Automation Assignment Report

First script:

# with open('system\_log.log', 'r') as file: logs = file.readlines():

* The purpose of this python script is contained within these sections of the code. The (with open('system\_log.log', 'r') as file) statement opens the file in read style ('r'). Using the with statement makes sure that the file is properly closed after its group of code finishes running, and even makes sure its closed properly even if an error occurs. The file.readlines() process reads all the lines from the file and returns them as a list, where each component corresponds to a line in the file. The lines read from the file are stored in the variable (logs), which is now a list holding each line of the log file as a separate string within the python code.

# import pandas as pd logs = pd.read\_csv('system\_log.csv'):

* The purpose of this python script is contained within these sections of the code. The line with import pandas as pd imports the pandas library, which is widely used for data control/running and analysis in Python. The ([pd.read](http://pd.read" \t "_blank)\_csv('system\_log.csv')) function reads the contents of the specified CSV file and loads it into a DataFrame called logs. As a DataFrame is a two-dimensional, size-mutable, potentially heterogeneous tabular data structure with rows and columns.

# sus\_logs= [log for log in logs if "failure" in log.lower() or 'unknown' in log.lower()] print(sus\_logs)

* The purpose of this part of the python script is that the code filters a list of log entries to extract those that contain the keywords "failure" or "unknown." This functionality is crucial for checking and analyzing logs, allowing users to quickly identify and address possible issues or security threats.

# suspicious\_logs = [log for log in logs if 'failed' in log.lower() or 'unauthorized' in log.lower()]:

* The purpose of the Python script code is to filter a list of log entries seemingly stored in a variable named (logs) to identify entries that contain specific keywords showing potential issues or security concerns. In this case, it looks for log entries that contain the words "failed" or "unauthorized". Here is a breakdown of the line of code: The code uses list understanding to create a new list called suspicious\_logs. This is a brief way to filter and transform data in Python. The condition checks each log entry (referred to as log): from log.lower() converts the log entry to lowercase, ensuring that the keyword search is case-insensitive.

# with open('summary\_report.txt', 'w') as f: f.write(f"Total suspicious logs found: {len(suspicious\_logs)}\n") for log in suspicious\_logs: f.write(log + '\n'):

* The purpose of this python script code is to is to create a summary report of mistrustful logs by writing the results to a text file named summary\_report.txt. Some of the main purposes include reporting suspicious or mysterious Logs, as the script sums up the findings of weird and suspicious log entries, which may indicate potential security issues or errors in the system. And to file output, as it outputs the total number of suspicious and mysterious logs and lists each one in the stated text file, making it easy to review and examine later.

The code of the implementation would be when the code runs a list of suspicious logs will print out on the console, as a line stating the total number of suspicious logs found.

Within our code that we created you can see from the results of the part of the code running:

**with** open('Linux\_2k.log', 'r') **as** file:

logs**=** file**.**readlines()

sus\_logs**=** [log **for** log **in** logs **if** "failure" **in** log**.**lower() **or** 'unknown' **in** log**.**lower()]

print(sus\_logs)

**with** open('summary\_report.txt', 'w') **as** f:

f**.**write(f"Total suspicious logs found: {len(sus\_logs)}\n")

**for** log **in** sus\_logs:

f**.**write(log **+** '\n')

The results of running the code:

['Jun 14 15:16:01 combo sshd(pam\_unix)[19939]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=218.188.2.4 \n', 'Jun 14 15:16:02 combo sshd(pam\_unix)[19937]: check pass; user unknown\n', 'Jun 14 15:16:02 combo sshd(pam\_unix)[19937]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=218.188.2.4 \n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20882]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-ip.hinet.net user=root\n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20884]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-ip.hinet.net user=root\n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20883]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-ip.hinet.net user=root\n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20885]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-ip.hinet.net user=root\n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20886]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-ip.hinet.net user=root\n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20892]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-ip.hinet.net user=root\n', 'Jun 15 02:04:59 combo sshd(pam\_unix)[20893]: authentication failure; logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=220-135-151-1.hinet-i

Our code shown above is implemented in the fallowing sequence and steps as the code runs:

Script begins by opening Linux\_2k.log in read mode, so it can read all the lines into a list called logs. It then processes these logs using a list understanding to create a new list, sus\_logs, that contains only those log entries that include the keywords "failure" or "unknown," regardless of case. This is achieved by changing each log entry to lowercase for the comparison. After filtering, the script prints the suspicious logs found to the console for fast review. It then opens a file named summary\_report.txt in write mode, where a summary of the findings is written. It first records the total number of suspicious logs identified and then repeats through the sus\_logs list to write each suspicious log entry into the summary report. As a result, a clear and organized output is produced that documents any potential issues within the original log file.

Second script:

# pip install psutil:

* The purpose of the command pip install psutil is to install the psutil library in Python. This library is a powerful instrument used for system and process management, allowing developers to re-claim information on system use (CPU, memory, disks, network, sensors) and running processes. As some of the main key features of psutil is system monitering, process management, and cross platform compatibility.

# cpu\_usage = psutil.cpu\_percent(interval=1)

# print(f"CPU Usage: {cpu\_usage}%"):

* The purpose of this script is to help you keep track of how much CPU resources are being used at any given moment. The application is useful for assessing the performance of applications, diagnosing performance problems, or monitoring the health of the system as a whole.

# memory\_info = psutil.virtual\_memory()

# print(f"Memory Usage: {memory\_info.percent}%"):

* In this part of the script, The purpose of it will be monitoring the memory usage of the system to detect performance issues, optimize applications, and ensure that the system is operating within acceptable limits, the purpose of which is to help diagnose performance issues, optimize applications, or ensure that the system is operating within acceptable limits.

# with open('performance\_log.txt', 'a') as f:

# f.write(f"CPU: {cpu\_usage}%, Memory: {memory\_info.percent}%\n"):

* This main purpose of this part of the python script is to log system performance metrics over time. By adding this information to a file, you can track changes in CPU and memory usage, which is helpful for Analyzing system performance trends, identifying potential bottlenecks or issues, and maintaining a historical record for debugging or performance review.

# cpu\_usage > 90: print("ALERT: High CPU usage detected!"):

* In this section of the Python script, the purpose of this snippet is primarily to serve as a simple monitoring tool, which alerts the user if there is a problem with CPU usage levels at a specific moment of time. The usage of the CPU by the system can be indicative of an overload or inefficiency of the system.

# smtplib from email.message import EmailMessage:

* As you can see, the purpose of this part of the python script is to prepare to send emails using Python in preparation for sending them. As a result of the imports of smtplib and EmailMessage, it allows the creation and sending of email messages programmatically, which can be very useful for the creation and sending of notifications, alerts, and other automated communication systems.

# subject= "something" body = "idk" def send\_alert(subject, body): msg = EmailMessage() msg.set\_content(body) msg['Subject'] = subject msg['From'] = 'dumby805@gmail.com' msg['To'] = 'eshaanvaranasi@gmail.com':

* The main purpose of this part of the python script is to define a function that prepares an email alert. It specifies the subject and body of the email, along with the sender and recipient addresses. The functionality of this feature can be integrated into a larger system in order to automate the process of sending notifications or alerts.

# smtplib.SMTP\_SSL('smtp.gmail.com', 465) as smtp: smtp.login('dumby805@gmail.com', 'ltym djkt xrnc yknw') smtp.send\_message(msg):

* The provided part of the script serves the purpose of setting up a secure connection to Gmail's SMTP server to send an email, logging in with the credentials provided by Gmail, and sending an email message to the specified address. The entire process is essential for automated email notifications and communication while at the same time ensuring the protection of sensitive data being transmitted. By using SMTP\_SSL, the script guarantees that the login credentials and email content are encrypted during transmission, which is fundamental for protecting sensitive info.

# cpu\_usage > .2: send\_alert('High CPU Usage Alert', f'CPU usage is {cpu\_usage}%'):

* The primary purpose is to check if the CPU usage exceeds a certain threshold (in this case, 20%) and, if it does, to send an alert notification. The main point is to serve as part of a monitoring system that alerts users or administrators when CPU usage exceeds a critical level. High CPU usage can specify overloaded processes and potential performance issues. This functionality is essential for maintaining system performance and quickly focusing on potential issues that could affect application steadiness or receptiveness.

Script Implemented:

**import** psutil

*# Get CPU usage*

cpu\_usage **=** psutil**.**cpu\_percent(interval**=**1)

print(f"CPU Usage: {cpu\_usage}%")

*# Get Memory usage*

memory\_info **=** psutil**.**virtual\_memory()

print(f"Memory Usage: {memory\_info**.**percent}%")

**with** open('performance\_log.txt', 'a') **as** f:

f**.**write(f"CPU: {cpu\_usage}%, Memory: {memory\_info**.**percent}%\n")

**if** cpu\_usage **>** 90:

print("ALERT: High CPU usage detected!")

**import** smtplib

**from** email.message **import** EmailMessage

subject**=** "something"

body **=** "idk"

**def** send\_alert(subject, body):

msg **=** EmailMessage()

msg**.**set\_content(body)

msg['Subject'] **=** subject

msg['From'] **=** 'dumby805@gmail.com'

msg['To'] **=** 'eshaanvaranasi@gmail.com'

**with** smtplib**.**SMTP\_SSL('smtp.gmail.com', 465) **as** smtp:

smtp**.**login('dumby805@gmail.com', 'ltym djkt xrnc yknw')

smtp**.**send\_message(msg)

**if** cpu\_usage **>** .2:

send\_alert('High CPU Usage Alert', f'CPU usage is {cpu\_usage}%')

The results of the code running:

CPU Usage: 4.8%

Memory Usage: 58.4%

Our code shown above is implemented in the fallowing sequence and steps as the code runs:

The script begins by importing the psutil module, which is used for saving information on system utilization like CPU and memory usage. First, it measures the CPU usage using psutil.cpu\_percent(interval=1), which waits for one second to compute the percentage of CPU usage and then prints the result. Next, it retrieves memory usage statistics through psutil.virtual\_memory() and prints the memory usage percentage.

The script then opens a file called performance\_log.txt in append mode and writes both the CPU and memory usage statistics into this file, logging the system’s performance data for future reference.

If the CPU usage exceeds 90%, the script prints an alert to the console: "ALERT: High CPU usage detected!", providing a quick notification of a possible performance issue.

After that, the script imports the smtplib module and the EmailMessage class to send an email alert. A function called send\_alert is defined, which builds an email message with a subject and body passed as parameters. The message is addressed to eshaanvaranasi@gmail.com from dumby805@gmail.com. The script then founds a secure connection to Gmail’s SMTP server using SMTP\_SSL on port 465, logs in to the sender’s email account using credentials, and sends the message.

Lastly, if the CPU usage exceeds 20% ( cpu\_usage > 0.2), the script beginnings the send\_alert function, informing the receiver that the CPU usage is high. This script serves as both a monitoring and alerting tool, keeping track of system performance and providing email notifications when necessary.

The third script:

# import subprocess:

* The purpose of this part of the Python script import subprocess is to import the subprocess module, which is part of Python's standard library. This module provides powerful facilities for spawning new processes, connecting to their input/output/error pipes, and obtaining their return codes. As this is highly and functionality is essential for scripting and automation tasks that require interaction with the operating system.

# run\_nmap(target): result = subprocess.run(['nmap', '-sV', target], capture\_output=True, text=True) print(result.stdout):

* Basically, the function facilitates network scanning using nmap, allowing users to check a target system's status quickly. It can also be used as part of a larger script to automate network discovery and security assessments. Essentially, the script defines a function that runs an nmap scan on a specified target and prints the results. Using this tool, network administrators and security professionals can discover hosts on a network, assess security vulnerabilities, and audit networks.

# run\_nmap('127.0.0.1') # Scan localhost:

* The main purpose of this command is to discover open ports and running services on a local machine. By scanning localhost, developers and system administrators can determine which services are active and their versions, test configurations and security settings without affecting external networks, and debug applications by ensuring the expected services are running. It provides users with insight into their own systems' services without affecting outside networks.

**import** subprocess

**def** run\_nmap(target):

result **=** subprocess**.**run(['nmap', '-sV', target], capture\_output**=True**, text**=True**)

print(result**.**stdout)

run\_nmap('127.0.0.1') *# Scan localhost*

*The results on running the code:*

Starting Nmap 7.95 ( https://nmap.org ) at 2024-10-28 19:27 Eastern Daylight Time

Nmap scan report for localhost (127.0.0.1)

Host is up (0.00037s latency).

Not shown: 993 closed tcp ports (reset)

PORT STATE SERVICE VERSION

135/tcp open msrpc Microsoft Windows RPC

445/tcp open microsoft-ds?

5357/tcp open http Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)

9000/tcp open zmtp ZeroMQ ZMTP 2.0

9001/tcp open zmtp ZeroMQ ZMTP 2.0

9002/tcp open zmtp ZeroMQ ZMTP 2.0

9003/tcp open zmtp ZeroMQ ZMTP 2.0

Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .

Nmap done: 1 IP address (1 host up) scanned in 13.31 seconds

Our code shown above is implemented in the fallowing sequence and steps as the code runs:

The script begins by importing the subprocess module, which enables the execution of external commands in Python. It defines a function named run\_nmap that takes a single parameter, target, representing the IP address or hostname to be scanned. In this function, the script uses subprocess.run() to perform the nmap command with the -sV option, which enables service version finding on the specified target. The capture\_output=True argument allows the script to capture the command's output, while text=True guarantees that the output is returned as a string rather than bytes. After performing the command, the script prints the captured output to the console. The function is then raised with '127.0.0.1' as the argument, which initiates a scan of the localhost. This implementation efficiently provides users with detailed information about the services running on their local machine, enabling quick assessments of network activity.

The fourth script:

# from scapy.all import

The main purpose of using Scapy is to enable users to perform various network-related tasks, such as Crafting Packets, Sending and Receiving Packets, Sniffing Network Traffic, **Network Scanning**, and Protocol Analysis. The part of the code from scapy.all import \* allows you to access the full functionality of the Scapy library, making it a multipurpose tool for network analysis and packet control.

# monitor\_packets(pkt): if pkt.haslayer(TCP) and pkt.haslayer(IP): print(f"Source IP: {pkt[IP].src}, Destination IP: {pkt[IP].dst}")

The main purpose of this function is to monitor and analyze TCP packets in real-time. It is possible to gain insights into network traffic patterns, identify communication between devices on a network, and troubleshoot network issues by observing packet flow by capturing and inspecting packets. This part of the code defines a function that monitors TCP packets and prints their source and destination IP addresses. In addition to network analysis, monitoring, and troubleshooting, this functionality makes it a valuable tool for network administrators and security professionals.

# (prn=monitor\_packets, count=10) # Capture 10 packets

The main point of this part of the Script is to specify parameters for a packet-sniffing operation using the Scapy library. Within Processing, the combination of these parameters is used to control the behavior of the packet-sniffing function, by specifying prn=monitor\_packets, the script processes each captured packet through the monitor\_packets function, which likely analyzes or logs details of the packet. Within Limiting, the count=10 parameter confirms that the sniffing operation does not run forever. Instead, it captures exactly 10 packets and then stops automatically. This is useful for testing or debugging purposes, allowing for a manageable amount of data to analyze. This is useful for analyzing a limited amount of network traffic efficiently.

**from** scapy.all **import** **\***

**def** monitor\_packets(pkt):

**if** pkt**.**haslayer(TCP) **and** pkt**.**haslayer(IP):

print(f"Source IP: {pkt[IP]**.**src}, Destination IP: {pkt[IP]**.**dst}")

sniff(prn**=**monitor\_packets, count**=**10) *# Capture 10 packets*

The results of running the code:

Source IP: 192.168.1.173, Destination IP: 13.107.138.10

Source IP: 192.168.1.173, Destination IP: 13.107.138.10

Source IP: 192.168.1.173, Destination IP: 13.107.138.10

Source IP: 192.168.1.173, Destination IP: 13.107.138.10

Source IP: 192.168.1.173, Destination IP: 13.107.138.10

Source IP: 192.168.1.173, Destination IP: 13.107.138.10

Out[13]:

<Sniffed: TCP:6 UDP:0 ICMP:0 Other:4>

Our code shown above is implemented in the fallowing sequence and steps as the code runs:

The script utilizes the Scapy library to monitor network traffic by defining a function called monitor\_packets that processes incoming packets. When a packet is captured, the function checks whether it holds both TCP and IP layers. If both layers are current, it obtains and prints the source and destination IP addresses of the packet. The script then calls the sniff function to begin capturing network packets, specifying that the monitor\_packets function should be performed for each captured packet. The count=10 parameter limits the capture to a total of ten packets. As the script runs, it offers real-time feedback on the network activity by displaying the relevant IP address information for the specified number of packets, effectively allowing monitoring of TCP communications on the network.