Python Fundamentals | Project 4: Python Project

Operation: Log Recon

Muhammad Termidzi Bin Azmi (S27)

CFC190324

James Lim

02 Jul 2024

TABLE OF CONTENTS

INTRODUCTION

Operation Log Recon is a project focused on developing a Python tool to meticulously analyze auth.log files. The primary aim is to extract, examine, and interpret the extensive information within these logs to gain insights into system operations, security, and anomalies. This initiative empowers cybersecurity teams with unparalleled insights, enhancing their ability to identify and mitigate risks, ensuring operational integrity and security resilience.

The tool will parse the auth.log file to extract command usage details, including timestamps, executing users, and the commands executed. It will also monitor user authentication changes by logging details of newly added or deleted users, password changes, and instances when users utilize su or sudo commands. Additionally, it will issue alerts for failed sudo attempts, ensuring comprehensive monitoring of system activities.

Operation Log Recon aims to transform raw log data into actionable intelligence, providing a clear understanding of system behaviors, security threats, and operational anomalies. This project ensures that the insights gained are effectively communicated to relevant stakeholders through detailed reports, enabling proactive risk identification and mitigation.

METHODOLOGIES

1. Function to read file content.

```
1  #!/usr/bin/python3
2
3  import time
4
5  # Function to read file content
6  def read_file_content(file_path):
7      with open(file_path, 'r') as file:  # Open the auth.log file in read-only mode and read its content
8          content = file.readlines()      # Read all lines in the file
9      return content
```

Methods used here:

- **(line 1) #!/usr/bin/python3**: Indicate which interpreter should be used to execute the script that follows.

- **(line 2) import time**: This line imports the *time* module, which provides various time-related functions. It is often used for creating delays, measuring time intervals, etc.

- **(line 6) def read_file_content(file_path):** This line defines a function named *read_file_content* that takes a single parameter, *file_path*. This parameter is expected to be a string representing the path to the file that you want to read.
    - **def:** a keyword used to define a function. Functions are blocks of reusable code that perform a specific task.

- **(line 7) with open(file_path, 'r') as file:** This line uses a with statement to open the file at *file_path* in read-only mode (`'r'`). The file is opened as a file object named *file*.
    - **with:** The with statement ensures that the file is properly closed after it finishes, even if an exception is raised.
    - **open(file_path, 'r'):** The *open* function is used to open a file and returns a file object.
    - **file_path:** This is a variable that holds the path to the file you want to open. It should be a string representing the file's location.
    - **open(file_path, 'r'):** This is the mode in which the file is opened. The 'r' mode stands for "read mode," meaning the file is opened for reading only.
    - **as file_path:** The as file part assigns the file object returned by open to the variable file.

- **(line 8) content = file.readlines()**: This line reads all the lines from the file and stores them in a list called content. Each element of the list is a string representing a line from the file.
    - **file:** This is the file object returned by the open function previously.
    - **readlines():** Method of the file object that reads all the lines of a file and returns them as a list of strings. Each string in the list represents a single line from the file.

- **content:** This is the variable that will store the list of strings returned by *file.readlines()*.

- **(line 9) return content:** This line returns the list content to the caller of the function. The function *read_file_content* will therefore provide the entire content of the specified file, with each line as a separate string in a list.

2. **Extracting command usage.**

```
11  # 1. Log Parse auth.log: Extract command usage.
12
13  # This function parses the auth.log file to extract command usage details.
14  def parse_auth_log(file_path):
15      content = read_file_content(file_path)
16      for line in content:                # Process each line to extract the required information
17              # Check if the line contains a sudo command execution
18              if 'COMMAND=' in line and 'sudo' in line:
19                  parts = line.split(';') # Split the line into parts
20                  # Extract the timestamp, user, and command
21
22                  # 1.1. Include the Timestamp.
23                  timestamp_parts = parts[0].strip().split()[:3]
24                  timestamp = ' '.join(timestamp_parts)
25
26                  #1.2. Include the executing user.
27                  user = [part.split('=')[1].strip() for part in parts if 'USER=' in part][0]
28
29                  # 1.3. Include the command.
30                  command = [part.split('=')[1].strip() for part in parts if 'COMMAND=' in part][0]
31
32                  # Print the extracted details
33                  print(f'[#] Timestamp: {timestamp}, User: {user}, Command: {command}')
```

Methods used here:

- **(line 14) def parse_auth_log(file_path):** This line defines a function named *parse_auth_log* that takes a single parameter, *file_path*.

- **(line 15) content = read_file_content(file_path):** This line calls the previously defined *read_file_content* function, passing *file_path* as an argument. It reads the content of the specified file and stores it in the variable content, which is a list of lines from the file.

- **(line 16) for line in content:** This line starts a loop that iterates over each line in the content list.
    - **for:** A for loop is used to iterate over a sequence, executing a block of code multiple times, once for each item in the sequence.
    - **line:** In the context of the for loop, line is a variable that represents the current item in the sequence during each iteration of the loop; single line from content list.
    - **in:** The in keyword is used to specify the sequence that the for loop will iterate over.

- **(line 18) if 'COMMAND=' in line and 'sudo' in line:** This line checks if the current line contains both the string COMMAND= and the string sudo. This is used to identify lines in the log that record the execution of a sudo command.
    - **if:** keyword is used to start a conditional statement in Python. It allows you to execute a block of code only if a specified condition is true.
    - **and:** The *and* operator is a logical operator that combines two conditions
- **(line 19) parts = line.split(';'):** Assigning results to variable *parts* .
    - **split(';'):** method in Python is used to split a string into a list of substrings based on a specified separator, `';'` .

- **(line 23) timestamp_parts = parts[0].strip().split()[:3] :** Extracts the first three whitespace-separated components from the first element of the *parts* list after removing any leading or trailing whitespace.
    - **parts[0]:** Optaining the first string of previosly split line.
    - **.strip():** Method is used to remove any leading and trailing whitespace characters from the string.
    - **.split()[:3]:** Split the element again, and slicing to select the first 3 elements.

- **(line 24) ' '.join(timestamp_parts):** joins together the elements in the timestamp_parts list into a single string with each element separated by a single space.

- **(line 27) user = [part.split('=')[1].strip() for part in parts if 'USER=' in part][0]:** This command clones the NIPE repository from GitHub to the local machine.
    - **=[<action if true><for loop><condition>]:** List comprehension, applies an expression to each element or subset of elements to create a new list.
    - **for part in parts:** This part iterates over each element part in the parts list.
    - **if 'USER=' in part:** This condition filters the elements of parts to only include those where the substring 'USER=' is present in part.
    - **part.split('=')[1].strip():** For each `part` that meets the condition `'USER=' in part`, this splits `part` into parts based on the `=` character and takes the second part (`[1]` index).

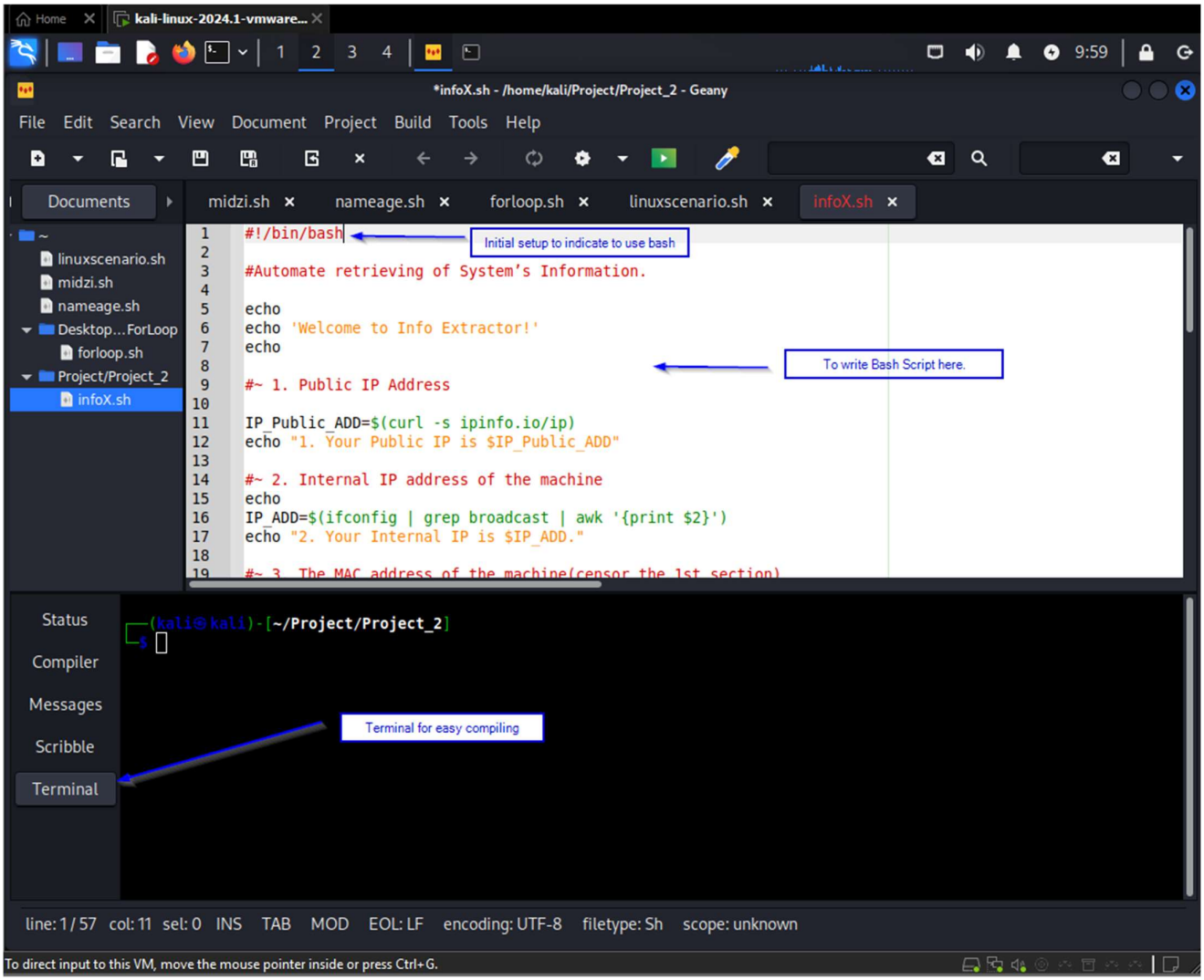
DISCUSSION

1. **Setting up Geany IDE**

- Using Geany - A free and open-source lightweight GUI text editor, including basic IDE feature.
- Download geany.
- Go into the intended folder to store the file.
- Type in “geany” into the linux command prompt followed by file name ending with .sh.

```
┌──(kali㉿kali)-[~/Project/Project_2]
└─$ geany infoX.sh
```

2. **Geany IDE navigation**



- Run with [**bash network research.sh**] on terminal.

3. Main Function

```
# This is the main fuction.
def main():
    file_path='/home/kali/Desktop/Project/Project_4/auth.log'
    #Please change file path accordingly to where auth.log is stored

    print('\n [*] Parsing the auth.log file to extract command usage details.\n')
    time.sleep(2)

    parse_auth_log(file_path)                          # Call the function with the path to your auth.log file
    time.sleep(2)

    print('\n [*] List of New Users \n')               # Call the function to list new users
    monitor_new_users(file_path)
    time.sleep(2)

    print('\n [*] List of Deleted Users \n')           # Call the function to list deleted users
    monitor_deleted_users(file_path)
    time.sleep(2)

    print('\n [*] List of Password Changes \n')        # Call the function to list password changes
    monitor_passwd_changes(file_path)
    time.sleep(2)

    print('\n [*] List of Attempted su Usage \n')      # Call the function to list attempted su usage
    monitor_su(file_path)
    time.sleep(2)

    print('\n [*] List of Attempted SUDO Commands \n') # Call the function to list attempted sudo commands usage
    monitor_sudo_users(file_path)

    time.sleep(1)
    print('\n *********** END OF SCRIPT ***********')

# Executing the main function.
# This conditional statement checks if the script is being run directly as the main programme.
if __name__ == '__main__':
    main()
```

- Main function is defined and called.
- File path is defined accordingly to where auth.log is stored.
- Here, the other functions are invoked such as:
    - *parse_auth_log(file_path)*
    - *monitor_new_users(file_path)*
    - *monitor_deleted_users(file_path)*
    - *monitor_passwd_changes(file_path)*
    - *monitor_su(file_path)*
    - *monitor_sudo_users(file_path)*
- Further explanation of each function below.

4. Function to Read auth.log File Content

```
import time

# Function to read file content
def read_file_content(file_path):
    with open(file_path, 'r') as file:  # Open the auth.log file in read-only mode and read its content
        content = file.readlines()      # Read all lines in the file
    return content
```

- Here we import the time module, to be called later with *sleep()*, to pause and give users a chance to read the data churned.

- A function to read the auth.log files are also provided. We create a function so it can be called upon multiple times for other functions to work upon.

5. Extracting Command Usage

```
11  # 1. Log Parse auth.log: Extract command usage.
12
13  # This function parses the auth.log file to extract command usage details.
14  def parse_auth_log(file_path):
15      content = read_file_content(file_path)
16      for line in content:                # Process each line to extract the required information
17              # Check if the line contains a sudo command execution
18              if 'COMMAND=' in line and 'sudo' in line:
19                  parts = line.split(';') # Split the line into parts
20                  # Extract the timestamp, user, and command
21
22                  # 1.1. Include the Timestamp.
23                  timestamp_parts = parts[0].strip().split()[:3]
24                  timestamp = ' '.join(timestamp_parts)
25
26                  #1.2. Include the executing user.
27                  user = [part.split('=')[1].strip() for part in parts if 'USER=' in part][0]
28
29                  # 1.3. Include the command.
30                  command = [part.split('=')[1].strip() for part in parts if 'COMMAND=' in part][0]
31
32                  # Print the extracted details
33                  print(f'[#] Timestamp: {timestamp}, User: {user}, Command: {command}')
```

- Here we define a function, *parse_auth_log*, to extract command usage from auth.log
- We go through line by line of auth.log. First, we filter for lines containing *'COMMAND='* and *'sudo'*. These lines contain command usage in auth.log.

```
Jun 30 03:36:39 server sudo:     tc : TTY=pts/1 ; PWD=/home/tc ; USER=root ; COMMAND=/usr/bin/netstat -tapn
```

- Next, we split the line according to ' ; ', and we work on the parts accordingly.
-

```
Jun 30 03:36:39 server sudo:     tc : TTY=pts/1 ; PWD=/home/tc ; USER=root ; COMMAND=/usr/bin/netstat -tapn
```

- **(line 23)** For the item in the orange box, we split them further by space, clean any whitespace, and take the first 3 items.
- Next, we join them back with a spacing. This will be the timestamp component.

```
Jun 30 03:36:39
```

- **(line 27)** For the item in the blue box, we cannot use the same coding as the *timestamp_parts* due to the inconsistencies of the value in parts[2].
- Therefore, we use a list comprehension to strictly match *USER=* to get the username.
- **(line 30)** Lastly, we extract the *command* using the same method as finding *user*.

```
┌──(kali㉿kali)-[~/Desktop/Project/Project_4]
└─$ python3 Project4.py

 [*] Parsing the auth.log file to extract command usage details.

[#] Timestamp: Jun 30 03:35:51, User: root, Command: /usr/bin/mkdir
[#] Timestamp: Jun 30 03:36:26, User: root, Command: /usr/bin/netstat -tapn
[#] Timestamp: Jun 30 03:36:39, User: root, Command: /usr/bin/netstat -tapn
[#] Timestamp: Jun 30 04:29:57, User: root, Command: /bin/bash
[#] Timestamp: Jun 30 04:30:04, User: root, Command: /usr/sbin/tc
[#] Timestamp: Jun 30 04:30:43, User: root, Command: /usr/sbin/adduser rogue_user
[#] Timestamp: Jun 30 04:31:56, User: root, Command: /usr/bin/netstat -tapn
[#] Timestamp: Jun 30 04:33:24, User: root, Command: /bin/bash
[#] Timestamp: Jun 30 04:34:02, User: root, Command: /usr/sbin/userdel -r rogue_user
[#] Timestamp: Jun 30 04:34:34, User: root, Command: /usr/bin/kill -9 102600
[#] Timestamp: Jun 30 04:34:56, User: root, Command: /usr/sbin/userdel -r rogue_user
```

6. Monitor User Authentication Changes

- Log Parse auth.log: Monitor user authentication changes
- Breakdown of parts below.

7. Print Details of Newly Added Users, Including the Timestamp

```python
# 2.1. Print details of newly added users, including the Timestamp
def monitor_new_users(file_path):
    content = read_file_content(file_path)
    for line in content:
            if 'new user' in line:
                parts = line.split(';') # Split the line into parts
                # Extract the timestamp, user

                # 1.1. Include the Timestamp.
                timestamp_parts = parts[0].strip().split()[:3]
                timestamp = ' '.join(timestamp_parts)

                #1.2. Include the New User.
                user = [part.split('=')[1].split(',')[0].strip() for part in parts if 'name=' in part][0]

                # Print the extracted details
                print('[#] New user added:')
                print(f'[#] Timestamp: {timestamp}, New User: {user}')
                print(f'[#] {line}')
                time.sleep(2)
```

- Here we create a *monitor_new_users* function to check if 'new users' are available in each line of the auth.log, and to print them.
- Next, we parse on the line to extract information using the same method as above for *timestamp* and *user*, with the addition of *split(';')* to clean the username.
- The *time.sleep(2)* was added so there is a 2 second delay for user to read the churned information.

```
 [*] List of New Users

[#] New user added:
[#] Timestamp: Jun 30 04:30:43, New User: rogue_user
[#] Jun 30 04:30:43 server useradd[102576]: new user: name=rogue_user, UID=1005, GID=1004, home=/home/rogue_user, shell
=/bin/bash, from=/dev/pts/2
```

8. Print Details of Deleted Users, Including the Timestamp

```
58  # 2.2. Print details of deleted users, including the Timestamp.
59  def monitor_deleted_users(file_path):
60      content = read_file_content(file_path)
61      for line in content:
62          if 'delete user' in line:
63              # Extract the timestamp
64              parts = line.split()                    # Split the line into parts
65              timestamp_parts = parts[:3]             # Get the first three parts
66              timestamp = ' '.join(timestamp_parts)   # Join them to form the timestamp
67
68              # Extract the deleted user
69              user_part = line.split("'")[1]          # Get the part between single quotes
70
71              # Print the extracted details
72              print(f'[#] Timestamp: {timestamp}, Deleted User: {user_part}')
73              print(f'[#] Details:\n {line}')
74              time.sleep(2)
```

- Here we create a *monitor_deleted_users* function to check if 'deleted users' are available in each line of the auth.log, and to print them.

```
 [*] List of Deleted Users

[#] Timestamp: Jun 30 04:34:56, Deleted User: rogue_user
[#] Details:
 Jun 30 04:34:56 server userdel[102783]: delete user 'rogue_user'
```

9. Print Details of Users Changing Password, Including the Timestamp

```
76  # 2.3. Print details of changing passwords, including the Timestamp.
77  def monitor_passwd_changes(file_path):
78      content = read_file_content(file_path)
79      for line in content:
80              if 'password changed for' in line:
81              # Extract the timestamp
82                  parts = line.split()
83                  timestamp_parts = parts[:3]
84                  timestamp = ' '.join(timestamp_parts)
85
86              # Extract the user whose password was changed
87                  user_part = line.split('password changed for ')[1].strip().split()[0]
88
89              # Print the extracted details
90                  print(f'[#] Timestamp: {timestamp}, Password Changed for User: {user_part}')
91                  print(f'[#] Details:\n {line}')
92                  time.sleep(2)
```

- Here we create a *monitor_passwd_changes* function to check if 'password changes' are available in each line of the auth.log, and to print them.

```
 [*] List of Password Changes

[#] Timestamp: Jun 30 04:30:53, Password Changed for User: rogue_user
[#] Details:
 Jun 30 04:30:53 server passwd[102587]: pam_unix(passwd:chauthtok): password changed for rogue_user

[#] Timestamp: Jun 30 04:33:03, Password Changed for User: rogue_user
[#] Details:
 Jun 30 04:33:03 server passwd[102634]: pam_unix(passwd:chauthtok): password changed for rogue_user
```

10. Print Details of Users Using SU Command, Including the Command

```
94   # 2.4. Print details of when users used the su command.
95  def monitor_su(file_path):
96      content = read_file_content(file_path)
97      for line in content:
98              if(
99                  'session opened for user' in line and          # to filter su session opened for user
100                 'server su' in line and
101                 'server sudo' not in line and
102                 'session closed for user' not in line
103             ):
104                 # Extract the timestamp
105                 timestamp_parts = line.split()[:3]
106                 timestamp = ' '.join(timestamp_parts)
107                 # Extract the user who opened the session
108                 user = line.split(' by ')[-1].split('(')[0].strip()
109                 # Extract the command
110                 command = line.split(': ')[-1].strip()
111                 # ~ print('[#] Su command used: ')
112                 print(f'[#] Timestamp: {timestamp}, User: {user}, Command: {command}')
113                 print(f'[#] Details: \n{line}\n')
```

- Here we create a *monitor_passwd_changes* function to check if 'su commands' are available in each line of the auth.log, and to print them.
- Multiple filters were needed to extract the exact su command.

```
 [*] List of Attempted su Usage

[#] Timestamp: Jun 30 04:31:30, User: tc, Command: session opened for user rogue_user(uid=1005) by tc(uid=10
[#] Details:
Jun 30 04:31:30 server su: pam_unix(su:session): session opened for user rogue_user(uid=1005) by tc(uid=1000


[#] Timestamp: Jun 30 04:32:10, User: tc, Command: session opened for user tc(uid=1000) by tc(uid=1005)
[#] Details:
Jun 30 04:32:10 server su: pam_unix(su:session): session opened for user tc(uid=1000) by tc(uid=1005)


[#] Timestamp: Jun 30 04:32:21, User: tc, Command: session opened for user rogue_user(uid=1005) by tc(uid=10
[#] Details:
Jun 30 04:32:21 server su: pam_unix(su:session): session opened for user rogue_user(uid=1005) by tc(uid=1000


[#] Timestamp: Jun 30 04:33:11, User: tc, Command: session opened for user tc(uid=1000) by tc(uid=1005)
[#] Details:
Jun 30 04:33:11 server su: pam_unix(su:session): session opened for user tc(uid=1000) by tc(uid=1005)
```

11. Print Details of Users Using SUDO Command, Including the Command

```python
# 2.5. Print details of users who used the sudo; include the command.
# 2.6. Print ALERT! If users failed to use the sudo command; include the command.
def monitor_sudo_users(file_path):
    content = read_file_content(file_path)
    for line in content:
            if 'COMMAND=' in line:
                parts = line.split(';')
                # Extract the timestamp
                timestamp_parts = parts[0].strip().split()[:3]
                timestamp = ' '.join(timestamp_parts)
                # Extract the user who used SUDO
                user = [part.split('=')[1].strip() for part in parts if 'USER=' in part][0]
                # Extract the command
                command = [part.split('=')[1].strip() for part in parts if 'COMMAND=' in part][0]
                # Check if it's a failed sudo attempt
                if 'sudo' in line and 'incorrect password attempts' in line:
                    print(f'[!] ALERT! Failed sudo attempt: \n Timestamp: {timestamp}, User: {user}, Command: {command}')
                    print(f'Details: \n {line} \n')
                else:
                    print(f'[#] Timestamp: {timestamp}, User: {user}, Command: {command}')
                    print(f'Details: \n {line} \n')
                time.sleep(1)
```

- Here we create a *monitor_sudo_users* function to check if 'COMMAND=' are available in each line of the auth.log, and to print them
- **(line 132)** We also filter for 'Failed sudo attempt', with Alerts!

```
 [*] List of Attempted SUDO Commands

[!] ALERT! Failed sudo attempt:
 Timestamp: Jun 30 03:35:51, User: root, Command: /usr/bin/mkdir
Details:
 Jun 30 03:35:51 server sudo:       tc : 3 incorrect password attempts ; TTY=pts/1 ; PWD=/home/tc ; USER=root ; COMMAND
=/usr/bin/mkdir


[!] ALERT! Failed sudo attempt:
 Timestamp: Jun 30 03:36:26, User: root, Command: /usr/bin/netstat -tapn
Details:
 Jun 30 03:36:26 server sudo:       tc : 3 incorrect password attempts ; TTY=pts/1 ; PWD=/home/tc ; USER=root ; COMMAND
=/usr/bin/netstat -tapn


[#] Timestamp: Jun 30 03:36:39, User: root, Command: /usr/bin/netstat -tapn
Details:
 Jun 30 03:36:39 server sudo:       tc : TTY=pts/1 ; PWD=/home/tc ; USER=root ; COMMAND=/usr/bin/netstat -tapn


[#] Timestamp: Jun 30 04:29:57, User: root, Command: /bin/bash
Details:
 Jun 30 04:29:57 server sudo:       tc : TTY=pts/1 ; PWD=/root ; USER=root ; COMMAND=/bin/bash


[#] Timestamp: Jun 30 04:30:04, User: root, Command: /usr/sbin/tc
Details:
 Jun 30 04:30:04 server sudo:     root : TTY=pts/2 ; PWD=/root ; USER=root ; COMMAND=/usr/sbin/tc


[#] Timestamp: Jun 30 04:30:43, User: root, Command: /usr/sbin/adduser rogue_user
Details:
 Jun 30 04:30:43 server sudo:       tc : TTY=pts/1 ; PWD=/home ; USER=root ; COMMAND=/usr/sbin/adduser rogue_user
```

CONCLUSION

Operation Log Recon has successfully achieved its objectives of parsing and analyzing critical data from the auth.log files, aimed at enhancing cybersecurity measures and operational integrity. The project's primary goal was to create a specialized Python tool to extract, examine, and understand information hidden within log files, particularly focusing on command usage and user authentication changes.

By meticulously analyzing command usage, including timestamps and executing users, and also alerting for changes or failed attempts, the tool provides a detailed view of system activities:

Behavioral Analysis: Understanding command usage patterns helps cybersecurity personnel identify normal and abnormal behaviors within the system. This allows them to detect suspicious activities such as unauthorized commands or unusual usage patterns that may indicate malicious intent.

Threat Detection: By monitoring command execution, cybersecurity teams can detect potential indicators of compromise (IOCs) or signs of an ongoing attack. This proactive approach enables early detection and mitigation of security threats before they cause significant damage.

Incident Response: Detailed command logs provide valuable forensic evidence during incident response. They allow investigators to reconstruct the sequence of events leading to a security incident, determine the scope of compromise, and identify the root cause of security breaches.

Privilege Management: Analyzing command usage helps in monitoring and managing user privileges effectively. It allows administrators to ensure that users are adhering to least privilege principles and are not performing unauthorized actions that could compromise system security.

Compliance and Auditing: Many compliance regulations and standards require organizations to maintain detailed logs of command execution for auditing purposes. Analyzing these logs helps ensure compliance with regulatory requirements and provides evidence of adherence to security policies.

Continuous Improvement: Insights gained from command usage analysis can be used to refine security policies, improve incident response procedures, and enhance overall cybersecurity posture. This iterative process helps organizations stay resilient against evolving threats.

In summary, the detailed view of system activities provided by analyzing command usage is beneficial for cybersecurity professionals as it supports threat detection, incident response readiness, privilege management, compliance with regulations, and ongoing improvement of security practices.

RECOMMENDATIONS

Security Enhancements:

Automation, Integration and Continuous Monitoring:

Integrate the log analysis tool seamlessly with existing security information and event management (SIEM) systems to establish a cohesive cybersecurity ecosystem. By integrating with SIEM, the log analysis tool can aggregate and correlate data from multiple sources, providing a unified view of security events and anomalies across the organization. This integration enables real-time alerts and automated incident response actions, enhancing operational efficiency and reducing response times to potential threats. Furthermore, leveraging SIEM capabilities allows for centralized monitoring and comprehensive reporting, facilitating proactive security measures and ensuring alignment with organizational security policies and regulatory requirements.

Behavioral Analytics:

Harness the power of advanced machine learning algorithms to conduct behavioral analytics on command usage patterns within the system. By analyzing historical and real-time data from auth.log, machine learning algorithms can identify deviations from normal behavior that may indicate potential security threats. Behavioral analytics enable proactive threat hunting by automatically detecting anomalous activities, such as unusual command sequences or privileged user actions, that could signify malicious intent or compromised accounts. This proactive approach strengthens the organization's defense posture by enabling early detection and response to insider threats, unauthorized access attempts, and other sophisticated cyberattacks. Integrating behavioral analytics into the log analysis framework enhances the accuracy and efficiency of threat detection efforts, empowering cybersecurity teams to mitigate risks effectively and safeguard critical assets.

REFERENCES

1. OpenAI. ChatGPT. Personal communication. Jul 1, 2024.

2. Gemini (Google AI, 2024). Personal communication, Jul 1, 2024.

3. Microsoft Copilot. Microsoft. Personal communication. Jul 1, 2024

4. SIEM Integration with Python: Boosting Security in a Data-Driven World
   https://www.cm-alliance.com/cybersecurity-blog/siem-integration-with-python-boosting-security-in-a-data-driven-world

5. How Python Can Benefit Your Cybersecurity Efforts?
   https://medium.com/@m2739345/how-python-can-benefit-your-cybersecurity-efforts-65ac355e9a31#:~:text=Automated%20dynamic%20analysis%20through%20Python,understand%20the%20threat%20landscape%20effectively.

6. Python for Cybersecurity
   https://medium.com/@AlexanderObregon/python-for-cybersecurity-adcbfe15600a

7. Python List Comprehension
   https://www.programiz.com/python-programming/list-comprehension