Prophet Security Take-Home Exercise

Table of Contents

---

1. Introduction

This backend service is designed to process user events, identify and flag suspicious activity based on user and IP history, and manage suspicious IP ranges. Developed primarily in Python, this service leverages the FastAPI framework for high performance and efficient request handling, and PostgreSQL for reliable data persistence.

2. Architecture Overview

The application consists of:

- **FastAPI**: Used as the web framework to provide an asynchronous API layer for handling user events.
- **PostgreSQL**: Serves as the data storage solution, chosen for its efficiency in managing relational data and supporting advanced CIDR range matching.
- **Python**: Powers the backend development, chosen for its ecosystem and robust support for asynchronous processing.

Data Flow:

1. **IP Range Management**:

   Suspicious IP ranges are configured via CRUD operations through dedicated API endpoints. These IP ranges are stored in PostgreSQL with CIDR types, allowing efficient containment checks.

2. **Event Processing**:

   User events are received through the `/process-event` endpoint. Each event is assessed to determine whether it is suspicious, based on factors such as IP address and previous flagged events.

3. **Suspicion Flagging and Logging**:

   All processed events are logged in PostgreSQL with their `suspicious` status, creating a historical record that tracks each event's assessment. Flagged events are marked for further review, and they can be retrieved for analysis through a specific endpoint.

---

3. Setup Instructions

On Local Machine

Prerequisites:

- **Python 3.8+**
- **PostgreSQL database**

Steps:

1. **Clone the repository:**

```
git clone https://github.com/TMhalsekar/prophet-security-takehome
```

2. **Install dependencies:**

```
pip install -r requirements.txt
```

3. **Configure PostgreSQL** in `database.py` with your credentials and database name:

```
DATABASE_URL = "postgresql://{username:password}@localhost:5432/{database_name}"
```

4. **Start the FastAPI server:**

```
uvicorn app:app --reload
```

5. **Access the application** at http://localhost:8000 and view the API documentation at http://localhost:8000/docs, where you can also test and interact with the application's endpoints directly.

Using Docker

1. **Clone the repository:**

```
git clone https://github.com/TMhalsekar/prophet-security-takehome
```

2. **Run with Docker Compose:**

```
docker-compose up --build
```

3. **Access the application** at http://localhost:8000 and view the API documentation at http://localhost:8000/docs, where you can also test and interact with the application's endpoints directly.

---

4. Key Features and Endpoints

After launching the app, navigate to http://localhost:8000/docs to view the OpenAPI documentation. Here is a brief overview:

1. Manage Suspicious IP Ranges

- **Create IP Range**
  Endpoint: `POST /ip-ranges/`
  Description: Adds a new IP range.
  Request Body: `{ "cidr": "173.99.253.0/24" }`
- **List IP Ranges**
  Endpoint: `GET /ip-ranges/`
  Description: Retrieves all IP ranges.
- **Delete IP Range**
  Endpoint: `DELETE /ip-ranges/{id}`
  Description: Deletes a specified IP range.

2. Submit Events

- **Process Events**
  Endpoint: `POST /events/`
  Description: Processes events in bulk, flagging any that match suspicious criteria.

  Request Body:

```
[
    {
        "timestamp": "2024-01-01T02:29:28Z",
        "username": "faith36",
        "source_ip": "215.248.64.140",
        "event_type": "file_download",
        "file_size_mb": 48,
        "application": "email",
        "success": true
    }
]
```


3. Retrieve Suspicious Events

- **List Suspicious Events**
  Endpoint: `GET /suspicious-events/`
  Description: Fetches all flagged events, with optional pagination parameters (e.g., limit, offset).

---

5. Design Decisions

a. Framework Choice: FastAPI

- **Rationale**: FastAPI offers high-performance asynchronous support, automatic documentation, and easy scalability, making it ideal for real-time processing.
- **Benefits**: The framework handles high data volumes efficiently, providing low-latency processing with minimal setup. FastAPI automatically generates OpenAPI specifications, providing a built-in, interactive API documentation at `/docs` (Swagger UI) and `/redoc` (ReDoc)

b. Database Choice: PostgreSQL with SQLAlchemy

- **Rationale**: PostgreSQL is robust for structured data and complex queries, supporting CIDR for efficient IP matching.
- **Benefits**: Leveraging PostgreSQL's CIDR type simplifies IP containment checks, while SQLAlchemy allows for quick model setup and reliable CRUD operations.

c. Data Model Design

- **IP Ranges**: Stored as CIDR ranges for efficient containment checks in PostgreSQL.
- **Flagging Mechanism**: Events are stored with timestamps and flags, allowing for later updates based on new data.

d. Separation of Concerns

- **Rationale**: Modules are organized to maintain clear separation of concerns.
- **Benefits**:
    - **Modularity**: Each file serves a specific function, simplifying maintenance.
    - **Reusability**: `crud.py` and `models.py` components can be reused across application modules.
    - **Scalability**: Independent components enable modular growth without tight coupling.

e. Handling Duplicate Events

- **Rationale**: Currently, duplicate events may be processed and stored without a strict duplication check, as proving an event is a duplicate requires comparing all columns for an exact match. Implementing a full duplication check on each event could introduce performance overhead with the current setup.
- **Benefits**: Allowing duplicate events simplifies initial processing but may lead to unnecessary data storage and processing. Future improvements could involve adding a composite unique constraint or hashing method for relevant columns, reducing storage and computational load by preventing exact duplicates from being stored.

---

6. Trade-offs and Considerations

1. Python vs. other languages

- **Trade-off**: Python was chosen for its readability, versatility, and extensive libraries that support rapid API development. Python's simplicity and large developer community make it well-suited for RESTful APIs, data processing, and integration with various databases and third-party services. However, Python is generally slower than languages like Go or Rust, which are optimized for low-level performance and concurrent execution.
- **Impact**: The choice of Python enables rapid development, easy maintenance, and access to rich libraries (e.g., FastAPI, SQLAlchemy, asyncpg). Python’s async capabilities, combined with FastAPI, help manage concurrency, which is essential for handling multiple API requests efficiently. While Python may not be as performant as lower-level languages, it provides a balance between development speed and scalability. This choice supports a faster go-to-market for the API but could limit performance under extreme high-load conditions, where languages with lower latency might offer advantages.

2. FastAPI vs. More Complex Frameworks

- **Trade-off**: FastAPI was chosen over larger frameworks (e.g., Django) due to its simplicity, modern async support, and performance efficiency for RESTful APIs. Unlike Django, which provides a broader range of features for full-stack applications,

FastAPI is specialized for building fast, lightweight APIs with asynchronous capabilities, making it well-suited for microservices.

- **Impact:** This decision optimizes performance and resource usage, especially in handling concurrent requests. FastAPI's async features and automatic documentation generation (Swagger and ReDoc) streamline development and API testing. However, FastAPI lacks some out-of-the-box features that Django provides, such as an integrated admin interface, a fully-fledged ORM with migrations, and tools for handling complex business logic in large applications. This choice supports a clean, minimal API structure but may require additional tooling for advanced use cases in larger systems.

3. Relational Database vs. NoSQL Database

- **Trade-off**: PostgreSQL, a relational database, was chosen over NoSQL options due to its robust support for complex queries, indexing, and data integrity, which align well with the project's structured data requirements. While NoSQL databases offer flexibility and easier horizontal scaling, PostgreSQL's relational model is a better fit for this project, as it involves structured data and relational querying.
- **Impact**: PostgreSQL allows efficient querying and filtering through advanced indexing and supports ACID transactions, ensuring data consistency, which is crucial for flagging and managing event data accurately. Additionally, PostgreSQL's support for JSON and full-text search offers flexibility for semi-structured data if needed in the future. However, this choice may impact horizontal scalability if the data volume grows significantly, as relational databases generally scale vertically.

4. Asynchronous Database Access

- **Trade-off**: Using asynchronous database access allows for non-blocking operations, which is crucial in high-throughput environments.
- **Impact**: While beneficial, asynchronous programming increases code complexity and requires careful handling of resources, especially with SQLAlchemy. However, this is necessary for handling real-time processing efficiently.

5. IP Range Checks in Application vs. Database

- **Trade-off**: Using PostgreSQL's CIDR type for IP range checks is efficient for the current setup, but this approach ties the application closely to PostgreSQL.
- **Alternative**: Implementing IP range checks within the application logic would allow for database flexibility (such as using a NoSQL database), though it would likely result in slower, less optimized queries and add processing load to the application.

6. Database Indexing: Performance vs. Storage and Write Speed

- **Trade-off**: Multiple indexes were created on key columns to optimize query speed, especially for filtering and retrieving flagged events, which are essential for the user experience. Indexes enable faster retrieval by reducing the time required to search

through rows of data. However, they increase storage requirements and can slow down write operations since each new entry must update all relevant indexes.
- **Impact**: The primary benefit is improved query performance, making data retrieval faster and more efficient for users when viewing or filtering events. This enhances responsiveness in high-traffic scenarios where users expect rapid access to flagged or filtered event data. However, the additional storage requirements and potential slowdown in write operations are trade-offs. Given the priority of read performance in this application, especially for frequently accessed event data, the impact of slower writes is manageable, and the overall user experience benefits from faster query response times.

7. Data Storage and Persistence of Events

- **Trade-off**: Storing every event permanently could lead to significant storage costs and degrade database performance over time.
- **Alternative**: Implementing a data retention policy where events are purged after a certain period (or archiving older events) would save storage and improve performance, but it may reduce the ability to conduct long-term analyses on historical data.

---

7. Challenges and Solution Strategy

1. Efficient IP Containment Check

- **Challenge**: Checking if an IP address falls within a list of CIDR ranges can be costly, especially with numerous ranges. Initial implementation involved in-memory checking in Python using libraries, leading to high compute time and multiple database fetches per call.
- **Solution**: PostgreSQL's native support for CIDR and IP address data types enables efficient IP containment checks directly within the database. By using PostgreSQL's `INET` and `CIDR` types, along with operators like `<<` for subnet containment, we can determine if an IP address falls within any specified range directly in SQL queries. This approach avoids the need to fetch data for in-memory checking and takes advantage of PostgreSQL's indexing capabilities on CIDR columns, allowing for fast, optimized lookups even with large lists of CIDR ranges.

2. Supporting Bulk and Single Event Processing

- **Challenge**: The endpoint needed to handle both single and bulk events, requiring flexible input handling and consistent processing logic
- **Solution**: FastAPI's type hints and Pydantic models allow the endpoint to accept both a single event object and a list of events. Single-event submissions are converted to lists internally to handle both cases uniformly. This approach eliminates duplicated logic and ensures consistency across single and bulk event processing.

3. Managing Real-time Event Flagging

- **Challenge**: Real-time processing of events poses scalability challenges, particularly when the system needs to handle large volumes of incoming events without degradation.
- **Solution**: By adopting FastAPI's asynchronous capabilities, the application can handle multiple requests concurrently, improving response times and enabling the service to scale more efficiently

4. Fetching large amount of event data

- **Challenge**: Fetching suspicious events was taking a significant amount of time due to the large dataset size.
- **Solution**: Implementing `limit` and `offset` parameters in the query allows for efficient pagination, reducing response time and database load by retrieving only a subset of events per request. This approach optimizes performance and improves user experience when handling large datasets.

5. Optimized checks

- **Challenge**:Performing checks for flagged IPs and users was impacting performance due to the need to search through potentially large datasets.
- **Solution**: The `EXISTS` clause in SQL was used to quickly verify if an IP or user is flagged. This approach is efficient because `EXISTS` stops searching once a match is found, reducing query time and resource usage. Using `EXISTS` ensures faster response times for checks, especially in scenarios with large numbers of records.

6. Handling Asynchronous Database Operations

- **Challenge**: Working with asynchronous database operations can introduce complexity, especially with SQLAlchemy, which traditionally supports synchronous access.
- **Solution**: By using the `databases` library with SQLAlchemy, we enabled async database interactions, making the service non-blocking and capable of handling multiple requests in real-time. Proper async error handling and resource management were critical for stability.

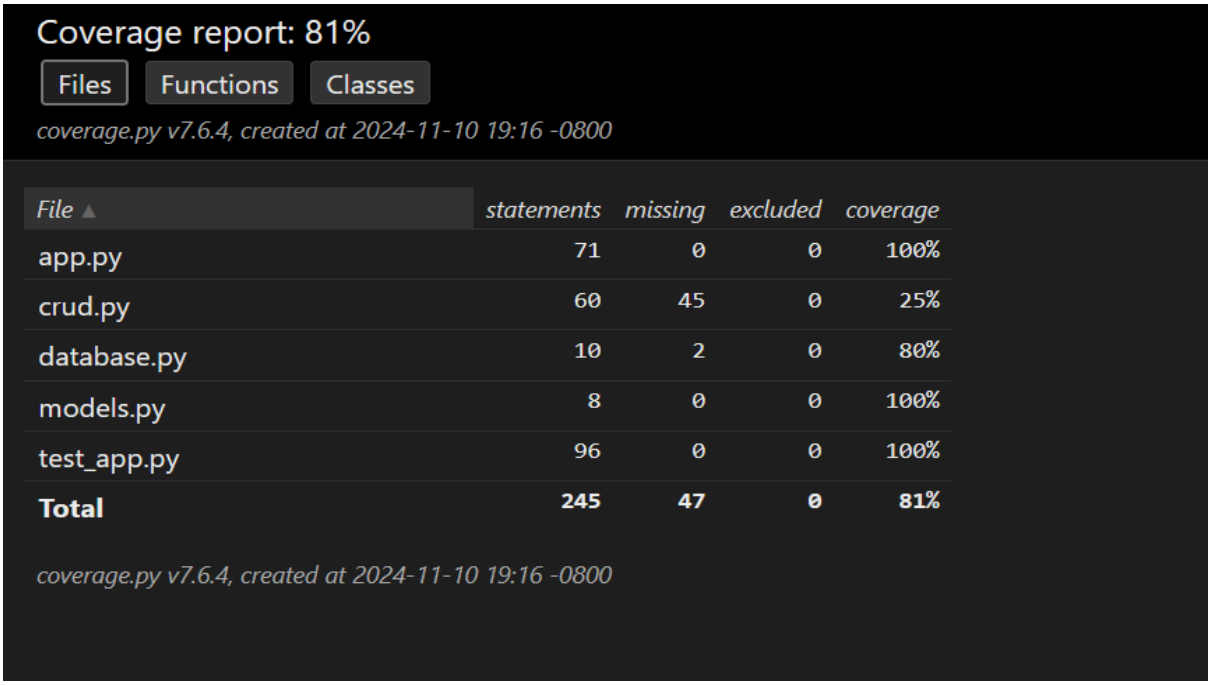
---

8. Testing Strategy

The testing strategy for this project uses **pytest** along with **AsyncClient** from `httpx` to make asynchronous HTTP requests for testing FastAPI endpoints. **Mocks** are utilized to simulate database interactions and to handle specific exceptions, ensuring isolated and reliable test cases.

Coverage and Focus

- **Coverage**: The test suite includes scenarios for valid data handling, error handling, and database interactions, covering the critical functionalities of each endpoint. This



ensures that all primary aspects of the service are robust and handle expected and edge cases.

- **Current Status**: Achieved 100% test coverage for `app.py`.
- **Future Testing for CRUD**: For `crud.py`, I would configure a dummy test database to validate database-specific operations and interactions directly. However, due to time constraints, this implementation was not completed.

Refer to the `htmlcov` folder in the repository for detailed coverage reports, including the specific functions and lines tested. The following is a screenshot of the coverage results for further reference.

---

9. Future Improvements

1. Improving Real-Time Performance of Flagging Suspicious Events

- **Trie-Based IP Lookup**: To optimize IP lookups, a trie-based approach can be implemented. Since the application likely has a high query-to-update ratio, a static in-memory trie structure can reduce latency for IP range checks, assuming the dataset can fit in memory and doesn't require frequent updates. This approach would significantly decrease flagging time for events.
- **Caching Mechanism**: Leveraging a caching layer (such as Redis) can further speed up flagging by storing frequently accessed data in memory, reducing the need for repeated lookups.
- **Event Prioritization**: Prioritize event processing based on type or specific criteria to enhance resource utilization. For instance, unsuccessful events could be assigned lower priority, allowing successful events to be processed immediately. This would

optimize resource allocation, focusing on events with a higher likelihood of suspicious behavior.

2. Unflagging Events

- **Dependency Tracking**: Implement dependency tracking between flagged events to allow efficient un-flagging of events flagged solely due to previous suspicious activity. A graph or linked data structure can trace flagged events back to their originating suspicious event, providing a clear dependency chain.
- **Recursive Un-flagging**: Utilize a recursive approach to un-flag events up the chain to the root event. By storing a `parent_event_id` for each flagged event, the system can track the lineage of suspicion and un-flag all dependent events if necessary.

3. Tolerating Out-of-Order Events

- **Event Buffering**: Use a short delay buffer to allow events to arrive and be reordered by timestamp before processing. This is beneficial for handling minor out-of-order events, where events within a short interval (e.g., 1 second) can be sorted and processed in the correct order.
- **Asynchronous Dependency Checks**: In cases where an out-of-order event affects previously processed events, an asynchronous job can periodically check for missed dependencies, re-evaluating flags if needed. However, frequent database polling may increase resource usage and latency, so this approach should be balanced with performance considerations.

4. Enhanced Search Functionality

- **Expanded Filtering Options**: Currently, filtering is primarily time-based. Future iterations can include additional filters such as event type, username, or other relevant attributes. This would improve the ability to locate specific events and conduct more granular analyses.

5. Degree of Suspicion

- **Suspicion Scoring**: Implement a suspicion scoring system to provide a more nuanced risk assessment. Factors like the number of suspicious events originating from the same IP or other key indicators can contribute to a cumulative suspicion score. This score would help prioritize high-risk events and streamline response efforts.

6. Custom Rules

- **Dynamic Rule Configuration**: Add support for custom rules that define criteria for marking an event as suspicious (e.g., flagging only failed login attempts). Allow these rules to be modified dynamically through an API endpoint, enabling tailored configurations for different operational needs.