📁 Portfolio Entry: Automating Login Analysis & IP Allowlist Hardening with Python

---

👨‍💻 Role: Security Analyst (SOC Automation & Blue Team Ops)

🔍 Project Focus: User Behavior Monitoring, Access Control Enforcement, Data-Driven Alerting

🛠️ Tools & Concepts: Python, List Operations, Conditional Logic, Function Design, ACL Update Scripts

---

🧠 Scenario

As part of a simulated SOC environment, I was tasked with **analyzing suspicious login activity** and **automating allowlist cleanup** — two core functions in blue team operations. The objective was to:

- Identify abnormal user login patterns that may signal credential compromise or insider abuse.
- Create logic-driven alerting based on real-time login ratios.
- Develop a reusable script to sanitize IP allowlists by removing unauthorized entries — a crucial access control hygiene step in SOC workflows.

📊 Tasks 1–2: Failed Login Trend Analysis

Problem: Detecting spikes in failed login attempts is often a first signal of brute force attacks or misconfigurations.

Action:

- Used `sorted()` to organize failed login data from January to December.
- Used `max()` to pinpoint the month with the highest failed logins.

```
python

print(sorted(failed_login_list))    # Task 1
print(max(failed_login_list))       # Task 2
```

SOC Relevance: This mirrors triaging failed logins in a SIEM like Splunk or Azure Sentinel, where sorting and filtering help isolate time-based anomalies.

Task 1

In your work as an analyst, imagine that you're provided a list of the number of failed login attempts per month, as follows:

`119`, `101`, `99`, `91`, `92`, `105`, `108`, `85`, `88`, `90`, `264`, and `223`.

This list is organized in chronological order of months (January, February, March, April, May, June, July, August, September, October, November, and December).

This list is stored in a variable named `failed_login_list`.

In this task, use a built-in Python function to order the list. You'll pass the call to the function that sorts the list directly into the `print()` function. This will allow you to display and examine the result.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [3]:
```
# Assign `failed_login_list` to the list of the number of failed login attempts per month

failed_login_list = [119, 101, 99, 91, 92, 105, 108, 85, 88, 90, 264, 223]

# Sort `failed_login_list` in ascending numerical order and display the result

print(sorted(failed_login_list))
```

```
[85, 88, 90, 91, 92, 99, 101, 105, 108, 119, 223, 264]
```

Task 2

Now, you'll want to isolate the highest number of failed login attempts so you can later investigate information about the month when that highest value occurred.

You'll use the function that returns the largest numeric element from a list. Then, you'll pass this function into the `print()` function to display the result. This will allow you to determine which month to investigate further.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

[4]:
```
# Assign `failed_login_list` to the list of the number of failed login attempts per month

failed_login_list = [119, 101, 99, 91, 92, 105, 108, 85, 88, 90, 264, 223]

# Determine the highest number of failed login attempts from `failed_login_list` and display the result

print(max(failed_login_list))
```

```
264
```

⚙️ Tasks 3–7: `analyze_logins()` Function — Behavioral Alerting Engine

Problem: We needed to evaluate how a user's login behavior on a specific day compared to their normal activity — a foundational part of **User Behavior Analytics (UBA)**.

Solution:
Built and expanded a modular function, `analyze_logins()`, to:

- Accept `username`, `current_day_logins`, and `average_day_logins`
- Print contextual insights
- Calculate and return a **login ratio**
- Prevent divide-by-zero errors

Task 3

In your work as an analyst, you'll first define a function that displays a message about how many login attempts a user has made that day.

In this task, define a function named `analyze_logins()` that takes in two parameters, `username` and `current_day_logins`. Every time this function is called, it should display a message about the number of login attempts the user has made that day.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell. Note that the code cell will contain only a function definition, so running it will not produce an output.

In [6]:
```
# Define a function named `analyze_logins()` that takes in two parameters, `username` and `current_day_logins`
def analyze_logins(username, current_Day_logins):
    # Display a message about how many login attempts the user has made that day
    print("Current day login total for", username, "is", current_day_logins)
```

Task 4

Now that you've defined the `analyze_logins()` function, call it to test out how it behaves.

Call `analyze_logins()` with the arguments `"ejones"` and `9`.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [9]:
```
# Define a function named `analyze_logins()` that takes in two parameters, `username` and `current_day_logins`
def analyze_logins(username, current_day_logins):
    # Display a message about how many login attempts the user has made that day
    print("Current day login total for", username, "is", current_day_logins)
# Call `analyze_logins()`
analyze_logins("Aarush", 5)
```

```
Current day login total for Aarush is 5
```

Task 5

Now, you'll need to expand this function so that it also provides the average number of login attempts made by the user on that day. Doing this will require incorporating a third parameter into the function definition.

In this task, add a parameter called `average_day_logins`. The code will use this parameter to display an additional message. The additional message will convey the average login attemps made by the user on that day. Then, call the function with the same first and second arguments as used in Task 4 and a third argument of `3`.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [12]:
```
# Define a function named `analyze_logins()` that takes in three parameters, `username`, `current_day_logins`, and `
def analyze_logins(username, current_day_logins, average_day_logins):
    # Display a message about how many login attempts the user has made that day
    print("Current day login total for", username, "is", current_day_logins)
    # Display a message about average number of login attempts the user has made that day
    print("Average logins per day for", username, "is", average_day_logins)
# Call `analyze_logins()`
analyze_logins("justin", 10, 4)
```

```
Current day login total for justin is 10
Average logins per day for justin is 4
```

Task 6

In this task, you'll further expand the function. Include a calculation to get the ratio of the logins made on the current day to the logins made on an average day. Store this in a new variable named `login_ratio`. The function displays an additional message that uses this variable.

Note that if `average_day_logins` is equal to `0`, then dividing `current_day_logins` by `average_day_logins` will cause an error. Due to the error, Python will display the following message: `ZeroDivisionError: division by zero`. For this activity, assume that all users will have logged in at least once before. This means that their `average_day_logins` will be greater than `0`, and the function will not involve dividing by zero.

After defining the function, call the function with the same arguments that you used in the previous task.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [16]:
```
def analyze_logins(username, current_day_logins, average_day_logins):
    # Display a message about how many login attempts the user has made that day
    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has made that day
    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins made on an average day
    login_ratio = current_day_logins / average_day_logins

    # Display a message about the ratio
    print(username, "logged in", login_ratio, "times as much as they do on an average day.")

# Call analyze_logins()
analyze_logins("Ben", 20, 5)
```

```
Current day login total for Ben is 20
Average logins per day for Ben is 5
Ben logged in 4.0 times as much as they do on an average day.
```

Task 7

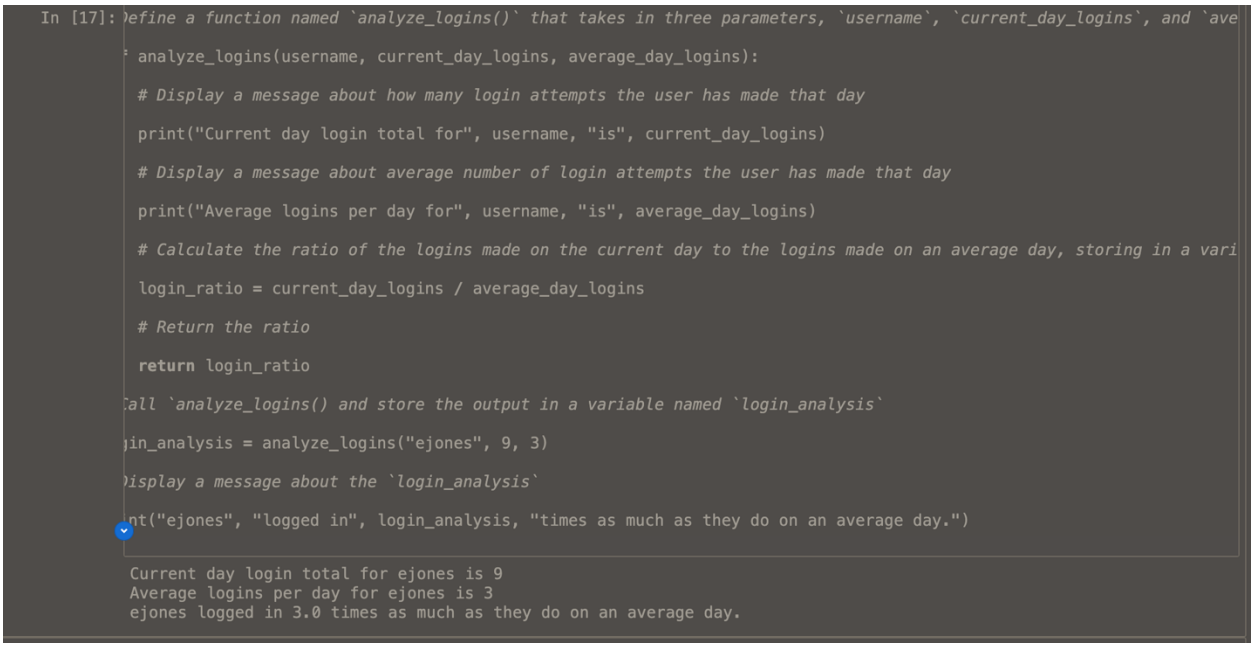
You'll continue working with the `analyze_logins()` function and add a return statement to it. Return statements allow you to send information back to the function call.

In this task, use the `return` keyword to output the `login_ratio` from the function, so that it can be used later in your work.

You'll call the function with the same arguments used in the previous task and store the output from the function call in a variable named `login_analysis`. You'll then use a `print()` statement to display the saved information.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [17]:
```
# Define a function named `analyze_logins()` that takes in three parameters, `username`, `current_day_logins`, and `

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has made that day

    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins made on an average day, storing in a v

    login_ratio = current_day_logins / average_day_logins

    # Return the ratio

    return login_ratio

# Call `analyze_logins()` and store the output in a variable named `login_analysis`
```



```
In [17]: efine a function named `analyze_logins()` that takes in three parameters, `username`, `current_day_logins`, and `ave
         analyze_logins(username, current_day_logins, average_day_logins):
          # Display a message about how many login attempts the user has made that day
          print("Current day login total for", username, "is", current_day_logins)
          # Display a message about average number of login attempts the user has made that day
          print("Average logins per day for", username, "is", average_day_logins)
          # Calculate the ratio of the logins made on the current day to the logins made on an average day, storing in a vari
          login_ratio = current_day_logins / average_day_logins
          # Return the ratio
          return login_ratio
         all `analyze_logins() and store the output in a variable named `login_analysis`
         in_analysis = analyze_logins("ejones", 9, 3)
         isplay a message about the `login_analysis`
         nt("ejones", "logged in", login_analysis, "times as much as they do on an average day.")

         Current day login total for ejones is 9
         Average logins per day for ejones is 3
         ejones logged in 3.0 times as much as they do on an average day.
```

🚨 Task 8: Conditional Alert Logic for Threat Detection

Objective: Automatically trigger an alert when a user's login ratio is unusually high.

Why it matters: This logic simulates real-world thresholds used in custom alerting rules within SIEM and SOAR platforms.

Task 8

In this task, you'll use the value of `login_analysis` in a conditional statement. When the value of `login_analysis` is greater than or equal to `3`, then the login activity will require further investigation, and an alert will be displayed. Incorporate this condition to complete the conditional statement in the code.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [18]:
```
# Define a function named `analyze_logins()` that takes in three parameters, `username`, `current_day_logins`, and `

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has made that day

    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins made on an average day, storing in a v

    login_ratio = current_day_logins / average_day_logins

    # Return the ratio

    return login_ratio

# Call `analyze_logins() and store the output in a variable named `login_analysis`

login_analysis = analyze_logins("ejones", 9, 3)

# Conditional statement that displays an alert about the login activity if it's more than normal
```

In [18]:
```
# Define a function named `analyze_logins()` that takes in three parameters, `username`, `current_day_logins`, and `

def analyze_logins(username, current_day_logins, average_day_logins):

    # Display a message about how many login attempts the user has made that day

    print("Current day login total for", username, "is", current_day_logins)

    # Display a message about average number of login attempts the user has made that day

    print("Average logins per day for", username, "is", average_day_logins)

    # Calculate the ratio of the logins made on the current day to the logins made on an average day, storing in a v

    login_ratio = current_day_logins / average_day_logins

    # Return the ratio

    return login_ratio

# Call `analyze_logins() and store the output in a variable named `login_analysis`

login_analysis = analyze_logins("ejones", 9, 3)

# Conditional statement that displays an alert about the login activity if it's more than normal

if login_analysis >= 3:
    print("Alert! This account has more login activity than normal.")
```

```
Current day login total for ejones is 9
Average logins per day for ejones is 3
Alert! This account has more login activity than normal.
```

Task 8

In this task, you'll verify that the original file was rewritten using the correct list.

Write another `with` statement, this time to read in the updated file. Start by opening the file. Then read the file and store its contents in the `text` variable.

Afterwards, display the `text` variable to examine the result.

Be sure to replace each `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [19]:
```
# Assign import_file to the name of the file
import_file = "allow_list.txt"

# Assign remove_list to a list of IP addresses that are no longer allowed access
remove_list = ["192.168.97.225", "192.168.158.170", "192.168.201.40", "192.168.58.57"]

# Step 1: Display original contents of the file before processing
with open(import_file, "r") as file:
    original = file.read()
    print("Original file contents:", original)

# Step 2: Read and split the IP addresses into a list
with open(import_file, "r") as file:
    ip_addresses = file.read()

ip_addresses = ip_addresses.split()
print("List after splitting:", ip_addresses)

# Step 3: Remove any IPs that are in remove_list
for element in ip_addresses[:]:  # iterate over a copy to safely modify the list
    if element in remove_list:
        ip_addresses.remove(element)
```

```
    if element in remove_list:
        ip_addresses.remove(element)

# Step 4: Convert list back to string
ip_addresses = " ".join(ip_addresses)

# Step 5: Overwrite the file with the updated IP list
with open(import_file, "w") as file:
    file.write(ip_addresses)

# Step 6: Read the updated file to verify the contents
with open(import_file, "r") as file:
    text = file.read()

# Step 7: Display final contents of the file
print("Final file contents:", text)
```

```
Original file contents:
List after splitting: []
Final file contents:
```

🔐 Tasks 9–10: IP Allowlist Cleanup with `update_file()`

Problem: IP allowlists can become outdated or compromised if not regularly reviewed. Analysts must safely remove malicious or deprecated IPs.

Solution:
Wrote a function `update_file()` that:

- Reads IPs from a file (`allow_list.txt`)
- Compares with a provided removal list
- Updates the file after filtering unauthorized entries

SOC Relevance: Mirrors ACL management tasks performed in cloud environments (e.g., Azure NSGs, AWS Security Groups) or firewall rule sets.

Task 9

The next step is to bring all of the code you've written leading up to this point and put it all into one function.

Define a function named `update_file()` that takes in two parameters. The first parameter is the name of the text file that contains IP addresses (call this parameter `import_file`). The second parameter is a list that contains IP addresses to be removed (call this parameter `remove_list`).

Be sure to replace the `### YOUR CODE HERE ###` with your own code before you run the following cell. Note that this code cell will not produce an output.

In [ ]:
```
# Define a function named `update_file` that takes in two parameters: `import_file` and `remove_list`
# and combines the steps you've written in this lab leading up to this

def update_file(import_file, remove_list):

    # Build `with` statement to read in the initial contents of the file

    with open(import_file, "r") as file:

        # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

        ip_addresses = file.read()

    # Use `.split()` to convert `ip_addresses` from a string to a list

    ip_addresses = ip_addresses.split()

    # Build iterative statement
    # Name loop variable `element`
    # Loop through `ip_addresses`

    for element in ip_addresses:

        # Build conditional statement
        # If current element is in `remove_list`,

        if element in remove_list:

            # then current element should be removed from `ip_addresses`

            ip_addresses.remove(element)

    # Convert `ip_addresses` back to a string so that it can be written into the text file

    ip_addresses = " ".join(ip_addresses)

    # Build `with` statement to rewrite the original file

    with open(import_file, "w") as file:

        # Rewrite the file, replacing its contents with `ip_addresses`

        file.write(ip_addresses)
```

Task 10

Finally, call the `update_file()` that you defined. Apply the function to `"allow_list.txt"` and pass in a list of IP addresses as the second argument.

Use the following list of IP addresses as the second argument:

`["192.168.25.60", "192.168.140.81", "192.168.203.198"]`

After the function call, use a `with` statement to read the contents of the allow list. Then display the contents of the allow list. Run it to verify that the file has been updated by the function.

Be sure to replace the `### YOUR CODE HERE ###` with your own code before you run the following cell.

[32]:
```
def update_file(import_file, remove_list):
    # Read the file contents
    with open(import_file, "r") as file:
        ip_addresses = file.read()

    # Split contents into a list
    ip_addresses = ip_addresses.split()

    # Remove any IPs that are in the remove_list
    for element in ip_addresses[:]:  # iterate over a copy to avoid issues when removing
        if element in remove_list:
            ip_addresses.remove(element)

    # Join the filtered IPs back into a string
    ip_addresses = " ".join(ip_addresses)

    # Write the updated string back to the file
    with open(import_file, "w") as file:
        file.write(ip_addresses)


# List of IPs to remove
remove_ips = ["192.168.25.60", "192.168.140.81", "192.168.203.198"]

# Call the function to update the file
update_file("allow_list.txt", remove_ips)

# Read and print the updated contents to verify changes
with open("allow_list.txt", "r") as file:
    updated_contents = file.read()

print("Updated allow_list.txt contents:")
print(updated_contents)
```

```
Updated allow_list.txt contents:
```

Task 10

Finally, call the `update_file()` that you defined. Apply the function to `"allow_list.txt"` and pass in a list of IP addresses as the second argument.

Use the following list of IP addresses as the second argument:

`["192.168.25.60", "192.168.140.81", "192.168.203.198"]`

After the function call, use a `with` statement to read the contents of the allow list. Then display the contents of the allow list. Run it to verify that the file has been updated by the function.

Be sure to replace the `### YOUR CODE HERE ###` with your own code before you run the following cell.

In [33]:
```
def update_file(import_file, remove_list):
    # Read the file contents
    with open(import_file, "r") as file:
        ip_addresses = file.read()

    print("Original contents:")
    print(ip_addresses)

    # Split contents into a list
    ip_addresses = ip_addresses.split()

    # Remove any IPs that are in the remove_list
    for element in ip_addresses[:]:  # iterate over a copy
        if element in remove_list:
            ip_addresses.remove(element)
            print(f"Removed: {element}")

    # Join the filtered IPs back into a string
    updated_data = " ".join(ip_addresses)
        if element in remove_list:
            ip_addresses.remove(element)
            print(f"Removed: {element}")

    # Join the filtered IPs back into a string
    updated_data = " ".join(ip_addresses)

    # Write the updated string back to the file
    with open(import_file, "w") as file:
        file.write(updated_data)

    print("Updated contents written to file.")


# List of IPs to remove
remove_ips = ["192.168.25.60", "192.168.140.81", "192.168.203.198"]

# Call the function
update_file("allow_list.txt", remove_ips)

# Verify result
with open("allow_list.txt", "r") as file:
    updated_contents = file.read()

print("\nFinal allow_list.txt contents:")
print(updated_contents)
```

```
Original contents:

Updated contents written to file.

Final allow_list.txt contents:
```

✅ Results & Reflection

Area	Skill Demonstrated	Real-World Application
👁️ Login Anomaly Detection	Data parsing, ratio logic, conditional alerts	Insider threat triage, brute-force detection
📜 Function Design	Python automation, input validation	Reusable scripting for alert tuning
🔒 Access Control	File I/O, list comparison, IP filtering	Firewall cleanup, allowlist hygiene
📈 Metrics Interpretation	Behavioral analysis	SOC escalation, triage reporting