try:
        timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
    except ValueError:
        return None
    user = match.group(5)
    command = match.group(6)
    return (timestamp, f"Command Log - Timestamp: {timestamp}, User: {user}, Command: {command}\n")
```

- This block of code does a few actions within the if loop:

Regular Expression Matching:

- `re.match(command_regex, line)` attempts to match the `line` against the pattern defined by `command_regex`.
- If the line matches the pattern, `match` will be a match object; otherwise, it will be `None`.

Extracting Timestamp Components:

- `timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"` constructs a timestamp string using the current year and groups extracted by the regex.
- `match.group(1)`, `match.group(2)`, and `match.group(3)` correspond to different parts of the timestamp captured by the regex (e.g., month, day, and time).

Parsing the Timestamp:

- `timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')` attempts to parse the constructed timestamp string into a `datetime` object.
- If the parsing fails, a `ValueError` is raised, and the function returns `None`.

Extracting User and Command:

- `user = match.group(5)` and `command = match.group(6)` extract the user and command details from the matched groups.

Returning the Result:

- `return (timestamp, f"Command Log - Timestamp: {timestamp}, User: {user}, Command: {command}\n")` returns a tuple containing the parsed `timestamp` and a formatted string with the log details.

- **User Authentication Changes**: Multiple regular expressions were defined to match events such as user additions (`useradd`), user deletions (`userdel`), password changes (`passwd`), and usage of the `su` command.

```python
# Monitor user authentication changes
match = re.match(user_regex, line)
if match:
    timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
    try:
        timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
    except ValueError:
        return None
    event = match.group(4)
    details = match.group(6)
    if event == "useradd":
        return (timestamp, f"User Add - Timestamp: {timestamp}, New user added: {details}\n")
    elif event == "userdel":
        return (timestamp, f"User Delete - Timestamp: {timestamp}, User removed: {details}\n")
    elif event == "passwd":
        return (timestamp, f"Password Change - Timestamp: {timestamp}, Password changed: {details}\n")
    elif event == "sudo":
        return (timestamp, f"ALERT!!!! sudo Command - Timestamp: {timestamp}, User used sudo command: {details}\n")
```

- This block of code does more action to filter and find out any details that should be capture

Regular Expression Matching:

- Similar to the previous section, `re.match(user_regex, line)` attempts to match the `line` against the `user_regex` pattern.
- If the line matches, `match` will be a match object.

Extracting Event and Details:

- `event = match.group(4)` extracts the specific event type (e.g., `useradd`, `userdel`, `passwd`, `sudo`).
- `details = match.group(6)` extracts additional details related to the event.

Handling Different Events:

- Depending on the value of `event`, different formatted strings are returned:
    - `useradd`: "New user added"
    - `userdel`: "User removed"
    - `passwd`: "Password changed"
    - `sudo`: "User used sudo command" (marked with `ALERT!!!!!` for emphasis)

```
# Monitor sudo command usage
match = re.match(sudo_command_regex, line)
if match:
    timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
    try:
        timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
    except ValueError:
        return None
    user = match.group(6)
    command = match.group(7)
    return (timestamp, f"sudo Command - Timestamp: {timestamp}, User: {user}, Command: {command}\n")
```

- This block of code does a few actions within the if loop:

Regular Expression Matching:

- The process is identical to the previous sections but uses `sudo_command_regex`.

Extracting User and Command:

- `user = match.group(6)` and `command = match.group(7)` extract the user and command details from the matched groups.

Returning the Result:

- `return (timestamp, f"Sudo Command - Timestamp: {timestamp}, User: {user}, Command: {command}\n")` returns a tuple containing the parsed `timestamp` and a formatted string with the log details.

- **Failed `sudo` Attempts**: A specific regular expression was designed to capture failed `sudo` attempts, which included the timestamp, user, and an indication of the failure.

```
# Monitor failed sudo attempts
match = re.match(sudo_failure_regex, line)
if match:
    timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
    try:
        timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
    except ValueError:
        return None
    user = match.group(7)
    return (timestamp, f"ALERT! Failed sudo - Timestamp: {timestamp}, User: {user}, Command: sudo command failed\n")
```

Regular Expression Matching:

- `re.match(sudo_failure_regex, line)` attempts to match the `line` against the pattern defined by `sudo_failure_regex`.
- If the line matches the pattern, `match` will be a match object; otherwise, it will be `None`.

Extracting Timestamp Components:

- `timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"` constructs a timestamp string using the current year and groups extracted by the regex.
- `match.group(1)`, `match.group(2)`, and `match.group(3)` correspond to different parts of the timestamp captured by the regex (e.g., month, day, and time).

Parsing the Timestamp:

- `timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')` attempts to parse the constructed timestamp string into a `datetime` object.
- If parsing fails, a `ValueError` is raised, and the function returns `None`.

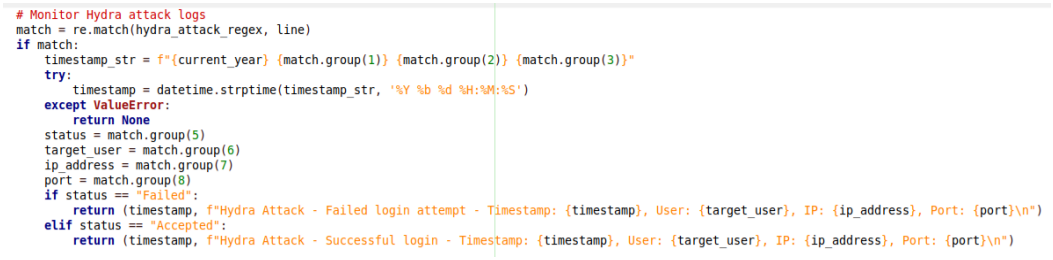
Extracting User:

- `user = match.group(7)` extracts the user details from the matched group.

Returning the Result:

i. `return (timestamp, f"ALERT! Failed sudo - Timestamp: {timestamp}, User: {user}, Command: sudo command failed\n")` returns a tuple containing the parsed `timestamp` and a formatted string with the log details indicating a failed sudo attempt.

- **Hydra Attacks**: A regular expression was also created to detect hydra attack logs, capturing details such as the timestamp, target user, IP address, and the result of the login attempt.



```python
# Monitor Hydra attack logs
match = re.match(hydra_attack_regex, line)
if match:
    timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"
    try:
        timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')
    except ValueError:
        return None
    status = match.group(5)
    target_user = match.group(6)
    ip_address = match.group(7)
    port = match.group(8)
    if status == "Failed":
        return (timestamp, f"Hydra Attack - Failed login attempt - Timestamp: {timestamp}, User: {target_user}, IP: {ip_address}, Port: {port}\n")
    elif status == "Accepted":
        return (timestamp, f"Hydra Attack - Successful login - Timestamp: {timestamp}, User: {target_user}, IP: {ip_address}, Port: {port}\n")
```

Regular Expression Matching:

- `re.match(hydra_attack_regex, line)` attempts to match the `line` against the pattern defined by `hydra_attack_regex`.
- If the line matches the pattern, `match` will be a match object; otherwise, it will be `None`.

Extracting Timestamp Components:

- `timestamp_str = f"{current_year} {match.group(1)} {match.group(2)} {match.group(3)}"` constructs a timestamp string using the current year and groups extracted by the regex.
- `match.group(1)`, `match.group(2)`, and `match.group(3)` correspond to different parts of the timestamp captured by the regex (e.g., month, day, and time).

Parsing the Timestamp:

- `timestamp = datetime.strptime(timestamp_str, '%Y %b %d %H:%M:%S')` attempts to parse the constructed timestamp string into a `datetime` object.
- If parsing fails, a `ValueError` is raised, and the function returns `None`.

Extracting Details:

- `status = match.group(5)` extracts the status of the login attempt (e.g., "Failed" or "Accepted").
- `target_user = match.group(6)` extracts the target user of the attack.
- `ip_address = match.group(7)` extracts the IP address from which the attack originated.
- `port = match.group(9)` extracts the port used in the attack.

Handling Different Statuses:

i. Depending on the value of `status`, different formatted strings are returned:
ii. `if status == "Failed"`: "Failed login attempt" message.
iii. `elif status == "Accepted"`: "Successful login" message.

---

2. **Parsing Log Entries**:
   - Each line of the `auth.log` file is processed using the defined regular expressions. When a match is found, relevant details are extracted and stored.
   - A function `parse_line` was implemented to handle the parsing logic. This function returns a tuple containing the timestamp and a formatted log entry string for each matched event.

```
# Regular expressions for parsing auth.log
command_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sudo\[(\d+)\]:\s+(\S+)\s+:\s+TTY=pts/\d+\s+;\s+PWD=\S+\s+;\s+USER=\S+\s+;\s+COMMAND=(.+)$"
user_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+(useradd|userdel|passwd|su|sudo)\[(\d+)\]:\s+(.*)$"
sudo_command_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sudo\[(\d+)\]:\s+(\S+)\s+:\s+TTY=pts/\d+\s+;\s+PWD=\S+\s+;\s+USER=root\s+;\s+COMMAND=(.+)$"
sudo_failure_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sudo\[(\d+)\]:\s+pam_unix\(sudo:auth\):\s+authentication\s+failure;\s+logname=\S+\s+uid=\d+\s+euid=\d+\s+tty=\S+\s+ruser=\S*\s+rhost=\S+\s+user=(\S+)$"
hydra_attack_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sshd\[(\d+)\]:\s+(Failed|Accepted)\s+password\s+for\s+(\S+)\s+from\s+(\S+)\s+port\s+(\d+)\s+ssh2$"
```

```
command_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sudo\[(\d+)\]:\s+(\S+)\s+:\s+TTY=pts/\d+\s+;\s+PWD=\S+\s+;\s+USER=\S+\s+;\s+COMMAND=(.+)$"
```

`command_regex` Pattern Explanation:

- `^(\S+)`: Matches the month (`\S+` matches a sequence of non-whitespace characters).
- `\s+(\d+)`: Matches the day of the month (one or more digits).
- `\s+(\d{2}:\d{2}:\d{2})`: Matches the time (HH:MM format).
- `\s+server\s+sudo\[(\d+)\]`: Matches the string `server sudo[PID]`, capturing the process ID.
- `:\s+(\S+)`: Matches the user executing the command.
- `\s+:\s+TTY=pts/\d+;\s+PWD=\S+;\s+USER=\S+;\s+COMMAND=(.+)`: Matches the command details, capturing the actual command executed.

```
user_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+(useradd|userdel|passwd|su|sudo)\[(\d+)\]:\s+(.*)$"
```

`user_regex` Pattern Explanation:

- `^(\S+)`: Matches the month.
- `\s+(\d+)`: Matches the day of the month.
- `\s+(\d{2}:\d{2}:\d{2})`: Matches the time.

- `\s+server\s+(useradd|userdel|passwd|su|sudo)\[(\d+)\]`: Matches the string `server useradd[PID]`, `server userdel[PID]`, etc., capturing the event type and process ID.
- `:\s+(.*)`: Captures the rest of the line, which contains additional details about the event.

```
sudo_command_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sudo\[(\d+)\]:\s+(\S+)\s+:\s+TTY=pts/\d+\s+;\s+PWD=\S+\s+;\s+USER=root\s+;\s+COMMAND=(.+)$"
```

`sudo_command_regex` Pattern Explanation:

- Similar to `command_regex`, but specifically matches lines where the command is run as the root user.

```
sudo_failure_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sudo\[(\d+)\]:\s+pam_unix\(sudo:auth\):\s+authentication\s+failure;\s+logname=\S+\s+uid=\d+\s+euid=\d+\s+tty=\S+\s+ruser=\S*\s+rhost=\S+\s+user=(\S+)$"
```

`sudo_failure_regex` Pattern Explanation:

- Matches failed sudo attempts with detailed information about the authentication failure.
- Captures the user attempting the sudo command.

```
hydra_attack_regex = r"^(\S+)\s+(\d+)\s+(\d{2}:\d{2}:\d{2})\s+server\s+sshd\[(\d+)\]:\s+(Failed|Accepted)\s+password\s+for\s+(\S+)\s+from\s+(\S+)\s+port\s+(\d+)\s+ssh2$"
```

`hydra_attack_regex` Pattern Explanation:

- Matches log entries generated by the Hydra tool (or similar brute-force tools) attempting to log in via SSH.
- Captures the login status (failed or accepted), target user, source IP address, and port number.

---

3. **Sorting and Writing Log Entries**:
   - All parsed log entries are stored in a list and sorted by their timestamp to ensure chronological order.
   - The sorted entries are then written to an output file `parsed_logs.txt`. This output file serves as a comprehensive log of all relevant events extracted from the `auth.log` file.

```
# Sort log entries by timestamp (date and time)
log_entries.sort(key=lambda x: x[0])

# Print out the entries before writing to the file
for entry in log_entries:
    print(f"Writing entry: {entry}")

# Write sorted log entries to the output file
with open('/home/kali/PythonProject/parsed_logs.txt', 'w') as outfile:
    for entry in log_entries:
        outfile.write(entry[1])

print("Finished writing to parsed_logs.txt")
```

```
log_entries.sort(key=lambda x: x[0])
```

Explanation:

- Sorts the `log_entries` list based on the timestamp.
- `log_entries` is a list of tuples, where each tuple contains a timestamp and a log entry string.
- `sort(key=lambda x: x[0])` sorts the list using the first element of each tuple (the timestamp) as the key.
- `lambda x: x[0]` is a lambda function that returns the first element of the tuple, which is the timestamp.

```
for entry in log_entries:
    print(f"Writing entry: {entry}")
```

Explanation:

- Prints each log entry to the console before writing it to the file.
- `for entry in log_entries` iterates over each sorted log entry.
- `print(f"Writing entry: {entry}")` prints a message to the console indicating that the entry is being written. This is useful for debugging and monitoring the process.

```
with open('/home/kali/PythonProject/parsed_logs.txt', 'w') as outfile:
    for entry in log_entries:
        outfile.write(entry[1])
```

Explanation:

- Writes the sorted log entries to a file for easier analysis
- `with open('/home/kali/PythonProject/parsed_logs.txt', 'w') as outfile` opens the file `parsed_logs.txt` in write mode (`'w'`). The `with` statement ensures the file is properly closed after writing.
- `for entry in log_entries` iterates over each sorted log entry.
- `outfile.write(entry[1])` writes the second element of the tuple (the log entry string) to the file.

```
print("Finished writing to parsed_logs.txt")
```

Explanation:

- Prints a message to the console indicating that the writing process is complete.
- This message confirms that all log entries have been successfully written to the file.

Significance

- **Enhanced Security Monitoring**:
  - By parsing and logging specific events from the `auth.log` file, the script enhances the system's security monitoring capabilities. Administrators can use the detailed logs to identify and respond to unauthorized access attempts, suspicious user activities, and potential security breaches in a timely manner.
- **Audit and Compliance**:
  - The detailed logs generated by the script support auditing and compliance requirements. Organizations can use these logs to demonstrate adherence to security policies, track user activities, and ensure accountability for administrative actions.
- **Proactive Threat Detection**:
  - The script's ability to detect failed `sudo` attempts and hydra attacks provides a proactive approach to threat detection. By identifying and highlighting these events, administrators can take immediate action to investigate and mitigate potential security threats, thereby reducing the risk of successful attacks.
- **Improved System Integrity**:
  - Monitoring user authentication changes and command usage helps maintain the integrity of the system. Unauthorized changes to user accounts or improper use of administrative commands can be quickly identified and addressed, ensuring the system remains secure and stable.

From CIA Triad

1. **Confidentiality**:
    - Monitoring `sudo` command usage and failed attempts ensures that only authorized users are accessing privileged commands, protecting sensitive information from unauthorized access.
    - Detecting hydra attacks and multiple failed login attempts helps prevent unauthorized access, safeguarding confidential data.
2. **Integrity**:
    - Logging user authentication changes, such as password changes and user additions or deletions, maintains the integrity of the system by ensuring that all changes are authorized and properly documented.
    - By detecting and responding to failed `sudo` attempts and potential brute force attacks, the script helps preserve the integrity of user accounts and system configurations.
3. **Availability**:
    - Continuous monitoring and logging of critical events ensure that system administrators can quickly identify and respond to potential threats, maintaining the availability of the system.
    - By proactively detecting security incidents, such as hydra attacks, the script helps prevent disruptions and ensures that system resources remain available to authorized users.

Conclusion

The project aimed to develop a Python script capable of parsing and monitoring crucial activities recorded in the `auth.log` file on an Ubuntu system. The project successfully achieved its objectives by implementing a script that accurately extracts and logs specific events, such as command executions, user additions and deletions, password changes, `sudo` command usage, failed `sudo` attempts, and hydra attacks.

Summary of Results

1. **Comprehensive Event Logging**:
    - The script effectively captured and logged various events from the `auth.log` file, providing detailed information about each event, including timestamps, user information, and executed commands. This comprehensive logging enables system administrators to audit user activities, ensure compliance with security policies, and maintain system integrity.
2. **Enhanced Security Monitoring**:
    - By highlighting critical events such as failed `sudo` attempts and hydra attacks, the script enhances the system's security monitoring capabilities. Administrators

can quickly identify and respond to unauthorized access attempts and potential security threats, reducing the risk of successful attacks.

3. **Proactive Threat Detection**:
    - The ability to detect and log failed `sudo` attempts and hydra attacks allows for proactive threat detection. Administrators can take immediate action to investigate and mitigate potential security breaches, ensuring the system remains secure and stable.
4. **Support for Auditing and Compliance**:
    - The detailed logs generated by the script support auditing and compliance requirements. Organizations can use these logs to demonstrate adherence to security policies, track user activities, and ensure accountability for administrative actions.

Areas for Improvement

1. **Real-time Monitoring**:
    - Integrating the script with a real-time monitoring system would enhance its capabilities, allowing administrators to receive immediate alerts for critical events and respond more quickly to potential security threats.
2. **Extended Log Parsing**:
    - Extending the script to parse additional log files, such as `syslog` or application-specific logs, would provide a more holistic view of the system's security status. This would enable administrators to monitor a wider range of activities and detect potential threats more effectively.
3. **Comprehensive Error Handling**:
    - Implementing more comprehensive error handling mechanisms would improve the script's robustness. This includes handling unexpected data formats, managing large log files efficiently, and ensuring the script continues to operate smoothly in various scenarios.
4. **User-friendly Interface**:
    - Developing a user-friendly interface for the script would make it more accessible to administrators with varying levels of technical expertise. This could include a graphical user interface (GUI) or a web-based dashboard for viewing and managing log entries.

Recommendations

Based on the findings and conclusions of the project, several recommendations can be proposed to enhance the script's capabilities and provide more comprehensive security monitoring.

1. **Integration with Real-time Monitoring Systems**:
    - **Suggestion**: Integrate the script with real-time monitoring systems such as SIEM (Security Information and Event Management) tools.
    - **Benefit**: This would allow for immediate detection and response to critical events, improving the overall security posture of the system. Real-time alerts can help administrators take swift action against potential security threats.
2. **Extended Log File Parsing**:
    - **Suggestion**: Extend the script to parse additional log files, such as `syslog`, application-specific logs, and other security-related logs.
    - **Benefit**: Parsing a broader range of log files would provide a more holistic view of the system's security status, enabling the detection of a wider array of potential threats and suspicious activities.
3. **Enhanced Error Handling**:
    - **Suggestion**: Implement more comprehensive error handling mechanisms to manage unexpected data formats, large log files, and other potential issues.
    - **Benefit**: Improved error handling would increase the robustness and reliability of the script, ensuring it continues to operate smoothly under various conditions and scenarios.
4. **User-friendly Interface Development**:
    - **Suggestion**: Develop a user-friendly interface for the script, such as a graphical user interface (GUI) or a web-based dashboard.
    - **Benefit**: A more accessible interface would make the script easier to use for administrators with varying levels of technical expertise, enhancing its usability and adoption.
5. **Regular Updates and Maintenance**:
    - **Suggestion**: Establish a routine for regular updates and maintenance of the script to ensure it remains effective against evolving security threats.
    - **Benefit**: Keeping the script up to date with the latest security trends and log file formats will maintain its relevance and effectiveness in providing comprehensive security monitoring.
6. **Documentation and Training**:
    - **Suggestion**: Provide detailed documentation and training materials for the script to help users understand its functionalities and how to utilize it effectively.
    - **Benefit**: Proper documentation and training will empower administrators to fully leverage the script's capabilities, ensuring consistent and accurate monitoring of system activities.

References:

- Comprehensive documentation for Python 3.9, covering all aspects of the language, including file handling, regular expressions, and datetime operations, https://docs.python.org/3.9/
- Detailed guide on using regular expressions in Python, explaining syntax and common patterns, https://docs.python.org/3/howto/regex.html
- A beginner-friendly guide to Python's `re` module, including examples and common functions, https://www.w3schools.com/python/python_regex.asp
- An in-depth series on using regular expressions in Python, including practical examples and advanced usage, https://realpython.com/regex-python/
- Tutorial on Python regular expressions, covering basics to advanced concepts with examples, https://www.programiz.com/python-programming/regex
- Official documentation for Python's `re` module, detailing all available functions and their usage, https://docs.python.org/3/library/re.html
- Python Software Foundation. "Python 3.9.0 Documentation." Python.org, 2020, https://docs.python.org/3/
- Friedl, Jeffrey E.F. "Mastering Regular Expressions." O'Reilly Media, 2006.
- Interactive tutorial with examples on how to use regular expressions in Python., https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial
- A comprehensive series on regular expressions in Python, including practical use cases and examples., https://realpython.com/regex-python/
- A collection of Python examples, including regular expression usage, to help you practice and implement regex in your scripts, https://www.programiz.com/python-programming/examples
- A beginner's guide to Python regular expressions, with detailed explanations and examples, https://www.machinelearningplus.com/python/python-regex-tutorial/
- Basic to advanced concepts of regular expressions in Python., https://www.tutorialsteacher.com/python/python-regex
- A collection of Python examples, including regular expression usage, to help you practice and implement regex in your scripts, https://www.programiz.com/python-programming/examples
- Introduction to regex in Python, including basic concepts and examples, https://realpython.com/regex-python-part-1/
- Detailed tutorial on using regular expressions in Python, including metacharacters and grouping, https://pynative.com/python-regular-expressions/
- Ubuntu Documentation. "Log Files." help.ubuntu.com, https://help.ubuntu.com/community/LinuxLogFiles.
- Scarfone, Karen, and Peter Mell. "Guide to Intrusion Detection and Prevention Systems (IDPS)." NIST Special Publication 800-94, 2007, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-94.pdf
- Kent, Karen, et al. "Guide to Computer Security Log Management." NIST Special Publication 800-92, 2006, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-92.pdf.

- Guide on Ubuntu log files, including the `auth.log` file, its structure, and its significance in system administration, https://help.ubuntu.com/community/LinuxLogFiles
- Guidelines on implementing and managing intrusion detection and prevention systems, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-94.pdf
- Best practices for managing security logs, including collection, storage, and analysis, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-92.pdf
- OpenAI, ChatGPT, for grammar and overall english check https://chatgpt.com