






UNIVERSITY  
OF WOLLONGONG  
AUSTRALIA



SIM  
Global  
Education

# CSCI262 System Security Assignment 3

Group: T05T06-A3-Group20

Members	UOW ID	UOW Email	Signature
Albert Adiputra	7907746	aa734@uowmail.edu.au	
Michelle Caroline Sugianto	7894508	mcs330@uowmail.edu.au	
Francis Teo	7895501	hhft749@uowmail.edu.au	

# Table of Contents

<b>Table of Contents.....</b>	<b>2</b>
Introduction.....	3
Initial Input.....	4
How we store the events and statistics internally.....	5
Events:.....	5
Statistics:.....	5
Potential inconsistencies between Events.txt and Stats.txt.....	6
Activity Engine & Logs.....	7
Event Generation Process.....	7
Log File Structure.....	8
Reasons behind using this format.....	8
Analysis Engine.....	9
Daily Totals and Statistical Data File.....	9
File Format.....	9
Alert Engine.....	13
Conclusion.....	15

# Introduction

For assignment 3, we have implemented an Intrusion Detection System (IDS) command-line interface (CLI) application designed to simulate real-world event data, analyse their behaviour, and detect anomalies.

The application takes two input files: an "Events" file defining the system events and their characteristics, and a "Stats" file specifying the statistical properties of these events. Based on these predefined thresholds, the IDS application will then synthesise and analyse event logs, and attempt the detection of anomalous behaviour(s). Additionally, the IDS provides a simulation feature where users can input a new statistics file along with the number of days to simulate a live scenario. This simulation generates live events, raising alerts if the anomaly count surpasses the threshold.

# Initial Input

1. The application begins by parsing the two input files:

```
python IDS.py Events.txt Stats.txt 30
```

- Events File

```
5
Logins:D:0:100:2:
Time online:C:0:1440:3:
Emails sent:D:0:100:1:
Emails opened:D:0:100:1:
Emails deleted:D:0:100:2:
```

- This file specifies the different event types, their associated minimum and maximum values, and their respective weights. Each event is represented by a line in the format: **event\_name:event\_type:min\_value:max\_value:weight.**

- Stats File

```
5
Logins:4:1.5:
Time online:150.5:25.00:
Emails sent:10:3:
Emails opened:12:4.5:
Emails deleted:7:2.25:
```

- This file provides the mean and standard deviation for each event, derived from historical data. The format is similar to the events file: **event\_name:mean:standard\_deviation.**

```

=====
Initializing Events and Statistics...
Initialization success!
=====

EVENTS DATA
=====
Logins      : {'type': 'D', 'min': 0.0, 'max': 100.0, 'weight': 2}
Time online : {'type': 'C', 'min': 0.0, 'max': 1440.0, 'weight': 3}
Emails sent  : {'type': 'D', 'min': 0.0, 'max': 100.0, 'weight': 1}
Emails opened : {'type': 'D', 'min': 0.0, 'max': 100.0, 'weight': 1}
Emails deleted : {'type': 'D', 'min': 0.0, 'max': 100.0, 'weight': 2}
=====

STATISTICS DATA
=====
Logins      : {'mean': 4.0, 'standard_deviation': 1.5}
Time online : {'mean': 150.5, 'standard_deviation': 25.0}
Emails sent  : {'mean': 10.0, 'standard_deviation': 3.0}
Emails opened : {'mean': 12.0, 'standard_deviation': 4.5}
Emails deleted : {'mean': 7.0, 'standard_deviation': 2.25}
=====

```

## How we store the events and statistics internally

The events and statistics are stored as dictionaries.

### Events:

**events** dictionary structure:

- **Key:** The event name (string).
- **Value:** A dictionary containing:
  - **"type"**: Type of the event (either **"C"** for continuous or **"D"** for discrete).
  - **"min"**: Minimum value (float).
  - **"max"**: Maximum value (float).
  - **"weight"**: Weight assigned to the event (integer).

### Statistics:

**stats** dictionary structure:

- **Key:** The event name (string).
- **Value:** A dictionary containing:
  - **"mean"**: Mean of the event values (float).
  - **"standard\_deviation"**: Standard deviation of the event values (float).

## **Potential inconsistencies between Events.txt and Stats.txt**

1. Mismatch Between Declared and Actual Number of Events in a File

```
if lines_read != num_events:  
    raise ValueError(f"Expected {num_events} events, but found {lines_read} in the file")  
  
return stats
```

The code validates that the number of events specified in the first line of `Events.txt` and `Stats.txt` corresponds to the actual number of lines containing event data. If these counts do not match, an error will be raised to prevent incorrect data processing caused by incomplete or extra entries in the file.

2. Mismatch in the Number of Events between `Events.txt` and `Stats.txt`

```
if len(events) != len(stats):  
    print("Error: Inconsistent number between Events and Statistics.")  
    return
```

This code checks that the total number of events listed in `Events.txt` matches the number of statistics in `Stats.txt`. If the lengths do not match, the program will show an error message and stop running. This ensures that every event has a corresponding set of statistics.

# Activity Engine & Logs

```
=====
Activity Engine and the Logs
=====
Generating events...
Generated 30 days of events!

Generating Logs File...
Event logs written successfully to logs.txt!

Accumulating daily totals...
Calculations Complete!
```

## Event Generation Process

Events are generated using the `generate_event` function, which creates random values following a truncated normal distribution based on the statistics provided in `Stats.txt`.

### 1. Input Parameters:

- **Mean** and **standard deviation** define the centre and spread of the distribution.
- **Min** and **max values** ensure the generated values fall within the specified range.

### 2. Truncated Normal Distribution:

- The `truncnorm` function imported from `scipy.stats` ensures that generated values stay within the defined range by truncating the normal distribution.
- The generated value is then adjusted according to the type of event:
  - **Discrete Events (D)**: The value is rounded to the nearest integer.
  - **Continuous Events (C)**: The value is rounded to two decimal places for precision.

This distinction ensures that discrete events align with real-world scenarios like counts, while continuous events maintain a higher level of accuracy for measurements.

Logs of the generated events will then be stored in `logs.txt`

## Log File Structure

### 1. **File Name and Format:**

The log file is named `logs.txt` and is formatted as a plain text file. Each day's events are grouped together, making it simple to understand and process.

### 2. **Structure:**

```
Day:1
Logins:5
Time online:137.02
Emails sent:6
Emails opened:18
Emails deleted:5

Day:2
Logins:3
Time online:112.27
Emails sent:20
Emails opened:11
Emails deleted:9

Day:3
Logins:2
Time online:136.68
Emails sent:9
Emails opened:9
Emails deleted:8
```

- Each day starts with a header, e.g., `Day:1`.
- Events are listed under the day, with their names and generated values in a `name:value` format.
- A blank line separates each day's data.

## Reasons behind using this format

- **Human-readable:** The format is simple, making it easy to look up event data for each day.
- **Simple structure:** The day count and values for each event are clearly separated, making the file easy to navigate.
- **Easy to use for analysis:** This format makes it easy to calculate statistics like mean or standard deviation and is ideal for further processing or analysis by the program.



## Analysis Engine

```
Analysis Engine
=====
Generating Data File...
Data successfully written to baseline.txt!

Calculating statistics...
Statistics successfully written to baseline_statistics.txt...
```

## Daily Totals and Statistical Data File

The file that stores daily totals and statistical data for each event is called **baseline.txt**. This file is a simple text file that records the cumulative data for each event over multiple days. The daily totals from **baseline.txt** is then used to calculate the mean and standard deviation for each event which is stored in **baseline\_statistics.txt**.

## File Format

**baseline.txt**[illegible]

**Event Data:** For each event, the daily totals are listed. Each row includes event names and their accumulated daily totals

**Structure:** Each day's data for the event is written out clearly, separated by event names, with values listed next to them.

**Day Count:** At the bottom of the file, the total number of days is written as **Day:X**, where X is the total number of days.

## baseline\_statistics.txt

```
5
Logins:3.57:1.63:
Time online:157.02:30.08:
Emails sent:9.87:3.53:
Emails opened:10.4:4.56:
Emails deleted:6.23:2.1:
```

**Event Count:** At the top of the file, in a similar fashion as **Stats.txt**, the number of events is displayed.

**Event Data:** For each event, the mean and standard deviation is displayed in the format of **event:mean:standard\_deviation**

## Event Accumulation

- The **accumulate\_events** function reads the log file (logs.txt) and accumulates the values for each event across all days.

```
def accumulate_events(logs_file):
    event_data = {}
    day_counter = 0

    with open(logs_file, "r") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue

            if line.startswith("Day"):
                day_counter += 1
            elif ":" in line:
                key, value = line.split(":")
                value = float(value)

                if key in event_data:
                    event_data[key].append(value)
                else:
                    event_data[key] = [value]

    event_data["Day"] = day_counter
    return event_data
```

## Baseline File Generation

- The **generate\_baseline\_file** function creates a baseline file (**baseline.txt**) that stores the accumulated event values for each day.

```
def generate_baseline_file(accumulated_events):
    try:
        with open("baseline.txt", "w") as f:
            f.write("Total Statistics\n")
            f.write("=====\n")

            for key, values in accumulated_events.items():
                if key == "Day":
                    f.write(f"{key}:{values}\n")
                else:
                    values_str = ", ".join(map(str, values))
                    f.write(f"{key}: {values_str}\n")

    except FileNotFoundError:
        print("Error: Could not open baseline.txt for writing. Please check the file path and try again.")
    except IOError:
        print("Error: An I/O error occurred while writing to baseline.txt")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

## Statistical Analysis

- The **compute\_statistics** function calculates the mean and standard deviation (s.d.) for each event based on the accumulated values in the baseline file.
- The results are written to a file titled as **baseline\_statistics.txt**.

```
def compute_statistics(baseline_file):
    # Dictionary to store event data across days
    event_values = {}

    try:
        # Read the data from the file, line by line
        with open(baseline_file, "r") as f:
            for line in f:
                line = line.strip()

                # Skip headers or empty lines
                if not line or line in ["Total Statistics", "====="]:
                    continue

                # Split the line into key and values (values e.g.: 4, 2, 5, 2, 1)
                key, value = line.split(":")
                key = key.strip()

                if key != "Day": # Exclude "Day" from the statistics
                    # Convert comma-separated values to a list of floats
                    values = [float(v.strip()) for v in value.split(",") if v.strip()]

                    # Log missing data if values list is empty
                    if not values:
                        print(f"Warning: Missing data for event '{key}' in {baseline_file}.")
                    else:
                        event_values[key] = values

    # Calculate mean and standard deviation for each event and write output
    with open("baseline_statistics.txt", "w") as f_statistics:
        # Write the number of events at the top
        f_statistics.write(f"{len(event_values)}\n")

        for key, values in event_values.items():
            if values:
                mean = round(statistics.mean(values), 2)
                stddev = round(statistics.stdev(values), 2) if len(values) > 1 else 0.0
                # Format the output as Event:mean:standard_deviation:
                f_statistics.write(f"{key}:{mean}:{stddev}\n")
            else:
                # Log fallback if there is no data for a given event
                f_statistics.write(f"{key}:Data missing:Data missing:\n")

    except FileNotFoundError:
        print("Error: Could not open data file. Please check the file path and try again.")
    except IOError:
        print("Error: An I/O error occurred while accessing the data file.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

# Alert Engine

Alert Engine

```
=====
Enter the new stats file for live data analysis (or 'q' to quit): new_stats.txt
Loading new statistics...
Enter the number of days for live data generation: 20
Generating live data for 20 days...
Accumulating live events...
```

Calculating anomaly scores for live data...

```
=====
Daily Reports
```

Anomaly Detection Threshold: 18

```
=====
Day 1: OK - Anomaly Score = 9.9
Day 2: OK - Anomaly Score = 12.09
Day 3: OK - Anomaly Score = 9.56
Day 4: OK - Anomaly Score = 6.77
Day 5: OK - Anomaly Score = 12.84
Day 6: OK - Anomaly Score = 7.46
Day 7: OK - Anomaly Score = 11.67
Day 8: OK - Anomaly Score = 11.04
Day 9: ALERT - Anomaly Score = 18.86
Day 10: OK - Anomaly Score = 6.53
Day 11: OK - Anomaly Score = 9.39
Day 12: OK - Anomaly Score = 8.38
Day 13: OK - Anomaly Score = 12.87
Day 14: OK - Anomaly Score = 15.3
Day 15: OK - Anomaly Score = 7.94
Day 16: OK - Anomaly Score = 15.87
Day 17: OK - Anomaly Score = 9.95
Day 18: OK - Anomaly Score = 9.44
Day 19: OK - Anomaly Score = 15.76
Day 20: OK - Anomaly Score = 13.24
```

```
=====
Enter the new stats file for live data analysis (or 'q' to quit): q
```

## 1. Stats Loading

- The **load\_new\_stats** function prompts the user to enter a new stats file and parses it to obtain updated mean and standard deviation values for each event. The new stats file has to be in the same format as the original one and located within the same directory of the IDS program.

## 2. Live Data Generation

- The **generate\_live\_data** function utilises the new stats and the original event definitions to generate live event data for the specified number of days. This data is stored in **live\_logs.txt** with the same format as **logs.txt**.

## 3. Anomaly Score Calculation

- The **calculate\_daily\_anomaly\_score** function calculates an anomaly score for each day based on the difference of each event's value from its baseline mean divided by the baseline standard deviation, and multiplied by the event's weight.

$$\text{Anomaly Score} = \frac{\text{Event Value} - \text{Baseline Mean}}{\text{Baseline Standard Deviation}} * \text{Event Weight}$$

## 4. Alerting

- The engine first calculates the anomaly threshold by using the sum of weight for every event and multiplying it by 2

$$\text{Anomaly Threshold} = \text{Total Weight} * 2$$

- Using the calculated anomaly score, the alert engine determines whether there is an anomaly based on whether the anomaly score exceeds the threshold or not.
- If an anomaly score exceeds the given threshold, an alert is triggered, and displayed into the command line.

## 5. Repeating the Process

- Once the alert engine has finished its live simulation, it will prompt the user to either input a new statistics file to simulate, or 'q' to end the program.

## **Conclusion**

The IDS application developed effectively simulates real-world event generation and anomaly detection. It provides a flexible framework for customising event definitions, statistical models, and anomaly detection thresholds. By leveraging the truncated normal distribution and statistical analysis techniques, the application can accurately model various event types and identify deviations from normal behaviour. The alerting mechanism allows for timely detection and response to potential security threats or system anomalies.