Monitor

A login activity monitor to stream log data, and identify suspicious activity.

Author: Ahad Sheriff

This code was written for a Take Home Project from Datadog. Contact: ahadsheriff@gmail.com

Project Requirements

The initial requirements for the project stated that it should have the following functionality:

- Real-time Streaming: Treat the traffic.csv file like a real-time event stream. Emulate a stream by reading one line of the csv file per second.
- Enrichment: Enrich events with any context you think is valuable.
- Statistics: Every 15 seconds, provide a statistical overview of the web traffic (you decide on what stats are important).
- Threat Detection: Pick one type of anomalous or suspicious activity and generate an actionable alert when it is detected.
- Future Improvements: In your documentation, explain how you would improve on the design of this application.

Installation

This Python program can be run with Docker.

Build:

```
docker build -t monitor .
```

Run:

```
docker run -it monitor
```

Here is what the Dockerfile looks like for reference:

```
FROM python:3
ADD monitor.py /
ADD traffic.csv .
RUN pip install user-agents
CMD [ "python", "./monitor.py", "-f", "traffic.csv"]
```

Documentation

The entire program is built around the data structure `login_stats` which is what I used to store statistics and monitoring data. `login_stats` is a dictionary that is structured as follows:

```
{
    "total_severe": 0,
    "total_warning": 0,
    "brute_force_users": {
        "userid": 0
    },
    "compromised_users": {
        "userid": 0
    },
    "deny_list": {
        "fingerprint": 0
    },
    "watch_list": {
        "fingerprint": 0
    },
    "success":
        {
            "total": 0,
            "users": {
                "userid": {
                    "fingerprints": [],
                    "attempts": 0,
                },
            },
            },
    "fail":
        {
            "total": 0,
            "users": {
                "fingerprints": [],
                "attempts": 0,
            },
        }
    }
```


Usage

Field	Description
total_severe	Tracks the total number of severe anomalies
total_warning	Tracks the total number of warning anomalies
brute_force_users	Tracks all users who may be affected by a brute force attack + number of times targeted.
compromised_users	Tracks all users who's account credentials may be comrpomised + number of times targeted.
deny_list	Tracks the fingerprints (unique ID's) of sources we want to block access to our systems.
watch_list	Tracks the fingerprints (unique ID's) of sources we want to monitor for future attacks.
success	Tracks all successful logins by userid, and keeps fingerprint ids of login origin and track the number of attempts.
fail	Tracks all failed logins by userid, and keeps fingerprint ids of login origin and track the number of attempts.

How It Works

1. Stream Logs

Once the data structure is initialized, the program calls upon the `stream_logs()` function to process the input file and parse the event data. The function reads the `csv` file into a dictionary object, then parses one line of data per-second, and sends the data to the `process_data()` function. Every 15-seconds, the function calls `get_stats()` to print monitoring statistics to the console.

2. Process Data

The purpose of the `process_data()` function is to extract relevant data from an event log. This data includes `userid`, `ip`, `browser`, `os`, and `device`. The `browser`, `os`, and `device` information is extracted from the `useragent` field of the event log via the third-party library `user-agents`.

Once this data is extracted, the function uses the built-in `hashlib` library to generate a MD5 hash of the event data:

```
event_data = userid + ip + browser + os + device
event_fingerprint = hashlib.md5(event_data.encode())
event_fingerprint_id = event_fingerprint.hexdigest()
```

This hash value becomes the `fingerprint` of the login, which is used as a unique identifier for the event.

3. Successful or Failed Logins

Based on whether the login was a success or failure, the `fingerprint` is passed along with the `userid` into the `login_success()` or `login_fail()` functions respectively. These functions will print the login event information to the console and add the specific event (`success` or `fail`) into the `login_stats` data structure.

This function also checks to see if the number of failed or successful logins for the user is abnormal, and will call `failed_login_anomalies()` or `successful_login_anomalies()` accordingly.

4. Failed Login Anomalies

Multiple failed logins from a single, or many sources, are indicators that the event may be a login brute-force atack. The rules to trigger this anomaly are:

1. (WARN) If a user has less than 5 failed login attempts from the same fingerprint, this could just be a user forgot their password. The program should notify user of suspicious activity, and suggest password reset.
2. (WARN) If a user has less than 5 failed login attempts from the multiple fingerprints, this indicates that an attacker(s) may be trying to gain access to the account via many agents. We then notify user of suspicious activity, and suggest password reset. We should also add the guilty fingerprints to a watch list to monitor for future attacks.
3. (SEVERE) If a user has more than 5 failed failed login attempts, from the same origin, this could possibly be an attacker attempting a brute-force login. We notify the user of suspicious activity, and explicitly require a password reset. The fingerprint should be added to a deny list to prevent future attempts.
4. (SEVERE) If a user has more than 5 failed login attempts from the multiple fingerprints, this is a indicates that an attacker(s) may be trying to gain access to the account via many agents. We then notify user of suspicious activity, and suggest password reset. We notify the user of suspicious activity, and explicitly require a password reset. The fingerprints should be added to a deny list to prevent future attempts.

5. Successful Login Anomalies

Multiple successful logins from a single, or many sources, are indicators that an account's credentials may have been compromised an attacker has gained access to a user account. Multiple logins within a few seconds is definitely abnormal behavior. The rules to trigger this anomaly are:

1. (WARN) If a user has less than 3 successful login attempts from multiple origins, it could be possible that the user has logged on from multiple devices. Notify user of suspicious account login, and ask to verify that it was them.
2. (SEVERE) If a user has more than 3 failed successful login attempts, from multiple origins, this could possibly be an attacker who has gained credentials to a users account and is attempting to access from many hosts. Or it could simply be a user logging into many devices. Regardless, we should notify user of suspicious account login, and suggest a password reset. We should also add the fingerprint ids for the event to the watchlist.
3. (SEVERE) If a user has more than 3 failed successful login attempts, from the same origin, this is could be a sign that an attacker is able to get into to a users account and is attempting to keep access by logging in several times. Notify user of suspicious account login, and require password reset. We should also add the fingerprint id to the deny list.

Future Work

There are several ways in which we could improve this program in terms of both functionality and design.

1. Collect additional metadata to improve anomaly detection - Supposing our logs had additional metadata related user login events, we could create even more powerful anomaly detection rules. Additional types of metadata could include `timestamp`, which would allow us to identify logins at odd hours of the day, and `location` data could help identify if a user logged in from somewhere random.
2. Command-line Interface - While the current monitoring solution is simple to use, we could add more options for filtering and viewing data with a simple command-line interface. We could add functionality to filter only certain event types, pipe our statistics to a log file, search by fingerprint or userid, and nuch more.
3. Improve `login_stats` data structue - This data structure is the brains of the program, and is a little unorganized in its current state. Future improvements would include being more meticulous about its design so that we can query data more efficiently.
4. Design for scale - The monitoring tool in its current state may not be able to scale well in a real world situation. Processing logs one second-at a time would be great, but in reality we should expect to monitor logs concurrently. Supporting concurrency requires rethinking the design of the program, as many calls in the `Monitor()` class may slow down the program at scale.
5. Actual real-time data streaming - Feeding data in via CSV is cool, but processing login event data in real-time from an actual data source would be even cooler!