Introduction

This project is created to process daily updates of software vulnerabilities from OSV databases. This data pipeline project ingests data from the data sources and processes data to finally store as delta tables in Azure delta lake for efficient querying and retrieval.

Overview

The project contains three main layers: Data Ingestion, Data Processing, Data Storage and retrieval for use.

Data Model :

Raw files, Intermediary and Validated files - These folders contain the json files from the OSV database partitioned and stored based on the modified date. If you want older files, modify the dag script with earlier dates to ingest them.

Field	Type	Description
id	String	Vulnerability identifier
summary	String	description
details	String	Full description
aliases	array	Alternate IDs
modified	string	Last modified date
published	string	Publication date
Schema version	string	Schema version used

Processed files

Vulnerabilities table

Column	type	Nullable	Description
vulnerability_id	string	No	Unique identifier
summary	string	yes	desc.
details	stirng	yes	Desc in detail
aliases	array[string]	No	Alternative id

modified	String timestamp	No	Last modified date
published	String timestamp	yes	Published time stamp
Schema_version	string	No	Schema version used for the record.

Fixed version - table

col	type	Nullable	description
vulnerability _id	string	No	identifier
package_name	string	No	name
ecosystem	string	No	git/Hex etc
fixed_version	string	yes	Version where issue was fixed
introduced_version	string	yes	Version where issue was identified.

Dags

Dag 1 (osv_ingestion_dag.py): Responsible for pulling in vulnerability data from the Open Source Vulnerabilities (OSV) database every day. It downloads the latest dataset from multiple sources in Google Cloud Storage, extracts the data, and filters out only the JSON files. Then, it checks if each file has the required fields like id, alias, affected, and modified before moving valid ones into a final directory. If something goes wrong, it logs errors and retries where possible.

Dag 2 (osv_data_processor.py): Runs a Spark job every day to process the validated vulnerability data. It executes a Python script that takes the most recent batch of data and processes it, making it ready for efficient querying. This involves structuring the data in a better format, optimizing it for faster searches, or preparing it for analytics. Basically, it ensures the data is cleaned up and stored properly in the data lake so it can be used later.

Data Lake Architecture for OSV Data Ingestion

This architecture uses **Delta Lake** to efficiently store and manage OSV data. Considering our needs such as Time travel & rollback capabilities, schema enforcement and evolution, ACID transactions and Optimized queries

Architecture Overview

- **Storage Format**: **Delta Lake** (backed by Apache Parquet)
- **Partition Strategy**: Based on **modified date (YYYY/MM/DD)**
- **Indexing Approach**: Delta's **Z-order indexing** to speed up time-range queries
- **Governance & Access Control**: Lakehouse ACLs & RBAC
- **Data Retention & Cleanup**: depends on how this data is used. Some vulnerabilities may not be resolved immediately. We might have to store the historical data of the version. But the json file can also contain the affected version. My rough estimate would be to hold for one year.

Data Lake Architecture Using Delta Lake

Storage Format

We can use **Delta Lake** format, which stores data in **Parquet files** with additional transactional metadata.

The raw files- the json files from OSV data can be saved in Azure Blob storage. The validated and filtered files in Azure file storage.
Delta enables **schema enforcement, time travel, indexing, and efficient reads** compared to plain Parquet.

Partition Strategy

To optimize performance and query efficiency, we will partition data by:

1. **Date (Modified Date in `YYYY/MM/DD`)**

```
osv-datalake/
      ├── year=2025/
      │   ├── month=02/
      │   │   ├── day=24/
      │   │   │   ├── data.parquet
      │   │   │   ├── data2.parquet

      ├── year=2025/
      │   ├── month=02/
      │   │   ├── day=24/
      │   │   │   ├── data.parquet
```

Advantages:

Queries are **efficient** by filtering on **ecosystem & time range**. Faster **incremental updates** when adding new data.

Indexing Approach

We use Delta Lake's Z-Ordering to speed up searches. This basically sorts the data in a way that keeps related stuff close together, so when you look for something, it doesn't have to scan everything but just the relevant parts. This makes queries much faster.

Time Travel & Rollback Capabilities

Delta Lake supports **time travel**, allowing queries on past snapshots. This Helps **rollback incorrect updates**. Supports **audits & historical analysis**.

Data Governance & Access Controls

Lakehouse Security Controls

- **Role-Based Access Control (RBAC)** via **AZURE IAM**
- **Encryption at rest** (Azure Blob encryption)

- **Audit Logs** using Delta History **-** Prevents unauthorized access. Tracks changes for compliance.

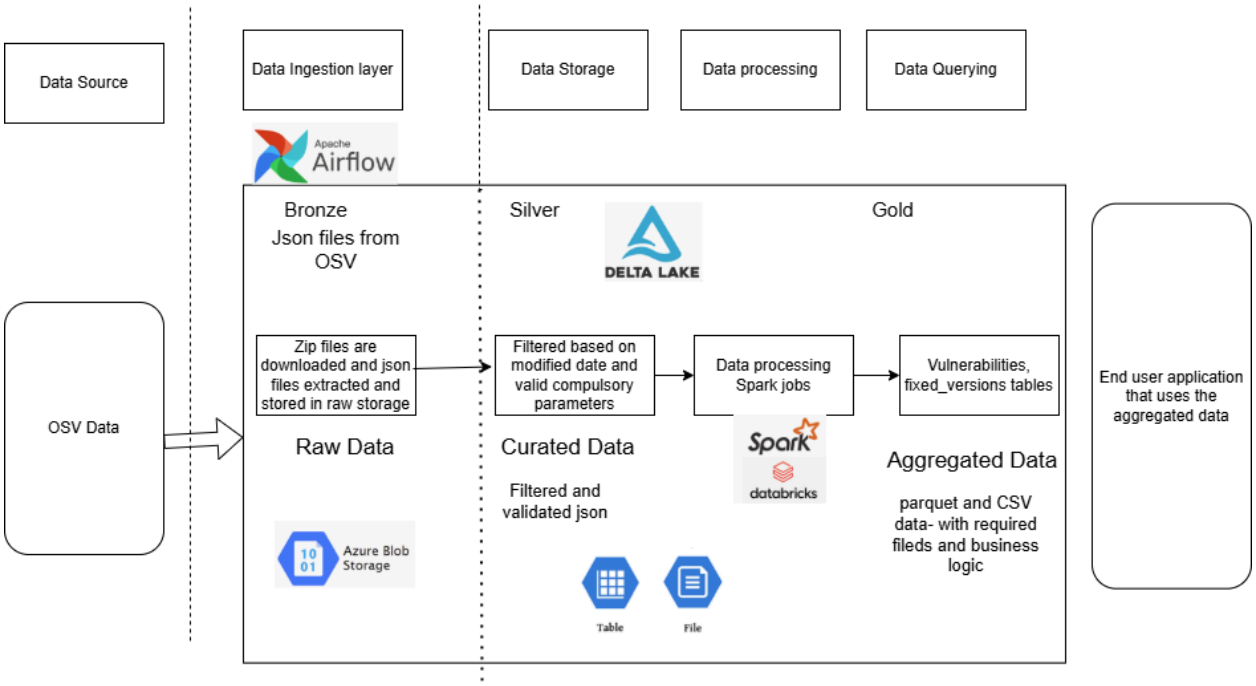
Vacuum & Retention Policies

One year of retention recommended. This reduces storage costs and helps in efficient querying.

Architecture diagram

– data processing layer - spark job

– dags= job 1 data ingestion , job 2 data processing



Terraform

The Terraform files provision the core Azure infrastructure for my data lake project. The main configuration sets up a resource group, an ADLS Gen2-enabled storage account with a dedicated container for both raw and Delta Lake data, an Azure Databricks workspace for running Spark jobs, and an AKS cluster to host Apache Airflow. This structure forms the backbone of the solution, ensuring scalable storage and compute resources that integrate seamlessly with data ingestion and processing workflows.

Additionally, I've separated out a Databricks-specific configuration file that creates a Spark cluster preconfigured with custom settings—such as increased shuffle partitions, optimized Delta Lake settings, and enhanced memory parameters. Variables are defined in a dedicated file for easy customization, and outputs expose key resource details for downstream integrations.

For packaging the project, further steps are required to prepare the application for production. Specifically, we need to containerize the Airflow environment by creating a Docker image that installs all dependencies from my requirements.txt file and includes my DAGs and plugins, ensuring consistency when deployed on AKS. Similarly, for the Spark jobs on Databricks, we need to build a Python wheel using the requirements.txt, which can then be uploaded and attached to the Databricks cluster.

SQL Queries

I have included sample SQL queries on the script called test_spark.py in the data-processing folder along with other scripts.

Deployment guide

The data ingestion script downloads the files using google cloud cli apis. For this I had to create a service account on GCP to generate my private keys with which I would access the data. For this project's scope, insert a sa-keys.json file in the main project folder for the data ingestion to work.

git clone <repository-url>
cd OSV-Data-Engineering

Create and activate a virtual environment to manage dependencies:
python3 -m venv osv_env
source osv_env/bin/activate

Install requirements
pip install -r requirements.txt

Find the `dags_folder` setting and update it:
airflow config get-value core dags_folder

nano ~/airflow/airflow.cfg
dags_folder = /mnt/c/Users/../OSV-Data-Engineering/dags

Run the following commands to set up and start Airflow
airflow db init # Initialize the database
airflow scheduler & # Start the scheduler
airflow webserver --port 8080 & # Start the web server on port 8080

Open Airflow UI:

- Go to http://localhost:8080 in your browser.
- Use `airflow dags list` to confirm the DAG is available.
- Use `airflow tasks list` to check available tasks.
- View logs in the Airflow UI to ensure proper execution.
- Click the **Trigger DAG** button.

Limitations:

Due to time constraints, I could not implement a monitoring solution. The data ingestion dag executes python code for downloading and filtering json files for further processing. A spark job that downloads and filters the json files would be more efficient. Since only the Dags were mentioned in the first step, I did not start with Spark. Further, I do not have access to Azure environment, I am not sure what more packaging is required for this project to be plugged.