

Python Fundamentals | Project: Log Analyzer

Kenneth Wong s32

CFC190324

Trainer: James

Table of Contents

Introduction.....	2
Methodologies.....	2-7
Discussion.....	8
Conclusion.....	8-9
Recommendation.....	9
References.....	10

Introduction

This report outlines the analysis and extraction of critical user activity data from the auth.log file on a Linux system. The auth.log file serves as a vital repository for recording authentication-related events and security activities crucial to system administration and security monitoring. It captures a wide range of activities, including user login attempts, both successful and unsuccessful, which are pivotal for detecting unauthorized access attempts and potential security breaches. Additionally, the log documents the use of commands like 'sudo' and 'su', detailing when users execute privileged operations or switch to other user accounts. This audit trail extends to user management actions such as adding or deleting user accounts and changing passwords, facilitating comprehensive oversight of user-related activities.

The script developed for this purpose parses the auth.log to achieve several objectives.

Firstly, it extracts command usage by including the timestamp to capture the exact time a command was executed, identifying the executing user, and logging the specific command executed. This information is crucial for auditing and tracking user activities on the system.

Secondly, the script monitors user authentication changes. It prints details of newly added users, including the timestamp to track when new users are created. It also logs details of deleted users, recording when users are removed from the system. Additionally, it monitors changes in user passwords with associated timestamps, providing insights into user password management.

Lastly, the script details when users utilize the su command and tracks the use of sudo, highlighting successful and failed attempts. For failed sudo attempts, it prominently outputs an alert to ensure immediate attention. This comprehensive monitoring aids in maintaining a secure and well-audited system environment.

Methodologies

For this script, we decided to go with more modular functions that target specific user-related events, ensuring both clarity and easy modifications in future.

We began by writing distinct functions for each type of event of interest; new user additions, user deletions, password changes, user switches using the su command, successful sudo commands, and failed sudo attempts. Each function parses the log file line by line, extracting relevant details such as timestamps, user names, process IDs, and commands.

Figure 1: Reading the log file

```
179 # ~ Open the auth.log file in readlines mode
180 with open('/var/log/auth.log', 'r') as file:
181     # Read all lines from the file
182     data = file.readlines()
```

The script begins by opening the 'auth.log' file in read mode by providing its absolute file path. Then, it proceeds to read all lines in it and stores into a list called 'data' with the 'readline()' method.

Figure 2: Cleaning up the data

```

184 # ~ Creating a new list to store the new cleaned data
185 cleandata=[]
186
187 print('Below are the commands used:')
188 print('')
189
190 # ~ Stripping the newline characters
191 for eachline in data:
192     cleanentry = eachline.strip('\n')
193     cleandata.append(cleanentry)

```

The list is then processed to remove newline characters, resulting in a cleaned list called 'cleandata'.

To handle the diverse types of log entries and ensure robustness, I implemented conditional checks and loops within each function. These checks ensure that only relevant log entries are processed, and loops help in identifying specific patterns within the entries.

Figure 3: Command execution tracking and logging

```

195 # ~ Identifying the lines with 'COMMAND' and processing them
196 for eachline in cleandata:
197     if 'COMMAND' in eachline:
198         # ~ breaks the line into separate items in a list
199         parts = eachline.split()
200         # ~ indicates the index of timestamp
201         timestamp = parts[0]
202         # ~ locates user and stores it in variable
203         user=parts[3]
204         # ~ Calculate index where the command starts
205         command_index = eachline.index('COMMAND')+len('COMMAND=')
206         # ~ Extract the command from the line
207         command=eachline[command_index:]
208         # ~ Split the command by '/'
209         command_parts=command.split('/')
210         # ~ Rejoin the command parts
211         command='/'.join(command_parts[0:])
212         print('The command used is', command, 'and it was ran at ', timestamp, 'by', user)
213

```

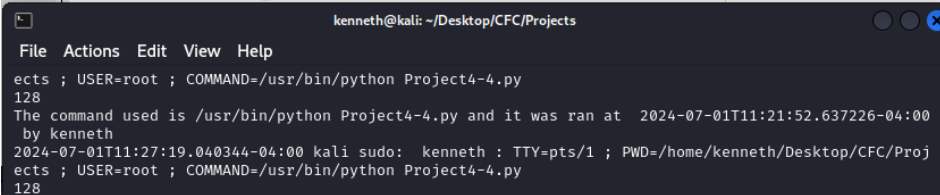
In this segment of the script, each line in cleaned log data is examined to identify entries that contain the word 'COMMAND'. When found, the script splits the line into individual components and then storing them into another list variable named 'parts'. The timestamp of the command execution is extracted from the first element of the list while the username is taken from the fourth element [3]. The script then calculates the starting index of the command within the line by locating the string 'COMMAND=' and adding its length to 'command_index'.

Figure 4: Printing 'command_index' to show how it works

```

205 command_index = eachline.index('COMMAND')+len('COMMAND=')
206 print(eachline)
207 print(command_index)

```



```

kenneth@kali: ~/Desktop/CFC/Projects
File Actions Edit View Help
ects ; USER=root ; COMMAND=/usr/bin/python Project4-4.py
128
The command used is /usr/bin/python Project4-4.py and it was ran at 2024-07-01T11:21:52.637226-04:00
by kenneth
2024-07-01T11:27:19.040344-04:00 kali sudo: kenneth : TTY=pts/1 ; PWD=/home/kenneth/Desktop/CFC/Proj
ects ; USER=root ; COMMAND=/usr/bin/python Project4-4.py
128

```

After which, the command is extracted from the starting position to the end of the string. To ensure proper formatting, the command string is split by '/' and then rejoined, which essentially fixes any path issues which was faced initially. Finally, the script prints a message which shows the extracted command, timestamp, and user who executed it.

Figure 5: Define function 'newusers(data)'

```

1  #!/usr/bin/python3
2
3  # ~ function for new users added
4  def newusers(data):
5      for line in data:
6          if 'new user' in line:
7              # ~ breaks the line into separate items in a list
8              parts = line.split()
9              # ~ indicates the index of timestamp
10             timestamp = parts[0]
11             # ~ searches for 'new' in items in the list(parts)
12             if 'new' in parts:
13                 # ~ stores the index of [3] where 'new' is
14                 user_index = parts.index('new')
15                 # ~ locates user which is 2 index after 'new', removes ',' and 'name='
16                 userlesscomma = parts[user_index + 2].strip(',')
17                 user=userlesscomma.strip('name=')
18
19             # ~ Check for 'UID'
20             for item in parts:
21                 # ~ Find the item that starts with 'UID='
22                 if item.startswith('UID='):
23                     UID = item.split('=')[1].strip(',')
24                     # ~ Exit loop once 'UID' found
25                     break
26
27             # ~ Check for 'GID'
28             for item in parts:
29                 if item.startswith('GID='):
30                     GID = item.split('=')[1].strip(',')
31                     # ~ Exit loop once 'GID' found
32                     break
33
34             # ~ Check for 'home'
35             for item in parts:
36                 if item.startswith('home='):
37                     home_directory = item.split('=')[1].strip(',')
38                     # ~ Exit loop once 'home' found
39                     break
40
41             # ~ Check for 'process ID'
42             for item in parts:
43                 if item.startswith('useradd['):
44                     processid = item.strip(':')
45             print('New user added:', user)
46             print('Time:', timestamp)
47             print('UID:', UID)
48             print('GID:', GID)
49             print('Home directory:', home_directory)
50             # ~ Prints an empty line for readability
51             print('Process ID:', processid)
52             print('')
53         else:
54             print('No new user added')
55

```

The above function processes log data to identify and extract details on new users added into the system. The function iterates each line, looking for the string 'new user' and then splits the parts with the split() function. Within these components, the word 'new' is located and index of 'new=test' is identified subsequently adding two positions after 'new'. The prefix 'name=' is then removed to obtain the username.

Figure 6: Explaining why 'new' is identified

```

Newly added users:
['2024-06-29T07:26:13.712112-04:00', 'kali', 'useradd[3842]:', 'new', 'user:', 'name=test,', 'UID=100
2,', 'GID=1002,', 'home=/home/test,', 'shell=/bin/bash,', 'from=/dev/pts/1']

```

Next, the function checks for 'UID', 'GID,' home' and 'process ID' and extracts only the relevant component without the prefixes and assigns them to the corresponding variables.

Finally, the function prints out the details along with the details extracted. If no user is found, it will print 'No new user added.'

Figure 7: Define function 'delusers(data)'

```
57 # ~ function for deleted users
58 def delusers(data):
59     for line in data:
60         if 'delete user' in line:
61             # ~ breaks the line into separate items in a list
62             parts = line.split()
63             # ~ indicates the index of timestamp
64             timestamp=parts[0]
65             if 'user' in parts:
66                 # ~ stores the index of [5] where 'user' is
67                 user_index = parts.index('user')
68                 # ~ locates user which is 1 index after 'new', removes "'"
69                 user = parts[user_index + 1].strip("'")
70                 # ~ locates the process id and removes ':'
71                 for item in parts:
72                     if item.startswith('userdel['):
73                         processid = item.strip(':')
74                 print('User deleted:', user)
75                 print('Time:', timestamp)
76                 print('Process ID:', processid)
77                 print('')
```

The function above identifies and extract details on deleted users, processing each line of the log data to locate entries which reflect user deletion.

For each line containing 'delete user', the function extracts timestamp from the initial component of the split line. It then searches for the keyword 'user'. Upon finding it, the function determines the username by examining the component immediately following 'user' and removes the single quote to get the actual username.

Next, the function searches for the process ID and it looks for strings that start with the prefix 'userdel[' to retrieve the process ID while removing the colon.

Lastly, this prints out details of the deleted user along with the relevant information. This overall approach ensures that the function accurately extracts the information. This methodology involves careful string manipulation and the use of conditional statements and loop to thoroughly examine each line to capture the details.

Figure 8: Define function 'passwdchanged(data)'

```
79 # ~ function for changed password
80 def passwdchanged(data):
81     for line in data:
82         if 'password changed' in line:
83             # ~ breaks the line into separate items in a list
84             parts = line.split()
85             # ~ indicates the index of timestamp
86             timestamp = parts[0]
87             # ~ locates user which the last element of the list and assigns it to user
88             user = parts[-1]
89             # ~ locates processid and assigns it to processid with ':' removed
90             processid=parts[2].strip(':')
91
92             print('Password changed for:', user)
93             print('Time:', timestamp)
94             print('Process ID:', processid)
95             print('')
```

Figure 9: Define function 'switchuser(data)'

```

97 # ~ function for switched users
98 def switchuser(data):
99     for line in data:
100         if 'su:session' in line:
101             # ~ breaks the line into separate items in a list
102             parts = line.split()
103             # ~ indicates the index of timestamp
104             timestamp = parts[0]
105             # ~ locates users and assign them to their respective variables
106             newuser = parts[-3]
107             olduser = parts[-1]
108             # ~ locates processid and assigns it to processid with ':' removed
109             processid = parts[2].strip(':')
110             print(olduser, 'switched to', newuser)
111             print('Time:', timestamp)
112             print('Process ID:', processid)
113             print('')
114         elif 'authentication failure' and 'su:' in line:
115             # ~ breaks the line into separate items in a list
116             parts = line.split()
117             # ~ indicates the index of timestamp
118             timestamp = parts[0]
119             # ~ locates users and assign them to their respective variables
120             ruser = parts[-3]
121             user = parts[-1]
122             print('Authentication failure occurred')
123             print(ruser, 'attempted to switch to user', user)
124             print('Time:', timestamp)
125             print('')
126

```

Both the 'passwdchanged(data)' and the 'switchuser(data)' functions perform rather similarly. They both searches for a respective string; 'password changed' and 'su:session' and then splits the line into parts to extract its elements and assigns them to variables. It then splits the line into parts to extract the timestamp from the first element, the username from the last element, and the process ID from the third element, stripping any colons. It prints the username, timestamp, and process ID of the password change. And prints the old, new user, timestamp and process ID of a user switch occurring.

In the case of authentication failures, the function looks for lines containing 'authentication failure' and 'su:', then splits the line, extracts the timestamp, the real user ruser, and the target user. It prints these details to indicate an attempted but failed user switch. This dual functionality helps in monitoring both successful and unsuccessful user switch attempts in the system.

Figure 10: Define function 'sudo(data)'

```

128 # ~ function for use of sudo
129 def sudo(data):
130     for line in data:
131         if 'USER=' and 'COMMAND' in line:
132             # ~ breaks the line into separate items in a list
133             parts=line.split()
134             # ~ indicates the index of timestamp
135             timestamp = parts[0]
136             # ~ locates user and stores it in variable
137             user=parts[3]
138             # ~ Calculate index where the command starts
139             command_index = line.index('COMMAND=')+len('COMMAND=')
140             # ~ Extract the command from the line
141             command=line[command_index:]
142             # ~ Split the command by '/'
143             command_parts=command.split('/')
144             # ~ Rejoin the command parts
145             command='/'.join(command_parts[0:])
146             print('Time:', timestamp)
147             print('User:', user)
148             print('Command used:', command)
149             print('')

```

Figure 11: Define function 'failedsudo(data)'

```
151 # ~ function for failed use of sudo
152 def failedsudo(data):
153     for line in data:
154         if 'incorrect' in line:
155             # ~ breaks the line into separate items in a list
156             parts=line.split()
157             # ~ indicates the index of timestamp
158             timestamp = parts[0]
159             # ~ locates user and stores it in variable
160             user=parts[3]
161             # ~ Calculate index where the command starts
162             command_index = line.index('COMMAND=')+len('COMMAND=')
163             # ~ Extract the command from the line
164             command=line[command_index:]
165             # ~ Split the command by '/'
166             command_parts=command.split('/')
167             # ~ Rejoin the command parts
168             command='/'.join(command_parts[0:])
169             # ~ display the below 'ALERT!' message more prominently
170             print('*' * 30)
171             print('*          ALERT!          *')
172             print('* FAILED SUDO ATTEMPT DETECTED *')
173             print('*' * 30)
174             print('Timestamp:', timestamp)
175             print('User:', user)
176             print('Command:', command)
177             print('')
```

The sudo(data) function identifies and logs the usage of the sudo command by extracting and printing relevant details from lines containing 'USER=' and 'COMMAND='. It isolates the timestamp, user, and command executed by splitting the line into parts. The function calculates the starting index of the command, extracts it, and cleans up its format before printing it alongside the timestamp and user. This helps monitor commands executed with elevated privileges, providing insight into user activities and system modifications.

The failedsudo(data) function detects failed sudo attempts by searching for lines containing 'incorrect', indicating an unsuccessful sudo command execution. The line is split to extract the timestamp, user, and command, similar to the sudo function. An alert message is prominently displayed to highlight the failed attempt, along with the relevant details.

Discussion

1. `split()`

The `split()` method is used to break each line into separate items based on whitespace, creating a list of parts for easier access to individual elements like timestamps, usernames, and commands.

2. `join()`

The `join()` method is used to reassemble the command parts after splitting them by '/' to ensure they are displayed in the correct format. This method concatenates the elements of a list into a single string with '/' as the separator.

3. `startswith()`

The `startswith()` method checks if a string begins with a specified substring. This is used to identify items in the log lines that contain certain keywords, such as 'UID=', 'GID=', and 'useradd['.

4. `strip()`

The `strip()` method removes leading and trailing characters from a string. In the script, it is used to remove unnecessary characters like ':' and ',' from parts of the log entries.

5. `index()`

The `index()` method returns the position of a specified value in a list. It is used to find the position of keywords such as 'new', 'user', and 'COMMAND=' within the log entry parts.

Conclusion

During development, we encountered several troubleshooting challenges. Handling different log entry formats required careful consideration to avoid false positives and ensure accurate data extraction. Debugging was facilitated by printing and executing the script throughout and closely inspecting log entries to identify patterns and anomalies.

This script will be able to significantly improve efficiency by providing detailed information and analysis on user activities. Monitoring sudo usage is crucial because it allows users to execute commands with escalated privileges. Being able to track it ensures that only authorised users perform high-privilege actions which is important as this can lead to security vulnerabilities and system compromise. By scripting this, it allows for quicker detection and response if needed.

Tracking failed login attempts is another important aspect of this script. Failed login attempts can indicate brute force attacks or unauthorised access. Monitoring these helps to identify breaches. Frequent failed attempts may indicate that the account is at risk, allowing for immediate action to secure the account.

Monitoring new user creation ensures that no unauthorised users are added which is yet another security risk. From a workplace perspective, ensuring that every new user added complies with security policies and guidelines is good practice. Similarly, tracking user deletion ensures that they are legitimate. This could potentially play a part in confidentiality, integrity and availability concerns.

In addition, it could also tell that there are malicious actors covering up their traces by deleting accounts.

Unauthorised password changes could be an indicator that the account has been compromised and detecting such changes quickly can help in securing the affected account.

When an attacker is trying to gain elevated privileges, they would try to use sudo logins. Monitoring such attempts can help mitigate potential threats. Tracking these ensures that users are using their privileges correctly.

In conclusion, this script plays an important role by providing insights into these critical components and comprehensive monitoring is essential for a cybersecurity practitioner in safeguarding the environment.

Recommendation

While developing this script, we faced significant challenges related to debugging and troubleshooting. The script's behavior was inconsistent, sometimes functioning correctly without any changes being made, which made it difficult to identify and resolve issues. To improve the script's reliability and maintainability, we recommend adopting a modular approach.

Modularization involves dividing the script into smaller, reusable functions, each responsible for a specific task. This approach offers several benefits. Firstly, it enhances readability. When the script is broken down into well-defined functions, it becomes easier to read and understand, which is particularly beneficial for anyone who may work on the script in the future. Secondly, modularization facilitates easier maintenance. Changes can be made to individual functions without impacting the rest of the script, reducing the likelihood of introducing new errors and simplifying the update process. This approach not only improves the script's overall quality but also makes it more manageable and robust.

Furthermore, modularization allows for easier debugging. By testing and debugging functions independently, issues can be identified and resolved more efficiently. This leads to more consistent and reliable performance.

To enhance the utility of this script, the script can be extended with additional functions to handle more varieties of information `auth.log` has, such as changes to user groups. Future versions of the script could be modified to support analyzing other kinds of logs as well. By adapting the script to parse and analyze other logs such as web server logs, it could provide insight from a security and user behaviour.

Additionally, the script could be integrated with a database to store the parsed log data to provide more in-depth analysis and comparisons.

References

"How to get a string after a specific substring," Stack Overflow, September 24, 2012, retrieved July 1, 2024, from <https://stackoverflow.com/questions/12572362/how-to-get-a-string-after-a-specific-substring>.

"How to remove items from list and make items join," Stack Overflow, July 8, 2021, retrieved July 1, 2024, from <https://stackoverflow.com/questions/68302430/how-to-remove-items-from-list-and-make-items-join>.

"Checking whether a string starts with XXXX," Stack Overflow, January 10, 2012, retrieved July 1, 2024, from <https://stackoverflow.com/questions/8802860/checking-whether-a-string-starts-with-xxxx>.

TechTarget. "Sudo (superuser do)," retrieved July 2, 2024, from <https://www.techtarget.com/searchsecurity/definition/sudo-superuser-do#:~:text=Sudo%20is%20a%20command%2Dline,run%20under%20their%20regular%20accounts>.

Stack Exchange. "Why do I have to use sudo for almost everything?" retrieved July 2, 2024, from <https://unix.stackexchange.com/questions/237221/why-do-i-have-to-use-sudo-for-almost-everything>.

Opensource.com. "How to use sudo on Linux," retrieved July 2, 2024, from <https://opensource.com/article/22/5/use-sudo-linux>.

OpenAI. (2024). *ChatGPT* (GPT-4). Retrieved from <https://chatgpt.com/>