Pooh Bear Intrusion Detection System

CSCI262 System Security (Spring 2018) Assignment 3 - Honeypot Intrusion Detection System.

Authors	Contact
Andrew (Dinh) Che (codeninja55)	dbac496@uowmail.edu.au
Duong Le (daltonle)	ndl991@uowmail.edu.au

Preface

Overview

Pooh Bear Intrusion Detection System (IDS), to be referred to as “System” or “The System” in this report is a honeypot event modeller and IDS/auditor system written in C++ using ANSI Standard 11. The system reads in vehicles information and traffic statistics and produce events consistent with this data. The collection of generated events forms the baseline data for the system, which are analysed to generate the baseline statistics. In the last phase, "live data" is compared to the baseline statistics to identify anomalies in the system.

Files

Directory Structure

Log files are stored in `dir/logs/` folder and data generated from the Analysis Engine is stored in `dir/data/`.

```
a3_pooh_bear_ids/
....A3_REPORT.md
....A3_REPORT.pdf
....start.sh
....Stats.txt
....Vehicles.txt
....Welcome_ascii
....Exit_ascii
....main.cpp
....Logger.cpp
....Vehicles.h
....Vehicles.cpp
....Utils.h
....Utils.cpp
....ActivityEngine.h
....ActivityEngine.cpp
....AnalysisEngine.h
....AnalysisEngine.cpp
....AlertEngine.h
....AlertEngine.cpp
```

Suffices

Log files and data files generated from statistics of initial input are succeeded with a suffix `_baseline`.

Each time live data is inputed by a user, the log and data files generated from this process are followed by a positive integer incrementing from 1, for example, `data_1, logs_1`, `data_2, logs_2`,...

Initial input

Initial input including vehicles information and traffic statistics are read from files specified by user at the command line arguments.

Usage

```
/a3_pooh_bear_ids$ chmod +x ./start.sh
$ ./start.sh
$ ./Traffic Vehicles.txt Stats.txt [Days]
```

Storing data

`SimTime` structure

The `SimTime` structure is commonly used throughout the system to represent both the time within the system and the time in the real world. The struct is a stripped down version of the `tm` structure as implemnted in the C `<ctime>` library. Importantly, a `simtime_t` type definition is declared in the system to be used as a double to represent a timestamp of each day beginning at 0. The timestamp is often stored and converted in other data structures to represent time movemnt within the system. Public functions are also implemented in the `SimTime` structure to allow easy use throughout the system. The implementation and definitions are stored in the `Utils.h` source file.

`Vehicles` class and `VehicleType` structure

The `Vehicles` class is an implementation of a wrapper to encapsulate member functions, variables, and constants relevant to each vehicle. A structure `VehicleType` stores information and statistics associated with each vehicle type in a dictionary with `string name` as key, including:

- Vehicle name and id
- Registration format
- Volume weight and speed weight
- The mean and standard deviation of the volume and speed.

Each time a new set of statistics is read, the data is stored in an instance of class `Vehicles`, which contains a `map` with the key being a `string name` of vehicle name and the value being the `VehicleType` structure. This instance is referred to as a "vehicles dictionary" throughout the system and is used by both the Activity and Analysis engines.

`VehicleStats`

The `VehicleStats` structure is the representation of each vehicle within the simulation. This struct holds the statistics and key information relating to each vehicle including `SimTime` for arrival and departure, and timestamps for arrivals, departures, parking events. Additionally, it also holds flags to represent whether a vehicle is expected to depart the system via a side road or not or whether the vehicle is permitted to park or speed.

`Event`

An `Event` structure is a representation of the discrete event within the system during generation and simulation. The `Event` structure holds the `SimTime` representation of time, the `EVENT_TYPE` enumerated event type, and `VehicleStats` structure.

Data inconsistencies

When data inconsistencies are found within a file or between input files, the system reports the inconsistency and exits out of the current system and allows the user to enter a new file. Below are the inconsistencies that were handled by the system:

- The number of vehicles being monitored is different to the number of vehicle statistics being listed.
- Vehicle names are inconsistent between vehicles file and statistics file.
- Inconsistencies with integers and doubles used in the file which cannot be converted within the system to an appropriate data type.

Other consistencies that could occur which were not handled may include:

- The use of a different delimiter to separate the numbers rather than `:` as required by the system.
- The use of negative numbers.
- The use of lowercase and uppercase in the vehicle type names which are different.

If any inconsistency is detected by the system, it will immediately abort.

Logging system

A logging module is used consistently across the whole system. Following are 5 logging levels in the order of increasing severity.

Level	Description
NOTSET	No level has been set.
INFO	Confirmation that things are working as expected.
DEBUG	Detailed information, typically of interest only when diagnosing problems.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

Log format

Each log event has the following format

```
Time,Module,LEVEL,LogType,EVENT,AdditionalInformation
```

where:

- **Time** is a string of time with the format `<DD-MM-YYYY HH:MM:ss>`
- **Module** is the module that logged the event, e.g. "Analysis Engine", "Activity Engine", ...
- **LEVEL** is the log level
- **Log type** is the type of the log record based on the type of information it refers to, e.g. "Activity Log", "Vehicle Log", etc.
- **EVENT** is an enumerated type represented in the system as discrete events which consists of the following representations:
    - `ARRIVAL` is the arrival of a vehicle into the simulation at the start of the road and only up to 23:00 on any given day.
    - `PARK_START` is the time the vehicle has started parking on the side of the road.
    - `VEHICLE_MOVE` is the time the vehicle has stopped parking and begun to move again.
    - `DEPART_SIDE_ROAD` is the time the vehicle has departed via a side road and is discard by the system for reporting purposes.
    - `DEPART_END_ROAD` is the time the vehicle has completely traversed the length of the road and an average speed is reocreded by the system for generating speeding tickets. The vehicle has now left the road system.
- **Additional information** is any information deemed necessary for the logged event, e.g. a message or the vehicle type, registration ID, speed, etc...

Activity Engine

Event generation

The Activity Engine is discrete event simulation (DES) model using previously supplied data and random generators. In addition to randomness, the system also uses deterministic variables that are defined in the `Utils.h` source file as configurations to the system. In particular, the system always uses the Mersenne Twister Engine as the Psuedorandom Number Genreator (PRNG) for all random distributions . This is due to the PRNG being a better implementation of pseudorandmness than previous implements such as the linear congruential generator. The implementation for the Mersenne Twister engine was used from the C++ `random` library.

All events for the Activity Engine are stored in a Future Events List (FEL) priority queue from the C++ Standard Template Library. A `Events` structure is used as the elements of the queue to store an ordered `SimTime` for the model. A custom compare method is passed to the constructor to use custom `SimTime` structure as a representation of time.

Generate arrival events

- Using the Gaussian distribution (as defined in the C++ `random` library, the number of occurences and speed for the vehicle type are randomly generated from the supplied mean and standard deviation statistics of the original `Stats.txt` file.
- The system then uses the number of vehicle types for that day to calculate a rate of occurence up to the arrival limit of 23:00. This rate of occurrence is calculated in seconds to allow better precision. Using a Poisson process and the exponential distribution, the rate of occurrence is used to randomly generate arrival times for each vehicle type consistent with the following probability density function.

$$p(x|\lambda) = \lambda e^{-\lambda x},\ x > 0$$

This process is consistent with other models to generate “arrival times.” The probability of the event occurs as the next interval over time (in this case) of an event occurring according to its rate of occurrence.

The arrival times are stored in a vector list which is then processed to generate further discrete events and information for the Analysis and Alert engines.

- The system uses a static method to generate registration numbers that are unique and consistent with the supplied format from the `Vehicles.txt` file.
- The system uses a Bernoulli distribution to decide whether a vehicle will depart via a side road or not.
- The system then generates two important properties stored within the `VehicleStats` structure named `estimate_departure_delta` and `estimate_departure_time`. Both these variables are calculated based on the arrival time and the arrival speed of each vehicle. The `estimate_departure_delta` is an estimate of the time, in seconds, it will take the vehicle to arrive at the end of the road if it were travelling consistently at it’s arrival speed. Some randomness is used to further estimate the `estimate_departure_time`, which is a timestamp that is estimated using a Gaussian distribution with the `estimate_departure_delta` as the mean and a standard deviation of 5 for vehicles departing at the end of the road and 15 for vehicles departing via the side.

Generate parking start and vehicle move events

- The system initially uses a Binomail distribution to randomly generate the number of times a vehicle will park with a constant average parking number and parking probability as deterministic values as configurations.
- The system then uses the Poisson process once again to generate random parking timestamps consistent with the rate of occurrence of 1 over average parking event (another deterministic constant).
- Given a list of parking timestamps, the system then uses the Poisson process to generate the parking durations. This is done by first finding the the next likely delta interval of the vehicle moving again. This timestamp is then used to generate the duration for the parking event.

Generate departure events

- Depature events are generated separately as either via a side road or end of road. As described above, the system uses a Gaussian distribution to estimated the most likely time a vehicle will depart the road system. Randomness is only used in the sense that the distributions can produce outliers within a standard deviation of 15 (for the vehicles departing via side roads) and 5 (for vehicles departing at the end of the road).
- If the vehicle has at least one parking event, the system sums the time spent parking and adds it to the departure time stamp.
- An average speed is calculated if the vehicle has departed via the end of the road.

After the events are generated and stored in the priority, the activity engine processes this FEL and log events to `dir/logs/logs_suffix` using the custom `Logger` module and custom `VehiclesLog`, `GenericLog`, and `ActivityLog` structures.

Errors

Errors may happen during the execution of the Activity Engine. The table below lists the possible errors and the actions that will be performed by the system if they occur.

Errors	Actions
The time of an arrival event exceeds the time limit of arrival events (23:00)	A vector of 1000 timestamps is created. If there are N vehicles arriving that day, the first N eligible timestamps will be chosen.
File errors (cannot open log file)	The system will reports the error to the console. If this error occurs, the system may still be able to run but data is likely to be lost.

Analysis Engine

Storing data

The Analysis engine reads the event logs generated by the Activity Engine and generates associated statistics.

Each time a vehicle enters the road, a `VehicleStats` struct is created and added to a `map` (key: `string` of registration ID, value: `VehicleStats` containing vehicle information). The vehicle volume and speed distributions of that day are also updated. When the vehicle leaves the road, its struct is simply taken off the `map`.

The daily totals (volume and average speed for each vehicle type) are stored in `dir/data/data_suffix`, the speeding tickets are stored in `dir/data/speeding_suffix` and the generated overall statistics is stored in `dir/data/stats_suffix`. The `data` and `speeding` files are written with the CSV format with a header, while the `stats` file has the same format as the original inputed `stats` file.

All activities of the Analysis engine are also continued to be logged in `dir/logs/logs_suffix`.

Possible anomalies

Anomalies in reading the logs can include reading incorrect formatting. However, the authors of the system took much care in ensuring the format was consistent across all logs, as demonstrated by the use of a template class `Logger` module and custom made Log structures.

Possible anomalies with statistics may include no deviation from the mean for certain vehicles as the supplied mean may be too low. In addition, a small number of days and iterations of the generation may not be large enough to produce large variations in the statistics. As a result, a small standard deviation or volume mean will result in very little difference in the system on a daily basis.

Another anomaly to consider is whether the road system will allow multiple vehicles to park within the same spot and time. At this stage, the system ignores this possible strange occurrence that would not be possible in a real world system.

Alert Engine

The Alert engine reads the `data` file associated with the "live data" and compare it to the baseline statistics (stored in `stats_baseline`). For each day, the Alert engine adds up the anomaly counters and compare them to the anomaly thresholds, then reports *"Volume/Speed anomaly detected"* or *"No volume/speed anomaly detected"* as appropirate.

Activities of the Alert engine are also continued to be logged in `dir/logs/logs_suffix`.