🔐 Project Title: Detecting Suspicious Login Patterns with Python — A Simulated SOC Analyst Investigation

---

🧠 Executive Summary

This project simulates a real-world SOC analyst workflow — from ingesting login attempt data, conducting behavioral analysis, and calculating risk thresholds, to triggering conditional alerts that align with threat detection logic. Using only native Python, I built a prototype logic engine capable of identifying abnormal user activity patterns, showcasing my fluency in core scripting principles, analytic decision-making, and translating technical detection goals into executable code. This work lays a foundation for deeper SIEM correlation rules, MITRE ATT&CK mapping (e.g., T1078: Valid Accounts), and Tier 1–2 blue team operations.

📍 Problem Statement:

As a security analyst, you're often handed raw login data and asked to spot anomalies — fast. This project begins with a dataset representing **monthly failed login attempts**. The challenge? Make sense of these numbers using pure logic and Python to detect outliers, profile user behavior, and flag potentially malicious activity in a way that could later be ported into a SIEM like Splunk or Sentinel.

🛠️ Technical Workflow Breakdown:

🔢 Task 1–2: Establishing the Data Baseline

- Sorted raw monthly login attempt data to quickly identify normal vs. extreme values.
- Extracted the **maximum failed login count**, which serves as the first behavioral benchmark — important for thresholds used in account brute-force detection.

✅ Tools Used: `sorted()`, `max()`
🔄 Real-World Relevance: These are early-stage data enrichment actions akin to log normalization in SIEM pipelines.

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
```

🧩 Task 3–5: Designing a User Behavior Function

- Created a reusable function `analyze_logins(username, current_day_logins)` to automate daily reporting.
- Enhanced with parameters that introduce **user-specific behavior modeling**, providing insight into each user's login volume per day.
- Extended logic to include **average logins**, offering a comparative baseline.

🎯 Impact: Simulates what a SOC analyst might script as part of a scheduled job or automated rule to profile user activity patterns.

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 6–7: Calculating Login Anomaly Ratios

- Introduced a custom **login ratio metric** — `current_day_logins / average_logins` — as a behavioral indicator.
- Captured the return value of the function and stored it in `login_analysis`, enabling us to **track risk numerically**.

🔍 This metric acts as a primitive User Entity Behavior Analytics (UEBA) indicator, highlighting when a user's activity deviates too sharply from normal.

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
         f 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: Implementing Conditional Alert Logic

- Built logic to check:

  ```
  python

  if login_analysis >= 3:
      print("⚠️ ALERT: Unusual login activity. Further investigation
  required.")
  ```

- This mirrors a SIEM correlation rule that could trigger an alert in response to anomalous login behavior.

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.
```

🧠 Real-World Tie-In: This parallels MITRE technique **T1078 (Valid Accounts)** where account misuse often surfaces through abnormal login frequency.
Could also support detection logic for **T1110 (Brute Force)**.

🔄 Task 9–10: Boolean Logic & Status Modeling

- Experimented with Boolean expressions to simulate account lockouts or abnormal login attempt logic.
- Introduced a `login_status` variable to represent real-time authentication state, and captured its datatype — a nod to **type enforcement and state monitoring** common in enterprise-grade tools.

✅ SOC Readiness: Boolean flags like these often control trigger conditions in SIEM alerts or access control decisions.

Task 9

This code continues to check for the Boolean value of whether `max_logins` is less than or equal to `login_attempts`. In this task, reassign other values to `login_attempts`. For example, you might choose a value that is higher than the maximum number of attempts allowed. Observe how the output changes.

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

In [13]:
```
# Assign `max_logins` to the value 3

max_logins = 3

# Assign `login_attempts` to a specific value

login_attempts = 7

# Determine whether the current number of login attempts a user has made is less than or equal to the maximum number
# and display the resulting Boolean value

print(login_attempts <= max_logins)
```

```
False
```

Task 10

Finally, you can also assign a Boolean value of `True` or `False` to a variable.

In this task, you'll create a variable called `login_status`, which is a Boolean that represents whether a user is logged in. Assign `False` to this variable and store its data type in a variable called `login_status_type` and display it.

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

In [14]:
```
# Assign `login_status` to the Boolean value False

login_status = False

# Assign `login_status_type` to the data type of `login_status`

login_status_type = type(login_status)

# Display `login_status_type`

print(login_status_type)
```

```
<class 'bool'>
```

🔍 Key Takeaways & Skills Demonstrated:

Category	Evidence
Python Scripting	Function creation, parameter passing, conditionals, Boolean logic
Data Analysis	Pattern recognition, ratio calculation, threshold setting
Threat Detection	Designed alert logic using login ratio anomalies
Behavioral Modeling	Simulated UEBA-style logic for per-user login baselining
SOC Relevance	Mapped outcomes to SIEM alert logic and MITRE ATT&CK techniques