Automated Log Monitoring

Solution for Turn a New Leaf

Table of Contents

1. Executive Summary

As the Access Log Analyst at Turn a New Leaf, this report outlines the implemented solution to monitor and analyze system logs for detecting unusual network activity. The organization's network includes both Linux and Windows systems, with Linux machines acting as web servers. Every Thursday, members log into the system to update their employment status and submit job search activities. This regular login process and network activity require close monitoring for any unusual or suspicious behavior, particularly failed login attempts and HTTP errors (404, 401, and 500), which can signal security threats or system vulnerabilities (ICDSoft, n.d.).

The solution uses two primary tools, Bash and Python, to automate log monitoring and alert generation. The Bash script extracts logs from the Linux server, specifically monitoring access and error logs. The Python script then analyzes these logs for significant HTTP error patterns such as 404 (not found), 401 (unauthorized), and 500 (internal server error), as well as failed login attempts (MITRE Corporation, n.d.). The system continuously tracks these events, and when thresholds for error occurrences or failed logins are exceeded, an automated email alert is triggered to inform the relevant stakeholders. These alerts contain key details such as the IP addresses involved, timestamps, and the total number of errors detected within the specified time window.

In addition to alerting, the system stores the processed data in report files for further analysis, allowing for a historical view of errors and login attempts over time. This facilitates a more in-depth analysis of recurring issues and patterns in network activity. The automated workflow runs on a regular schedule using cron jobs to execute the Bash and as well as the Python scripts as a scheduled task on regular interval, ensuring that monitoring is performed continuously without manual intervention. This solution ensures that the network remains secure and that any suspicious activities are detected promptly.

The monitoring solution runs at specified intervals, capturing new logs and analyzing them in near real-time. Weekly reports summarizing system performance and any detected security issues are also generated and shared. The iteration frequency of the monitoring process can be adjusted based on system needs, such as increasing the monitoring frequency during high-traffic periods.

Future iterations of this solution could incorporate additional layers of analysis, including integrating machine learning models for anomaly detection and expanding monitoring coverage to other potential Indicators of Compromise (IoCs). These improvements will enhance the organization's ability to detect and respond to security threats in a timely and efficient manner.

2. Solution Overview

The solution involves two major components: automated log extraction and log analysis. It monitors for three key types of errors:

- **404 (Not Found)** – This error occurs when a requested resource cannot be located on the server.
- **401 (Unauthorized)** – This error indicates failed login attempts or unauthorized access attempts.
- **500 (Internal Server Error)** – This signals potential misconfigurations or issues with server resources. (ICDSoft, n.d.).

When the number of these errors exceeds predefined thresholds within a certain time window, an email alert is automatically generated and sent to the security team. Additionally, failed login attempts are also monitored, and alerts are triggered if the failed attempts exceed the threshold.

Bash Script for Log Extraction

The Bash script is responsible for extracting logs from the server. The logs are filtered to capture HTTP errors and failed login attempts. This script is scheduled to run regularly via a cron job.

Python Script for Log Analysis and Alerting

Once the logs are extracted, the Python script analyzes the data, identifying patterns of HTTP errors and failed logins. It uses regular expressions to search for specific error codes in the logs and calculates the number of occurrences within a specified time frame. If the count exceeds the predefined thresholds, an email alert is sent.

3. Programming Workflow

The workflow is fully automated, relying on cron jobs to handle the scheduling and execution of both Bash and while Python scripts. The Bash script runs every minute to ensure that new logs are captured promptly. The Python script is responsible for analyzing these logs and determining if an alert needs to be sent based on the error thresholds.

Example of the Bash Script

```
#!/bin/bash
# Define the paths for log file and output file
LOG_FILE="/var/log/apache2/access.log"
OUTPUT_FILE="/media/sf_test/Linux/linux_log/extracted_access_log.txt"
# Check if OUTPUT_FILE exists; if not, create it
if [ ! -f "$OUTPUT_FILE" ]; then
    touch "$OUTPUT_FILE"
```

```
fi
# Get the last processed line number
LAST_LINE=$(wc -l < "$OUTPUT_FILE")
# Extract new lines from the log file and use tail to get new lines after the last processed line
tail -n +$((LAST_LINE + 1)) "$LOG_FILE" >> "$OUTPUT_FILE"
```

Example of Cron Job for Bash Script

```
* * * * * /home/student/log_monitoring.sh
```

Example of the Python Script

```python
#filename: log_parsing.py
import os
import re
from datetime import datetime, timedelta
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

# Extracted log file path in shared
LOG_FILE_PATH = r"C:\Users\kakai\Documents\Canada Job Search\Cryptography & 
CyberSecurity\LHL Bootcamp\test\Linux\linux_log\extracted_access_log.txt"

# Path for the report file in the same folder as the log file
REPORT_FILE_PATH = os.path.join(os.path.dirname(LOG_FILE_PATH), 'report_log.txt')

THRESHOLD_404 = 5  # Number of 404 errors to trigger alarm
THRESHOLD_401 = 3  # Number of 401 errors to trigger alarm
THRESHOLD_500 = 5  # Number of 500 errors to trigger alarm
TIME_SPAN = timedelta(minutes=10)  # Time window for detecting multiple errors
ALERT_EMAIL = "sumit.giri199@gmail.com"  # Recipient email
SENDER_EMAIL = "sumitgiridrive5@gmail.com"  # Sender email
SENDER_PASSWORD = "****************"  # Sender email's app password

# Function to send an alert email
def send_email_alert(recent_errors):
    subject = "Multiple Error Attempts Detected"
    body = f"Warning: Detected errors in the last {TIME_SPAN}.\n\n"

    # Each type of error
    for error_type, details in recent_errors.items():
        body += f"{error_type}: {len(details)} occurrences\n"
        for error in details:
            body += f"IP: {error['ip']}, Time: {error['time']}\n"
        body += "\n"

    msg = MIMEMultipart()
    msg['From'] = SENDER_EMAIL
    msg['To'] = ALERT_EMAIL
    msg['Subject'] = subject
    msg.attach(MIMEText(body, 'plain'))

    # Setting up the SMTP server
    try:
        server = smtplib.SMTP('smtp.gmail.com', 587)  # SMTP server details
        server.starttls()
        server.login(SENDER_EMAIL, SENDER_PASSWORD)
        server.sendmail(SENDER_EMAIL, ALERT_EMAIL, msg.as_string())
        server.quit()
        log_to_file(f"Alert email sent to {ALERT_EMAIL}.")
    except Exception as e:
        log_to_file(f"Failed to send email: {e}")

# Function to log output to the report file
def log_to_file(message):
    with open(REPORT_FILE_PATH, 'a') as report_file:
        report_file.write(f"{datetime.now()}: {message}\n")

# Function to parse the log file
def parse_log():
    if not os.path.exists(LOG_FILE_PATH):
        log_to_file(f"Log file not found at {LOG_FILE_PATH}")
        return

    # Regular expressions to match errors in the log
    log_regex_404 = r'(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2}) .* 404 .*'
    log_regex_401 = r'(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2}) .* 401 .*'
    log_regex_500 = r'(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2}) .* 500 .*'

    time_format = "%d/%b/%Y:%H:%M:%S"

    # Dictionary to hold recent errors
    recent_errors = {
        "404 Errors": [],
        "401 Errors": [],
        "500 Errors": []
    }

    with open(LOG_FILE_PATH, 'r') as log_file:
        for line in log_file:
            current_time = datetime.now()

            # Check for 404 errors
            match_404 = re.search(log_regex_404, line)
            if match_404:
```

```python
                log_time_str = match_404.group(1)
                log_time = datetime.strptime(log_time_str, time_format)
                ip_address = line.split()[0]
                recent_errors["404 Errors"].append({'time': log_time, 'ip': ip_address})

            # Check for 401 errors
            match_401 = re.search(log_regex_401, line)
            if match_401:
                log_time_str = match_401.group(1)
                log_time = datetime.strptime(log_time_str, time_format)
                ip_address = line.split()[0]
                recent_errors["401 Errors"].append({'time': log_time, 'ip': ip_address})

            # Check for 500 errors
            match_500 = re.search(log_regex_500, line)
            if match_500:
                log_time_str = match_500.group(1)
                log_time = datetime.strptime(log_time_str, time_format)
                ip_address = line.split()[0]
                recent_errors["500 Errors"].append({'time': log_time, 'ip': ip_address})

    # Filtering recent errors based on time span
    for error_type in recent_errors:
        recent_errors[error_type] = [
            error for error in recent_errors[error_type]
            if current_time - error['time'] <= TIME_SPAN
        ]

    # Check for thresholds and send alerts if needed
    alert_triggered = False
    for error_type, details in recent_errors.items():
        if len(details) >= THRESHOLD_404 and error_type == "404 Errors":
            alert_triggered = True
        elif len(details) >= THRESHOLD_401 and error_type == "401 Errors":
            alert_triggered = True
        elif len(details) >= THRESHOLD_500 and error_type == "500 Errors":
            alert_triggered = True

    if alert_triggered:
        log_to_file(f"ALERT: Multiple errors detected within {TIME_SPAN}.")
        send_email_alert(recent_errors)
    else:
        log_to_file(f"Errors are under control: {recent_errors}")

if __name__ == "__main__":
    parse_log()
```

Automating Python Script as a Subprocess

```python
#filename: run_log_parsing.py
import sched
import time
import subprocess

# Path to the log parsing script
LOG_PARSING_SCRIPT = 'log_parsing.py'

def run_log_parsing(scheduler):
    # Schedule the next call first
    scheduler.enter(60, 1, run_log_parsing, (scheduler,))  # Call this function every 60
seconds

    # Run the log parsing script
    try:
        # Execute the log parsing script and wait for it to complete
        result = subprocess.run(['python', LOG_PARSING_SCRIPT], check=True)
        print(f"Log parsing script executed successfully: {result}")
    except subprocess.CalledProcessError as e:
        # Error Handling
        print(f"An error occurred while running the log parsing script: {e}")

    print("Log parsing process completed.")

# Create a scheduler instance
my_scheduler = sched.scheduler(time.time, time.sleep)

# Schedule the first execution of the log parsing script
my_scheduler.enter(0, 1, run_log_parsing, (my_scheduler,))  # Initial call without delay
my_scheduler.run()  # Start the scheduler
```

4. Expected Output

The expected output of the scripts includes log extraction from the web server and email alerts when error thresholds are exceeded. The errors tracked include HTTP 404, 401, and 500 responses, as well as failed login attempts. The output is saved in a report file for future analysis.

Code Output Snapshot

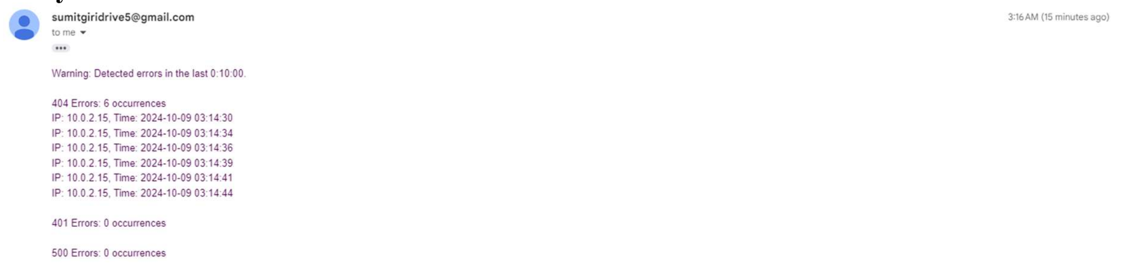
Below is an example of the console output after running the Python script:

```
2024-10-09 03:12:34.269089: Errors are under control: {'404 Errors': [], '401 Errors': [], '500 Errors': []}

2024-10-09 03:13:34.219216: Errors are under control: {'404 Errors': [], '401 Errors': [], '500 Errors': []}

2024-10-09 03:14:34.205773: Errors are under control: {'404 Errors': [], '401 Errors': [], '500 Errors': []}
```

```
2024-10-09 03:15:34.211179: ALERT: Multiple errors detected within 0:10:00.

2024-10-09 03:15:36.599630: Alert email sent to sumit.giri199@gmail.com.

2024-10-09 03:16:34.206512: ALERT: Multiple errors detected within 0:10:00.

2024-10-09 03:16:37.236568: Alert email sent to sumit.giri199@gmail.com.

2024-10-09 03:17:34.190079: ALERT: Multiple errors detected within 0:10:00.

2024-10-09 03:17:37.093492: Alert email sent to sumit.giri199@gmail.com.
```

Sample Email Alert

Subject: Multiple 404 Errors Detected
Body:



sumitgiridrive5@gmail.com
to me
3:16 AM (15 minutes ago)

```
Warning: Detected errors in the last 0:10:00.

404 Errors: 6 occurrences
IP: 10.0.2.15, Time: 2024-10-09 03:14:30
IP: 10.0.2.15, Time: 2024-10-09 03:14:34
IP: 10.0.2.15, Time: 2024-10-09 03:14:36
IP: 10.0.2.15, Time: 2024-10-09 03:14:39
IP: 10.0.2.15, Time: 2024-10-09 03:14:41
IP: 10.0.2.15, Time: 2024-10-09 03:14:44

401 Errors: 0 occurrences

500 Errors: 0 occurrences
```

Extracted Log File

```
10.0.2.15 - - [09/Oct/2024:03:14:30 -0400] "GET /1 HTTP/1.1" 404 488 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0"
10.0.2.15 - - [09/Oct/2024:03:14:34 -0400] "GET /2 HTTP/1.1" 404 487 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0"
10.0.2.15 - - [09/Oct/2024:03:14:36 -0400] "GET /3 HTTP/1.1" 404 487 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0"
10.0.2.15 - - [09/Oct/2024:03:14:39 -0400] "GET /4 HTTP/1.1" 404 487 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0"
10.0.2.15 - - [09/Oct/2024:03:14:41 -0400] "GET /5 HTTP/1.1" 404 487 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0"
10.0.2.15 - - [09/Oct/2024:03:14:44 -0400] "GET /6 HTTP/1.1" 404 487 "-" "Mozilla/5.0 (X11;
Ubuntu; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0"
```

---

5. Documentation Process

The monitoring process is thoroughly documented through log files generated by the scripts. Each script appends relevant events to the **extracted_access_log.txt** file, allowing for easy tracking of historical incidents. Additionally, the **report_log.txt** files record the Python output, ensuring comprehensive documentation of the monitoring activities. The email alerts sent to stakeholders also serve as immediate documentation of detected security issues, providing a clear and prompt notification of potential threats.

---

6. Potential Iterations

Future iterations of this project may involve:

1. **Integration of Machine Learning Models:** To improve anomaly detection, machine learning could be applied to historical data to identify patterns of normal behavior and flag deviations more effectively.
2. **Broader Coverage of Logs:** Expanding monitoring to cover additional log types from other services (e.g., FTP, DNS) could provide a more comprehensive security posture.
3. **Enhanced Alerting Mechanisms:** Implementing multi-tiered alerting could ensure that critical incidents are escalated appropriately based on severity.
4. **Dashboard Visualization:** Developing a web-based dashboard to visualize real-time log analysis results and alert statuses could facilitate quicker response times.

---

7. Conclusion

This automated monitoring solution successfully detects unusual network activity, specifically targeting HTTP errors and failed login attempts. The solution operates efficiently through scheduled Bash and Python scripts and offers prompt email alerts when defined thresholds are met. The documentation and reporting mechanisms ensure transparency in security monitoring efforts, and the proposed iterations can further strengthen the security posture of Turn a New Leaf.

---

8. References

1. ICDSoft. (n.d.). Apache error codes: 401, 403, 404, 412, 500. https://www.icdsoft.com/en/kb/view/1251_apache_error_codes_401_403_404_412_500
2. MITRE Corporation. (n.d.). T1110 - brute force. https://attack.mitre.org/techniques/T1110/