Log Analyzer

Python Fundamentals

Edwin Lum
S8
CFC130124

TABLE OF CONTENTS

Introduction

In the ever-evolving landscape of cybersecurity, the ability to monitor and analyze system log files will enhance the cybersecurity posture in an organization. Log files contain a wealth of information that can offer insights into system activities, user behaviours, and potential security incidents. However, manually parsing through these logs can be time-consuming and inefficient, often leading to crucial details being overlooked.

To address this challenge, a Python-based log analysis tool, the log analyzer, was designed to streamline the process of extracting essential information from log files. This tool aims to provide cybersecurity professionals with a robust and efficient solution for monitoring system activities and identifying potential security breaches.

This report provides an overview of the functionality and usage of the log analyzer. By automating the process of log analysis, organizations can maintain the integrity and security of their IT infrastructure and mitigate the impact of data breaches and system compromises. The automation helps streamline security operations, increasing efficiency and allowing for more in-depth analysis of the log files by cybersecurity professionals.

Methodologies

Log Analyzer

This Log Analyzer was designed using Python. It helps cybersecurity professionals automate the analysis of log files, and review the log files for potential breaches. The following information are extracted from the log file:

1) Details of all commands used that are not run together with *sudo*
2) Details of all information regarding newly added users
3) Details of all information regarding deleted users
4) Details of all information regarding any password changes
5) Details of all information regarding any use of the *su* command
6) Details of all information regarding use of *sudo* and commands run with *sudo*
7) Details of any failed use of *sudo* and subsequent printing of an alert message

```
import re
import time

file = open ("auth.log", "r")
fdata = file.readlines()

#Part 1: Details of command usage (as all commands were used with sudo, they are only printed in Part 2E)
linecounter1 = -1
for line in fdata:
    line = line.strip()
    linecounter1 += 1
    if "COMMAND=" in line and "sudo:" not in line:
        if linecounter1+1 <= len(fdata):
            if "by (" in fdata[linecounter1+1]:
                exeuser1 = re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounter1+1])
            else:
                exeuser1 = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
        date1 = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
        time1 = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
        print("EVENT_TYPE: command_usage")
        if exeuser1.group() == "uid=1000":
            print("EXECUTION_USERNAME: kali")
        elif  exeuser1.group() == "uid=0":
            print("EXECUTION_USERNAME: root")
        else:
            print(f"EXECUTION_USERNAME: {exeuser1.group()}")
        if linecounter1+1 <= len(fdata):
            if "/usr/bin" in line and "by (" in fdata[linecounter1+1]:
                cmdtype1 = re.search("(?<=/usr/bin/)(.*)", line)
                print(f"COMMAND_EXECUTED: {cmdtype1.group()}")
            elif "/usr/sbin" in line and "by (" in fdata[linecounter1+1]:
                cmdtype1 = re.search("(?<=/usr/sbin/)(.*)", line)
                print(f"COMMAND_EXECUTED: {cmdtype1.group()}")
            else:
                if "/usr/bin" in line:
                    cmdfailed1 = re.search("(?<=/usr/bin/)(.*)", line)
                    print(f"COMMAND_EXECUTED: {cmdfailed1.group()}")
                elif "/usr/sbin" in line:
                    cmdfailed1 = re.search("(?<=/usr/sbin/)(.*)", line)
                    print(f"COMMAND_EXECUTED: {cmdfailed1.group()}")
                else:
                    cmdfailed1 = re.search("(?<=COMMAND=)(.*)", line)
                    print(f"COMMAND_EXECUTED: {cmdfailed1.group()}")
        print(f"TIMESTAMP: {date1.group()} {time1.group()}")
        if linecounter1+1 <= len(fdata):
            if "by (" not in fdata[linecounter1+1]:
                print ("COMMAND_USAGE_STATUS: FAILED")
        print(" ")
        time.sleep(3)
```

Figure 1: Part 1 of Log Analyzer

Lines 1 - 2
The *import re* and *import time* commands were first used to import the regular expressions (RegEx) and Time modules to be used later in the code.

Lines 4 - 5
The commands, *file = open("auth.log", "r")* and *fdata = file.readlines()* were used to open and read the auth.log file, parsing each line as a separate element in a list.

Part 1

Part 1 of the script aims to print out the details for all the commands that were run without the use of *sudo* in the auth.log file.

Lines 9 - 11
A *for* loop was used to iterate through every line in the auth.log file from the list. A variable, *linecounter1*, was used to count the number of lines during the iteration for each line.

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 12 - 20
An *if* statement was used on line 12 to ensure that the command printed is not run using *sudo* privileges.
A nested *if* statement was then used to check that the index of *linecounter1* does not extend beyond the length of the list of lines.

The next nested *if* statement used *linecounter1* to check the next line for the presence of a uid, which indicates that the command was successfully run and to extract the uid of the executing user using RegEx into the variable, *exeuser1*.
A successful check will allow the command, *re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounter1+1])* to use positive lookahead and positive lookbehind to extract the substring between "by (" and ")", which is the uid of the user executing the command.
If the check was unsuccessful, a similar RegEx command will save the username printed from the substring between "sudo: " and " :" on the same line, to the variable, *exeuser1*.

The date and time information were also saved into their respective variables, *date1* and *time1* using RegEx.
The characters within [ ] indicate the characters that the command matches in the substring. The numbers within { } indicate the range of occurrences the command matches for the characters in the substring.
For example, the command, *re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)* indicates that the *re.search()* command is looking for any substring containing between 1 to 4 digits followed by a hyphen and 1 to 2 digits, and then another hyphen and 1 to 2 digits in each line. The matched substring is saved into the variable, *date1*.
A similar RegEx command was used to save the time into the variable, *time1*.

The *print("EVENT TYPE: command_usage")* then prints the event type as command_usage to inform the script user that this line contains a command that was run.

Lines 21 - 26

The *if* statement is used to check for the uid in the variable, *exeuser1*. The command, *exeuser1.group()* returns the part of the string where there was a match, which is checked against the uid in the existing log file using the *if* statement.
The execution username will then be printed out according to the uid. If there is no matching uid or the command failed, either the uid or the username from the line will be printed out directly.

Lines 27 - 43

The *if* statement on line 27 checks that the index of *linecounter1* does not extend beyond the length of the list of lines.
An *if* statement uses the directories /usr/bin and /usr/sbin to check for the command as they contain the command binaries, and the next line is also checked for the uid to confirm that the command was succesfully executed.
RegEx lookahead is used to extract and save the successfully executed commands into the variable, *cmdtype1*.
The extracted substring will then be printed if the command was successfully executed.
If the command failed to execute, RegEx extracts the command and saves it into the variable, *cmdfailed1*.
The extracted substring will then be printed for the failed command.

Lines 44 - 49

The timestamp is printed using the *date1* and *time1* variables defined earlier.
An *if* statement is used to check for presence of uid on the next line, and a message showing that the command failed is printed if there is no uid on the next line.

The *print(" ")* command was used between each part of the script to print an empty line, allowing the script to be visually clearer when run. The *time.sleep(3)* command provides a 3 second pause between each printed segment, giving the script user time to read each segment of the information printed.

Part 2A

```
51  #Part 2A: Details of newly added users
52  linecounterA = -1
53  for line in fdata:
54      line = line.strip()
55      linecounterA += 1
56      if "/usr/sbin/useradd" in line:
57          if linecounterA+1 <= len(fdata):
58              if "by (" in fdata[linecounterA+1]:
59                  exeuserA = re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounterA+1])
60              else:
61                  exeuserA = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
62          newuserA = re.search("(?<=useradd\s)(.*)", line)
63          dateA = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
64          timeA = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
65          print("EVENT TYPE: new_user")
66          if exeuserA.group() == "uid=1000":
67              print("EXECUTION_USERNAME: kali")
68          elif  exeuserA.group() == "uid=0":
69              print("EXECUTION_USERNAME: root")
70          else:
71              print(f"EXECUTION_USERNAME: {exeuserA.group()}")
72          print(f"NEW_USERNAME: {newuserA.group()}")
73          print(f"TIMESTAMP: {dateA.group()} {timeA.group()}")
74          if linecounterA+1 <= len(fdata):
75              if "by (" not in fdata[linecounterA+1]:
76                  print ("NEW_USER_STATUS: FAILED")
77          print(" ")
78          time.sleep(3)
```

Figure 2: Part 2A of Log Analyzer

The codes used for Part 2A to 2D are very similar as they involve extracting similar details from different commands used.

Part 2A of the script is used to print out details of the new users that are added into the system.

Lines 52 - 55
Similar to Part 1, a *for* loop was used to iterate through every line in the auth.log file from the list. A different variable, *linecounterA*, was used to count the number of lines for Part 2A

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 56 - 65
An *if* statement was used to check for the *useradd* command on the line, followed by a nested *if* statement to ensure that the index of the list does not exceed the length of the list.
The nested *if* statement checks the next line for a successful use of the *useradd* command. RegEx lookahead and lookbehind was used on line 59, extracting the substring containing the uid on the next line, saving it into *exeuserA* variable.
If the command was unsuccessful, RegEx lookahead and lookbehind would be used to extract the substring containing the username on the same line, saving it into *exeuserA* variable.

RegEx lookahead was then used to extract the substring containing the new username, saving it into the variable, *newuserA*.

The date and time information were also saved into their respective variables, *dateA* and *timeA* using RegEx, similar to Part 1.

The event type of new_user was then printed on the next line of the output.

Lines 66 - 71
An *if* statement was used to check for the correct uid and print the executing username in the variable, *exeuserA*.

If there is no matching uid or the command failed, the variable execution username is taken from the same line and printed.

Lines 72 - 78
The next part of the code then prints the username of the newly added user, followed by the date and time.

The next *if* statements check if the command failed on the next line, while still on the *for* loop with the command on the current line.
The status of the failed event type is then printed if the command failed.

Lastly, *print(“ ”)* and *time.sleep(3)* commands were used to ensure the information are easily legible.

Part 2B

```
80  #Part 2B: Details of deleted users
81  linecounterB = -1
82  for line in fdata:
83      line = line.strip()
84      linecounterB += 1
85      if "/usr/sbin/userdel" in line:
86          if linecounterB+1 <= len(fdata):
87              if "by (" in fdata[linecounterB+1]:
88                  exeuserB = re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounterB+1])
89              else:
90                  exeuserB = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
91          deluserB = re.search("(?<=userdel\s)(.*)", line)
92          dateB = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
93          timeB = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
94          print("EVENT TYPE: delete_user")
95          if exeuserB.group() == "uid=1000":
96              print("EXECUTION_USERNAME: kali")
97          elif  exeuserB.group() == "uid=0":
98              print("EXECUTION_USERNAME: root")
99          else:
100             print(f"EXECUTION_USERNAME: {exeuserB.group()}")
101         print(f"DELETED_USERNAME: {deluserB.group()}")
102         print(f"TIMESTAMP: {dateB.group()} {timeB.group()}")
103         if linecounterB+1 <= len(fdata):
104             if "by (" not in fdata[linecounterB+1]:
105                 print ("DELETE_USER_STATUS: FAILED")
106         print (" ")
107         time.sleep(3)
```

Figure 3: Part 2B of Log Analyzer

Part 2B of the script is used to print out details of the deleted users that are removed from the system.

Lines 81 - 84

A *for* loop was used to iterate through every line in the auth.log file from the list. A variable, *linecounterB*, was used to count the number of lines for Part 2B.

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 85 - 94

An *if* statement was used to check for the *userdel* command on the line, followed by a nested *if* statement to ensure that the index of the list does not exceed the length of the list.
The nested *if* statement checks the next line for a successful use of the *userdel* command. RegEx lookahead and lookbehind was used to extract the substring containing the uid on the next line, saving it into *exeuserB* variable.
If the command was unsuccessful, RegEx lookahead and lookbehind would be used to extract the substring containing the username, saving it into *exeuserB* variable.

RegEx lookahead was then used to extract the substring containing the deleted username on the same line, saving it into the variable, *deluserB*.

The date and time information were also saved into their respective variables, *dateB* and *timeB* using RegEx.

Lines 95 - 100
An *if* statement was used to check for the correct uid and print the executing username in the variable, *exeuserB*.

If there is no matching uid or the command failed, the variable *exeuserB*, showing the execution username is taken from the same line and printed.

Lines 101 - 107
The next part of the code then prints the username of the deleted user, followed by the date and time.

The next *if* statements check if the command failed on the next line, while still on the *for* loop with the command on the current line.
The status of the failed event type is then printed if the command failed.

Lastly, *print(“ ”)* and *time.sleep(3)* commands were used to ensure the information are easily legible.

Part 2C

```
109  #Part 2C: Details of password change
110  linecounterC = -1
111  for line in fdata:
112      line = line.strip()
113      linecounterC += 1
114      if "/usr/bin/passwd" in line:
115          if linecounterC+1 <= len(fdata):
116              if "by (" in fdata[linecounterC+1]:
117                  exeuserC = re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounterC+1])
118              else:
119                  exeuserC = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
120          changepwC = re.search("(?<=passwd\s)(.*)", line)
121          dateC = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
122          timeC = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
123          print("EVENT TYPE: change_password")
124          if exeuserC.group() == "uid=1000":
125              print("EXECUTION_USERNAME: kali")
126          elif  exeuserC.group() == "uid=0":
127              print("EXECUTION_USERNAME: root")
128          else:
129              print(f"EXECUTION_USERNAME: {exeuserC.group()}")
130          print(f"USER_OF_PASSWORD: {changepwC.group()}")
131          print(f"TIMESTAMP: {dateC.group()} {timeC.group()}")
132          if linecounterC+3 <= len(fdata):
133              if "passwd[" in fdata[linecounterC+3] and "couldn't" in fdata[linecounterC+3]:
134                  print ("PASSWORD_CHANGE_STATUS: FAILED")
135          print (" ")
136          time.sleep(3)
```

Figure 4: Part 2C of Log Analyzer

Part 2C of the script is used to print out details when the command for password change is used.

Lines 110 - 113
A *for* loop was used to iterate through every line in the auth.log file from the list. A variable, *linecounterC*, was used to count the number of lines for Part 2C.

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 114 - 123
An *if* statement was used to check for the *passwd* command on the line, followed by a nested *if* statement to ensure that the index of the list does not exceed the length of the list.
The nested *if* statement checks the next line for a successful use of the *passwd* command. RegEx lookahead and lookbehind was used to extract the substring containing the uid on the next line, saving it into *exeuserC* variable.
If the command was unsuccessful, RegEx lookahead and lookbehind would be used to extract the substring containing the username, saving it into *exeuserC* variable.

RegEx lookahead was then used to extract the substring containing the deleted username on the same line, saving it into the variable, *changepwC*.

The date and time information were also saved into their respective variables, *dateC* and *timeC* using RegEx.

Lines 124 - 129
An *if* statement was used to check for the correct uid and print the executing username in the variable, *exeuserC*.

If there is no matching uid or the command failed, the variable *exeuserC*, showing the execution username is taken from the same line and printed.

Lines 130 - 136
The next part of the code then prints the username of the user whose password is being changed, followed by the date and time.

The next *if* statements check if the command failed on the next line, while still on the *for* loop with the command on the current line.
The status of the failed event type is then printed if the command failed.

Lastly, *print(“ ”)* and *time.sleep(3)* commands were used to ensure the information are easily legible.

Part 2D

```
138  #Part 2D: Details of su command usage
139  linecounterD = -1
140  for line in fdata:
141      line = line.strip()
142      linecounterD += 1
143      if "/usr/bin/su" in line:
144          if linecounterD+1 <= len(fdata):
145              if "by (" in fdata[linecounterD+1]:
146                  exeuserD = re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounterD+1])
147              else:
148                  exeuserD = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
149          dateD = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
150          timeD = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
151          print("EVENT TYPE: su_command")
152          if exeuserD.group() == "uid=1000":
153              print("EXECUTION_USERNAME: kali")
154          elif  exeuserD.group() == "uid=0":
155              print("EXECUTION_USERNAME: root")
156          else:
157              print(f"EXECUTION_USERNAME: {exeuserD.group()}")
158          print(f"TIMESTAMP: {dateD.group()} {timeD.group()}")
159          if "user NOT in sudoers" in line:
160              print ("SU_COMMAND_STATUS: FAILED")
161          print (" ")
162          time.sleep(3)
```

Figure 5: Part 2D of Log Analyzer

Part 2D of the script is used to print out details when the *su* command is used.

Lines 139 - 142

A *for* loop was used to iterate through every line in the auth.log file from the list. A variable, *linecounterD*, was used to count the number of lines for Part 2D.

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 143 - 151

An *if* statement was used to check for the *su* command on the line, followed by a nested *if* statement to ensure that the index of the list does not exceed the length of the list.
The nested *if* statement checks the next line for a successful use of the *su* command. RegEx lookahead and lookbehind was used to extract the substring containing the uid on the next line, saving it into *exeuserD* variable.
If the command was unsuccessful, RegEx lookahead and lookbehind would be used to extract the substring containing the username on the same line, saving it into *exeuserD* variable.

The date and time information were also saved into their respective variables, *dateD* and *timeD* using RegEx.

Lines 152 – 157

An *if* statement was used to check for the correct uid and print the executing username in the variable, *exeuserD*.

If there is no matching uid or the command failed, the variable *exeuserD*, showing the execution username is taken from the same line and printed.

Lines 158 – 162
The next part of the code then prints the date and time.

The *if* statement checks if the command failed due to executing user not being in the *sudo* list. The status of the failed event type is then printed if the command failed.

Lastly, *print(“ ”)* and *time.sleep(3)* commands were used to ensure the information are easily legible.

Part 2E

```
164  #Part 2E: Details of sudo usage
165  linecounterE = -1
166  for line in fdata:
167      line = line.strip()
168      linecounterE += 1
169      if "sudo:" and "COMMAND=" in line:
170          if linecounterE+1 <= len(fdata):
171              if "by (" in fdata[linecounterE+1]:
172                  exeuserE = re.search("(?<=by\s\()(.*?)(?=\))", fdata[linecounterE+1])
173              else:
174                  exeuserE = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
175          dateE = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
176          timeE = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
177          print("EVENT TYPE: sudo_usage")
178          if exeuserE.group() == "uid=1000":
179              print("EXECUTION_USERNAME: kali")
180          elif exeuserE.group() == "uid=0":
181              print("EXECUTION_USERNAME: root")
182          else:
183              print(f"EXECUTION_USERNAME: {exeuserE.group()}")
184          if linecounterE+1 <= len(fdata):
185              if "/usr/bin" in line and "by (" in fdata[linecounterE+1]:
186                  cmdtypeE = re.search("(?<=/usr/bin/)(.*)", line)
187                  print(f"COMMAND_EXECUTED: {cmdtypeE.group()}")
188              elif "/usr/sbin" in line and "by (" in fdata[linecounterE+1]:
189                  cmdtypeE = re.search("(?<=/usr/sbin/)(.*)", line)
190                  print(f"COMMAND_EXECUTED: {cmdtypeE.group()}")
191              else:
192                  if "/usr/bin" in line:
193                      cmdfailedE = re.search("(?<=/usr/bin/)(.*)", line)
194                  elif "/usr/sbin" in line:
195                      cmdfailedE = re.search("(?<=/usr/sbin/)(.*)", line)
196                  else:
197                      cmdfailedE = re.search("(?<=COMMAND=)(.*)", line)
198                  print(f"COMMAND_EXECUTED: {cmdfailedE.group()}")
199          print(f"TIMESTAMP: {dateE.group()} {timeE.group()}")
200          print (" ")
201          time.sleep(3)
```

Figure 6: Part 2E of Log Analyzer

Part 2E of the script is used to print out details of *sudo* usage along with the commands used.

Lines 165 - 168
A *for* loop was used to iterate through every line in the auth.log file from the list. A variable, *linecounterE*, was used to count the number of lines for Part 2E.

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 169 - 177
An *if* statement was used to check whether *sudo* and command usage are on the line, followed by a nested *if* statement to ensure that the index of the list does not exceed the length of the list.
The nested *if* statement checks the next line for a successful use of a command. RegEx lookahead and lookbehind was used to extract the substring containing the uid on the next line, saving it into *exeuserE* variable.
If the command was unsuccessful, RegEx lookahead and lookbehind would be used to extract the substring containing the username on the same line, saving it into *exeuserE* variable.

The date and time information were also saved into their respective variables, *dateE* and *timeE* using RegEx.

Lines 178 – 183
An *if* statement was used to check for the correct uid and print the executing username in the variable, *exeuserE*.

If there is no matching uid or the command failed, the variable *exeuserE*, showing the execution username is taken from the same line and printed.

Lines 184 - 198
The *if* statement on line 184 checks that the index of *linecounterE* does not extend beyond the length of the list of lines.
A nested *if* statement uses the directories /usr/bin and /usr/sbin to check for the command as they contain the command binaries, and the next line is also checked for the uid to confirm that the command was succesfully executed.
RegEx lookahead is used to extract and save the successfully executed commands into the variable, *cmdtypeE*.
The extracted substring will then be printed if the command was successfully executed.
If the command failed to execute, RegEx extracts the command and saves it into the variable, *cmdfailedE*.
The extracted substring will then be printed for the failed command.

Lines 199 - 201
The next part of the code then prints the date and time.

Any indication for failed *sudo* commands is not printed in Part 2E as it will be printed in Part 2F.

Lastly, *print(“ ”)* and *time.sleep(3)* commands were used to ensure the information are easily legible.

Part 2F

```
203  #Part 2F: Details of failed sudo usage
204  for line in fdata:
205      line = line.strip()
206      if "sudo:" and "COMMAND=" in line:
207          dateF = re.search("[0-9]{1,4}-[0-9]{1,2}-[0-9]{1,2}", line)
208          timeF = re.search("[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}", line)
209          exeuserF = re.search("(?<=sudo:\s)(.*?)(?=\s:)", line)
210          if "/usr/bin" and "user NOT in sudoers" in line:
211              cmdfailedF = re.search("(?<=COMMAND=/usr/bin/)(.*)", line)
212              print("ALERT!")
213              print("EVENT TYPE: sudo_failed")
214              print(f"EXECUTION_USERNAME: {exeuserF.group()}")
215              print(f"COMMAND_EXECUTED: {cmdfailedF.group()}")
216              print(f"TIMESTAMP: {dateF.group()} {timeF.group()}")
217              print (" ")
218              time.sleep(3)
219          elif "/usr/sbin" and "user NOT in sudoers" in line:
220              cmdfailedF = re.search("(?<=COMMAND=/usr/sbin/)(.*)", line)
221              print("ALERT!")
222              print("EVENT TYPE: sudo_failed")
223              print(f"EXECUTION_USERNAME: {exeuserF.group()}")
224              print(f"COMMAND_EXECUTED: {cmdfailedF.group()}")
225              print(f"TIMESTAMP: {dateF.group()} {timeF.group()}")
226              print (" ")
227              time.sleep(3)
228          elif "command not allowed" in line:
229              cmdfailedF = re.search("(?<=COMMAND=)(.*)", line)
230              print("ALERT!")
231              print("EVENT TYPE: sudo_failed")
232              print(f"EXECUTION_USERNAME: {exeuserF.group()}")
233              print(f"COMMAND_EXECUTED: {cmdfailedF.group()}")
234              print(f"TIMESTAMP: {dateF.group()} {timeF.group()}")
235              print (" ")
236              time.sleep(3)
```

Figure 7: Part 2F of Log Analyzer

Part 2F of the script is used to print out details of failed *sudo* usage along with the commands.

Lines 204 - 205
A *for* loop was used to iterate through every line in the auth.log file from the list.

The *line = line.strip()* command removes the unnecessary newlines and white spaces on each line of the file.

Lines 206 - 209
The *if* statement checks whether *sudo* and command are on the same line and then saves the date, time as well as the username into their respective variables, *dateF*, *timeF* and *exeuserF* using RegEx.

Lines 210 - 236
The *if* and *elif* statements use the directories /usr/bin and /usr/sbin to check for the respective commands and that the command failed due to the user not being a *sudo* user.
The following *elif* statement was used to check for the command not being allowed on the Linux command line using the string “command not allowed”.

Each of the statements will then use RegEx to extract the command and save it into the variable, *cmdfailedF*.
An alert message will then be printed, followed by the event type, “sudo_failed”.
The user that executed the command is printed using the *exeruserF.group()* command.
The command that failed is then printed using the *cmdfailedF.group()* command.
Next, the timestamp is printed using *dateF* and *timeF* variables.
Lastly, *print(“ ”)* and *time.sleep(3)* commands were used to ensure the information are easily legible.

Discussion

Log Analyzer

The Python-based log analyzer has been developed to streamline the process of extracting essential information from log files efficiently. The log analyzer also prioritized automation and ease of usage as cybersecurity professionals only need to run the script to extract all the information.

The tool reads log files line by line, extracting relevant information using regular expressions. Each log entry is analyzed to determine its type and then extract relevant details.

Each feature is implemented as a separate module, allowing for easier enhancement and debugging of the code in the future.

The tool generates structured output, in the form of text, containing details of the analyzed log entries. The output includes information such as event type, execution username, command executed, timestamp, and status (e.g., success or failure).

Part 1 of the script extracts details of commands executed on the system, providing insights into user activities and potential malicious behaviour. By analyzing command usage, cybersecurity professionals can identify suspicious activities, unauthorized software installations, or attempts to exploit system vulnerabilities.

Part 2A of the script identifies details of newly added users, enabling cybersecurity professionals to monitor user account creation activities. This feature is crucial for detecting unauthorized access attempts or potential insider threats.

Part 2B of the script detects details of deleted users, allowing cybersecurity professionals to monitor user account removal activities. Detecting deleted users helps in identifying attempts to cover tracks by attackers or remove evidence of unauthorized access.

Part 2C of the script tracks changes to user passwords, providing insights into user account management activities. Monitoring password changes helps in ensuring compliance with security policies and detecting unauthorized password modifications.

Part 2D of the script identifies users using the *su* command, which allows users to switch to the root user. Monitoring *su* command usage helps in detecting potential privilege escalation attempts or unauthorized access to sensitive system resources.

```
┌──(kali㉿kali)-[~/archive/Python/project]
└─$ python3 loganalyzer.py
EVENT TYPE: new_user
EXECUTION_USERNAME: kali
NEW_USERNAME: testuser
TIMESTAMP: 2024-04-10 06:34:21

EVENT TYPE: delete_user
EXECUTION_USERNAME: kali
DELETED_USERNAME: testuser
TIMESTAMP: 2024-04-10 06:39:07

EVENT TYPE: change_password
EXECUTION_USERNAME: kali
USER_OF_PASSWORD: testuser
TIMESTAMP: 2024-04-10 06:34:42
PASSWORD_CHANGE_STATUS: FAILED

EVENT TYPE: su_command
EXECUTION_USERNAME: kali
TIMESTAMP: 2024-04-10 06:33:42
```

Figure 8: Output of Log Analyzer from Part 2A to Part 2D

Part 2E of the script tracks users using the *sudo* command, which allows users to execute commands with root privileges. The log files keep an audit trail to help in monitoring *sudo* command usage for authorized administrative activities and potential misuse of privileges.

```
EVENT TYPE: sudo_usage
EXECUTION_USERNAME: kali
COMMAND_EXECUTED: apt install openssh-server
TIMESTAMP: 2024-04-10 06:29:24

EVENT TYPE: sudo_usage
EXECUTION_USERNAME: kali
COMMAND_EXECUTED: dpkg-reconfigure openssh-server
TIMESTAMP: 2024-04-10 06:30:09

EVENT TYPE: sudo_usage
EXECUTION_USERNAME: kali
COMMAND_EXECUTED: systemctl start ssh
TIMESTAMP: 2024-04-10 06:30:17
```

Figure 9: Showing part of the output of Log Analyzer from Part 2E

In addition, Part 2F identifies users who failed to use the *sudo* command successfully. Failed *sudo* command attempts could indicate misconfigurations, user privilege issues, or unauthorized access attempts that require further investigation.

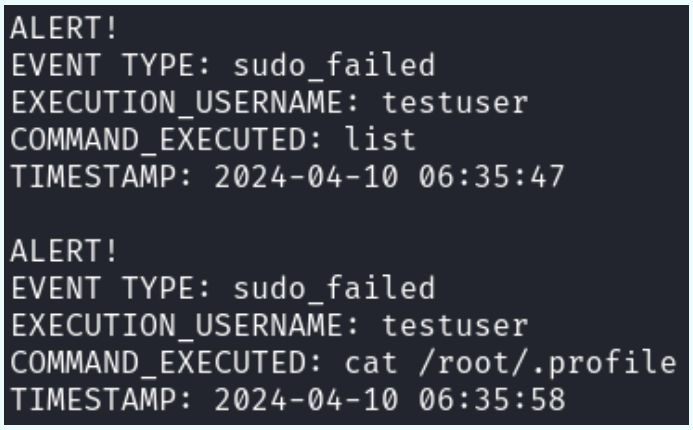

Figure 10: Output of Log Analyzer from Part 2F

Potential Uses of Log Analyzer

The log analyzer could be used during incident response to analyze log files for evidence of security breaches, unauthorized access, or suspicious activities. For example, detecting failed *sudo* command attempts may indicate brute-force attacks or unauthorized access attempts.

It can also be used in digital forensics to potentially help reconstruct events leading up to a security incident or data breach. Analyzing user activities, command usage, and password changes can provide insights into the attack vector and identify compromised accounts.

It can help to monitor compliance with security policies and regulations. For example, tracking user account creations and deletions helps ensure compliance with user access control policies, while monitoring password changes ensures adherence to password management guidelines.

Conclusion

The Python log analyzer aims to provide organizations with a simple solution for monitoring and analyzing system logs effectively. Running the straightforward and automated script enables cybersecurity professionals to extract critical details from log files, including command usage, user management activities, and privilege escalation attempts.

By automating the process of log analysis, this Python tool empowers organizations to enhance their cybersecurity posture and identify potential threats and vulnerabilities. The ability to track command usage allows administrators to monitor user activities and detect anomalous behaviour indicative of malicious intent. Similarly, the tool's capability to identify changes in user accounts, such as newly added users or password modifications, facilitates timely response to unauthorized access attempts.

Moreover, the detection of privilege escalation attempts, including the usage of *su* and *sudo* commands, provides insights into potential security breaches and unauthorized access to sensitive system resources. The identifying of failed attempts to use the *sudo* command also serves to diagnose misconfigurations in the system or detect suspicious activities that require further investigation.

In conclusion, the Python log analyzer represents a versatile tool for an organization seeking to bolster its cybersecurity defenses and maintain the integrity and security of its infrastructure. By leveraging automation, it creates a user-friendly solution for monitoring system logs and detecting of potential security breaches. With continuous updates and improvements to the tool, its useability and effectiveness can further improve, enabling the organization to stay resilient in the face of evolving cyber threats.

Recommendations

Log Analyzer

The current script is very lengthy due to the separate *for* loops and *if* statements used for each feature. Exploring the possibility of consolidating common functionalities into functions could make the code more organized, easier to read, and easier to update.

The use of RegEx allows complex and pre-defined patterns in strings to be matched. However, it makes the code less readable and as they can be difficult to understand. Hence, maintenance of the code could be challenging in the future. RegEx is also not suitable for complex and recursive data formats.

Implement proper error handling to handle exceptions for cases where RegEx patterns do not match any part of the log file.

Use of *with* statement to open the file would ensure that the file properly closes after the script has been completely run.

The output could be structured using dictionaries or objects for better readability and ease of further processing. This could make it easier to analyze and manipulate the extracted data.

Instead of hardcoding the log file path in the script, the Sys module could be used to allow users to specify the file path as a command-line argument or through a configuration file. This would give the tool more flexibility in different environments.

The log file in which the log analyzer is tested on does not cover all scenarios that can occur. For example, all the commands in the log file are used with *sudo* privileges, hence Part 1 of the tool does not print any output as they are all channeled to Part 2E and 2F. In addition, there are also no successful commands from testuser nor are there any successful password changes.

Testing the log analyzer with all possible scenarios would ensure that the log analyzer is more robust in handling different situations.

In addition, the tool could be improved by accounting for a wider range of uids or assessing the username without matching with the uids. The need to match the uid to the usernames increased the number of conditions required and complexity of the code. An alternative is to print the uid without the usernames.

An addtional check for the use of *su* command to switch to other users other than root, will help to check for possible horizontal privilege escalation.

References

1. Python Logo and Symbol, Meaning, History, PNG. 1000logos.net/python-logo/. Accessed 21 Apr. 2024.

2. “Regular Expression to Validate a Timestamp.” Stack Overflow, stackoverflow.com/questions/1057716/regular-expression-to-validate-a-timestamp. Accessed 21 Apr. 2024.

3. “Regex Match All Characters between Two Strings.” Stack Overflow, stackoverflow.com/questions/6109882/regex-match-all-characters-between-two-strings. Accessed 21 Apr. 2024.

4. W3Schools. “Python RegEx.” W3schools.com, 2019, www.w3schools.com/python/python_regex.asp. Accessed 21 Apr. 2024.

5. Wikipedia Contributors. “Filesystem Hierarchy Standard.” Wikipedia, Wikimedia Foundation, 26 Sept. 2019, en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard. Accessed 21 Apr. 2024.

6. “Differences between /Bin, /Sbin, /Usr/Bin, /Usr/Sbin, /Usr/Local/Bin, /Usr/Local/Sbin.” Ask Ubuntu, askubuntu.com/questions/308045/differences-between-bin-sbin-usr-bin-usr-sbin-usr-local-bin-usr-local. Accessed 21 Apr. 2024.

7. Cynet. “Understanding Privilege Escalation and 5 Common Attack Techniques.” Cynet, 2020, www.cynet.com/network-attacks/privilege-escalation/. Accessed 21 Apr. 2024.

8. “Coding Games and Programming Challenges to Code Better.” CodinGame, www.codingame.com/playgrounds/218/regular-expressions-basics/next-steps. Accessed 21 Apr. 2024.

9. “Is It Still Unsafe to Process Files without Using with in Python 3?” Stack Overflow, stackoverflow.com/questions/57992734/is-it-still-unsafe-to-process-files-without-using-with-in-python-3. Accessed 30 Apr. 2024.