Report: CI/CD Security Automation Framework with GitHub Actions

1. Executive Summary

This report details the development of a CI/CD Security Automation Framework by Yashvardhan. The framework automates security scanning using Bandit, Semgrep, Trivy, and Grype within a GitHub Actions pipeline. Scan results are parsed and stored in an SQLite database via a custom Python script, enabling historical tracking of vulnerabilities. A Flask-based web dashboard was implemented to visualize the latest scan results for ease of monitoring. Additionally, the system includes automated Discord alerts to notify stakeholders of detected vulnerabilities in real-time. The framework effectively integrates multiple scanning tools with centralized result storage, visualization, and alerting, fulfilling the project objectives.

2. Objective and Scope

2.1 Objective

The goal was to build an automated security testing framework integrated into a CI/CD pipeline to identify vulnerabilities early. This included code and container scanning, storing results, visualization, and alerting through Discord.

- **Automating static code analysis with Bandit and Semgrep:**
  These tools scan source code to detect security issues such as insecure coding practices and potential vulnerabilities before deployment.
- **Performing container image vulnerability scans using Trivy and Grype:**
  Trivy and Grype analyze Docker images for known security flaws and misconfigurations to ensure container security.

- **Storing and tracking scan results in an SQLite database:**
  Scan outputs are parsed and stored in a lightweight database to maintain a historical record of findings for auditing and trend analysis.
- **Visualizing results on a Flask-based dashboard:**

A web interface was created to provide an accessible summary of the latest scan results for quick assessment by stakeholders.

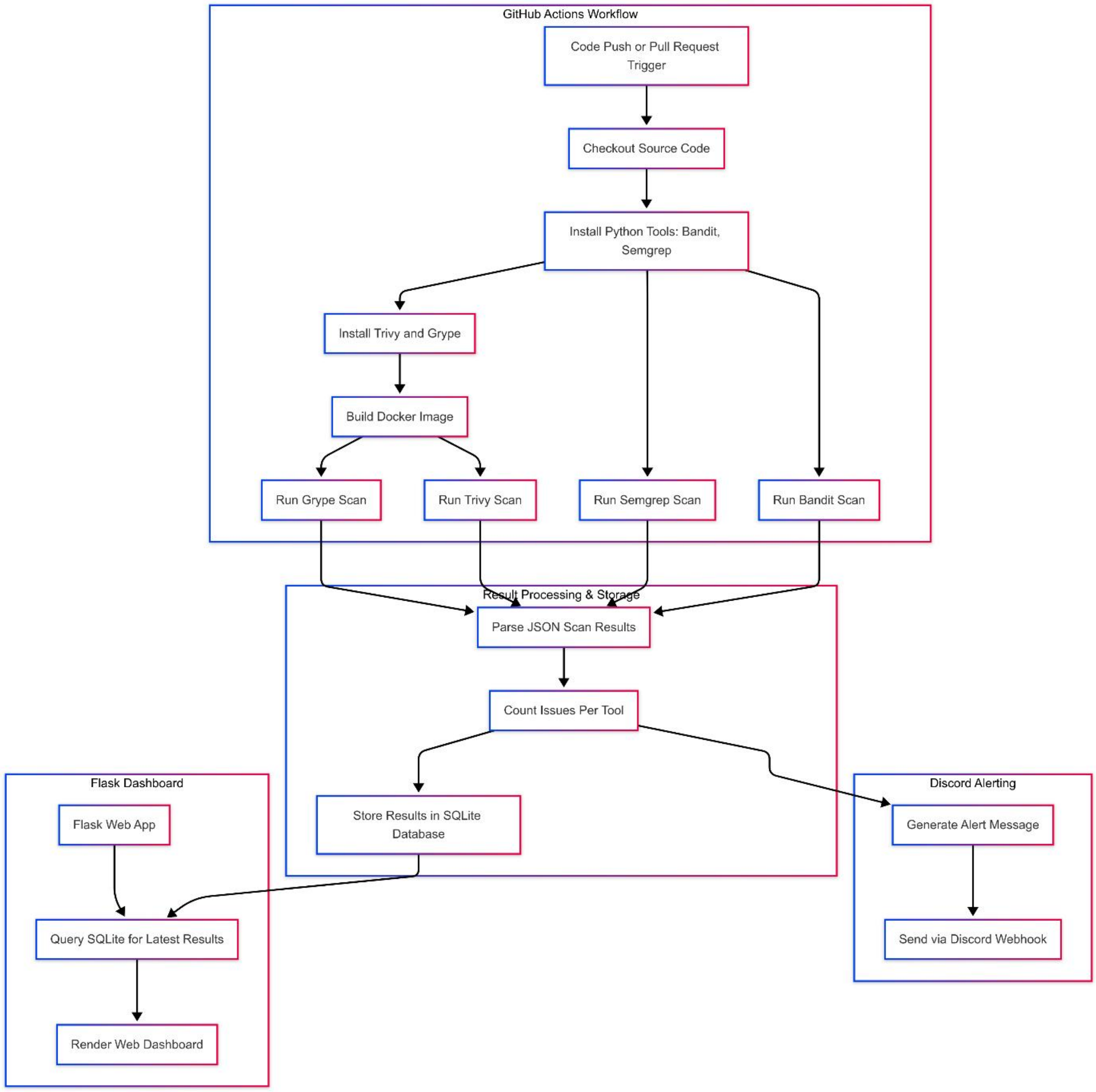
- **Sending automated Discord alerts on detected vulnerabilities:**
  Alerts notify the development and security teams in real-time via Discord whenever vulnerabilities are detected, enabling timely responses.

2.2 Scope of Work

This project covers automating security scans in a CI/CD pipeline, storing and visualizing results, and sending alerts. The focus is on integrating multiple tools to ensure continuous vulnerability detection before deployment.

- **Integration of Bandit and Semgrep for code scanning:**
  Both tools were incorporated into the pipeline to perform comprehensive static code analysis across the project source code.
- **Use of Trivy and Grype for container image scanning:**
  These scanners were configured to inspect built Docker images to identify vulnerabilities before deployment.
- **Implementation of a Python script to parse scan outputs and insert results into SQLite:**
  A dedicated script processes JSON scan results and inserts summarized data into the database for persistence.
- **Development of a Flask dashboard for viewing scan summaries:**
  The dashboard fetches and displays the latest scan data, allowing users to review vulnerabilities via a user-friendly web page.
- **Configuration of Discord webhook alerts for vulnerability notifications:**
  Alerts are automatically sent to a designated Discord channel, ensuring prompt visibility of security issues.
- **Full automation of these processes within a GitHub Actions workflow:**
  The entire sequence—from scanning to reporting and alerting—is automated to run on each push or pull request without manual intervention.

3. System Architecture Diagram



4. Tools and Techniques

- **GitHub Actions:** Utilized to orchestrate automated workflows triggered on code pushes and pull requests. It manages sequential execution of scanning tools, result storage, and notifications.
- **Bandit:** A Python security linter used to perform static analysis on the codebase to identify common security issues.
- **Semgrep:** A static analysis tool configured to scan code for patterns and potential security flaws using automated rules.

- **Trivy:** A vulnerability scanner focused on Docker images, analyzing container layers for known security issues and generating JSON reports.

- **Grype:** Another container image vulnerability scanner that outputs JSON reports, complementing Trivy's coverage.

- **SQLite:** A lightweight relational database used to store scan results persistently. It supports querying latest results and tracks scan metadata such as scan date, tool, and issue counts.

- **Flask:** A minimal Python web framework employed to create a dashboard that retrieves scan data from SQLite and renders it for user-friendly visualization.

- **Discord Webhooks:** Configured to send real-time notifications to a Discord channel whenever vulnerabilities are detected, enabling immediate alerting and response.

5. Methodology and Implementation Details

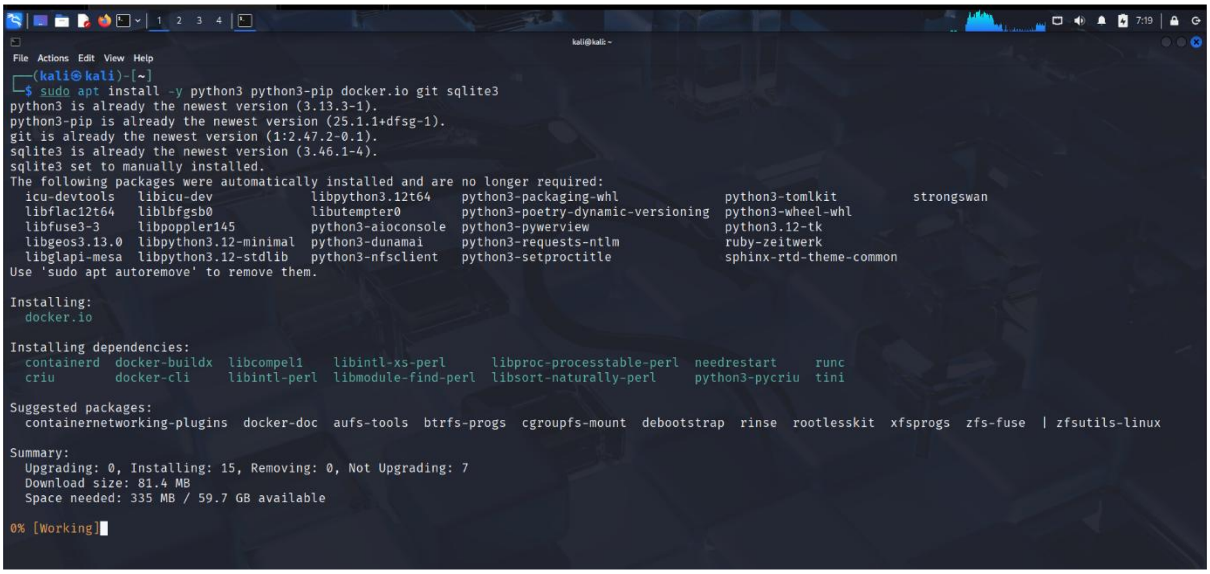
This section describes the step-by-step process of integrating multiple security scanning tools into a CI/CD pipeline using GitHub Actions. It details how static code analysis and container image scanning were automated, scan results collected and stored in a database, and alerts triggered via Discord webhooks. Additionally, it covers the development of a Flask-based dashboard for visualization of the latest scan outcomes.

5.1. Environment Setup

This phase outlines the initial setup required to prepare the development and deployment environment for the CI/CD security automation framework.

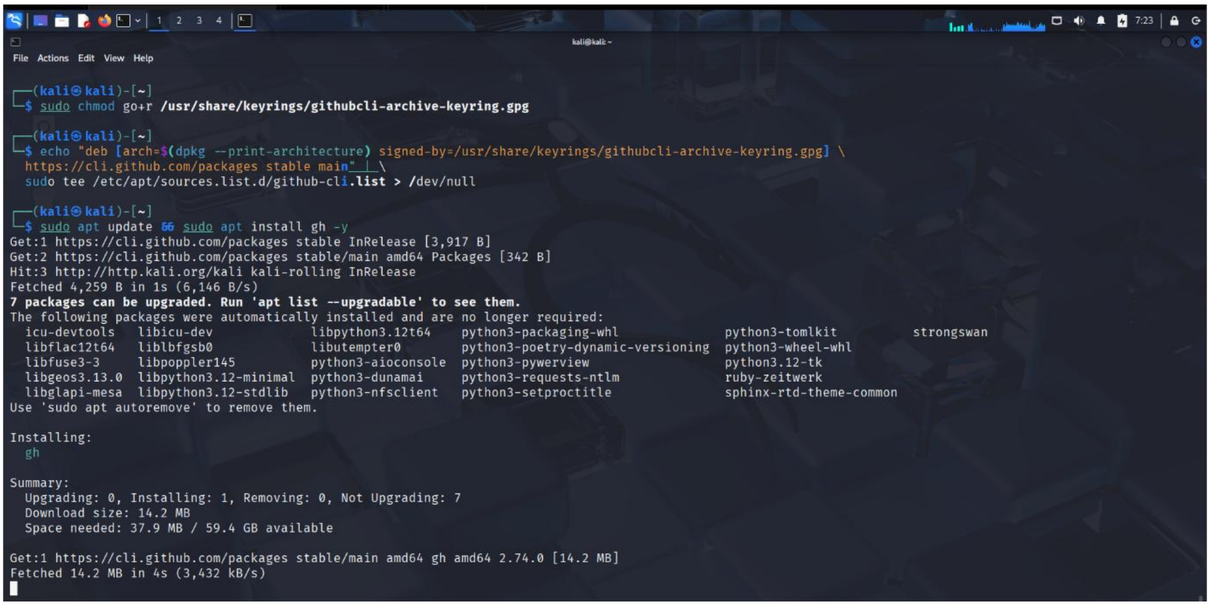
- **Install Required Dependencies**

  Ensure the system has all necessary packages and tools installed, including Python, pip, Docker, and SQLite.



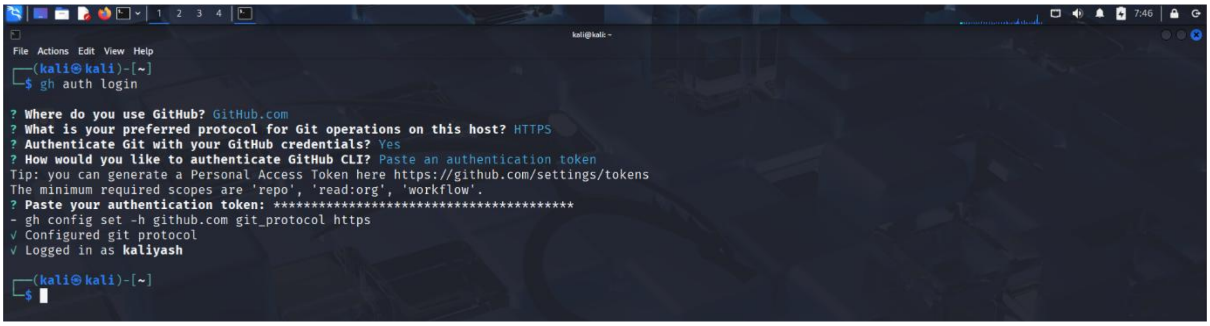
- **Download and Configure Git CLI**

  Install Git CLI to enable version control and repository management.



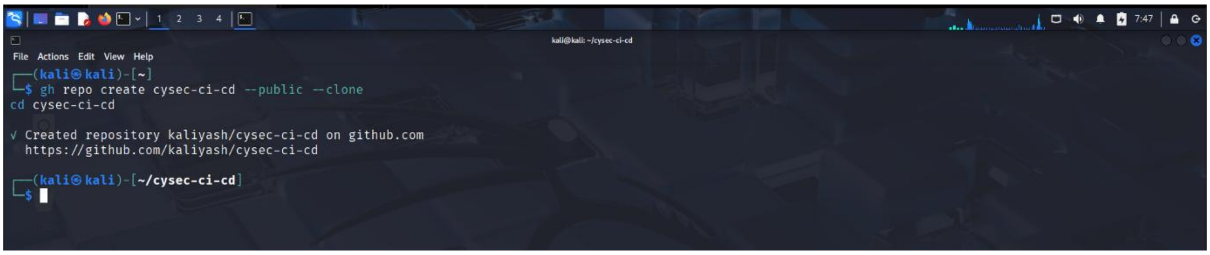
- **Authenticate Git Using Personal Access Token (PAT)**

  Generate a GitHub Personal Access Token (PAT) with appropriate scopes (repo, workflow) and authenticate it for secure Git operations. Configure Git with:



- **Create GitHub Repository**

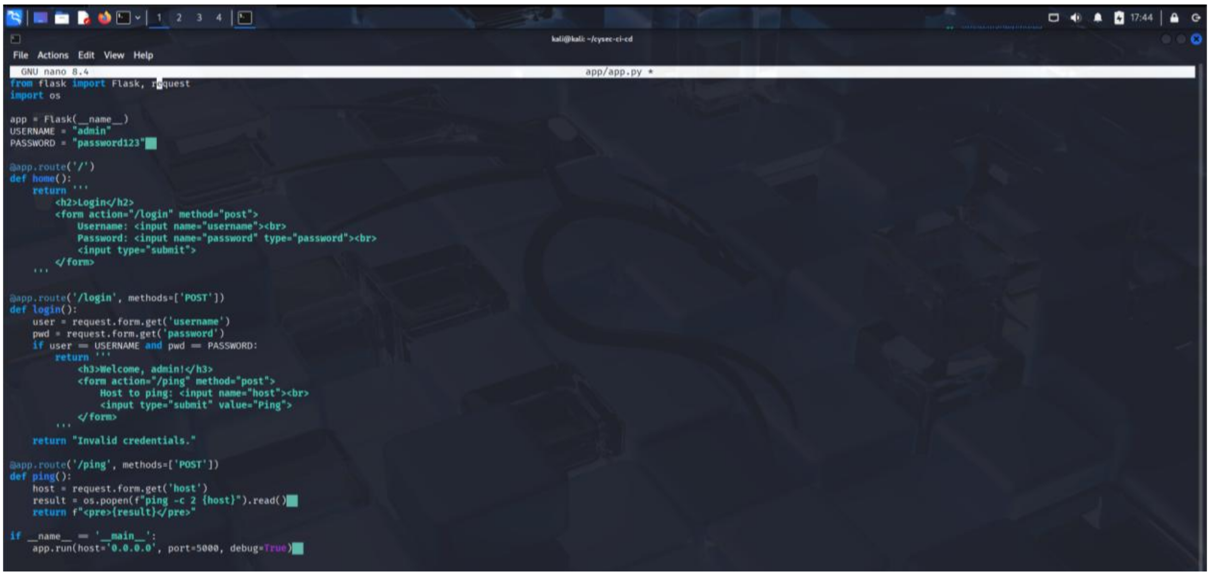
  A public GitHub repository named cysec-ci-cd was created to host the complete source code, configurations, CI workflows, and documentation for the project. This repository serves as the central point for version control and collaboration.



- **Add a vulnerable Dockerfile to the repository**

  Include a vulnerable Python application along with its Dockerfile in the repository. This Docker image will be scanned by **Trivy** and **Grype** during workflow execution.

  Dockerfile used for building the vulnerable application image.



(Source code of the vulnerable Python application included for scanning)

5.2. Pipeline Setup and Security Scanning

Establish an automated continuous integration pipeline that systematically performs static code analysis and container image vulnerability scanning, capturing security findings in structured JSON reports for further processing and alerting.

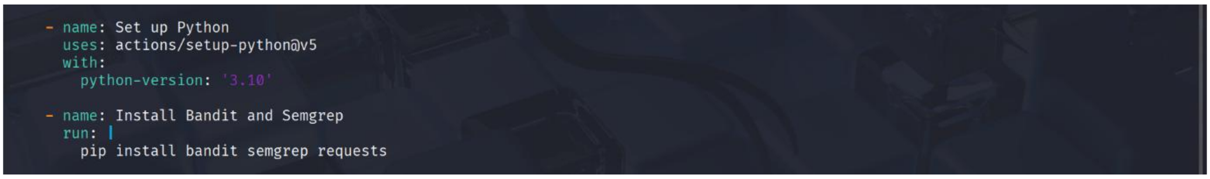
- **Checkout Source Code**

  Fetches the most recent repository code on the GitHub runner to facilitate scanning and build processes.

```
- name: Checkout code
  uses: actions/checkout@v4
```

- **Set Up Python Environment and Install Static Scanning Tools**
  a) Configures the Python 3.10 runtime environment.
  b) Installs Bandit and Semgrep to enable static code analysis.
  c) Integrates the Requests library for facilitating alert notifications.

```
- name: Setup Python and Install Tools
  uses: actions/setup-python@v5
  with:
    python-version: '3.10'

- run: pip install bandit semgrep requests
```



```
- name: Set up Python
  uses: actions/setup-python@v5
  with:
    python-version: '3.10'

- name: Install Bandit and Semgrep
  run: |
    pip install bandit semgrep requests
```

- **Install Container Image Vulnerability Scanners**
  a) Installs Trivy and Grype to perform comprehensive container image vulnerability scanning.
  b) Updates the system PATH environment variable to ensure seamless accessibility of the installed tools.

```
- run: |
    curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh
| sh -s -- -b /usr/local/bin
    curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s -- -
b /usr/local/bin

- run: echo "/usr/local/bin" >> $GITHUB_PATH
```

```
- name: Install Trivy
  run: |
    curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin

- name: Add /usr/local/bin to PATH for Trivy
  run: echo "/usr/local/bin" >> $GITHUB_PATH

- name: Install Grype
  run: |
    curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s -- -b /usr/local/bin

- name: Add /usr/local/bin to PATH for Grype
  run: echo "/usr/local/bin" >> $GITHUB_PATH
```

- **Prepare Directories for Scan Reports**

  Establishes organized directories to systematically store static analysis and container image scan results.

  - name: Create output directories

    run: mkdir -p scans/code scans/image

```
- name: Create output directories
  run: |
    mkdir -p scans/code
    mkdir -p scans/image
```

- **Run Static Code Scans**

  a) Executes Bandit and Semgrep scans recursively on the source code.

  b) Generates JSON reports with fallback to empty results on failure.

  - name: Run Bandit scan

    run: bandit -r . -f json -o scans/code/bandit.json || echo '{}' > scans/code/bandit.json

  - name: Run Semgrep scan

    run: semgrep scan --config=auto --json --output=scans/code/semgrep.json || echo '{"results":[]}' > scans/code/semgrep.json

```
File  Actions  Edit  View  Help
  GNU nano 8.4                         .github/workflows/security_full.yml
    mkdir -p scans/image

- name: Run Bandit scan
  run: |
    bandit -r . -f json -o scans/code/bandit.json || echo '{}' > scans/code/bandit.json
    test -f scans/code/bandit.json || echo '{}' > scans/code/bandit.json

- name: Run Semgrep scan
  run: |
    semgrep scan --config=auto --json --output=scans/code/semgrep.json || echo '{"results":[]}' > scans/code/semgrep.json
    test -f scans/code/semgrep.json || echo '{"results":[]}' > scans/code/semgrep.json
```

- **Build Docker Image**

  Builds the Docker image insecure-app for subsequent vulnerability scanning.

  ```
  - name: Build Docker image
    run: docker build -t insecure-app .
  ```

  ```yaml
  - name: Build Docker image
    run: docker build -t insecure-app .
  ```

- **Run Container Image Scans**
  a) Performs comprehensive vulnerability scanning of the Docker image using Trivy and Grype.
  b) Ensures scan outputs are saved in JSON format with robust error handling to maintain the integrity and availability of result files.

  ```
  - name: Run Trivy scan
    run: trivy image -f json -o scans/image/trivy.json insecure-app || echo '{"Results":[]}' > scans/image/trivy.json

  - name: Run Grype scan
    run: grype insecure-app -o json > scans/image/grype.json || echo '{"matches":[]}' > scans/image/grype.json
  ```

  ```yaml
  - name: Run Trivy scan
    run: |
      trivy image -f json -o scans/image/trivy.json insecure-app || echo '{"Results":[]}' > scans/image/trivy.json
      test -f scans/image/trivy.json || echo '{"Results":[]}' > scans/image/trivy.json

  - name: Run Grype scan
    run: |
      grype insecure-app -o json > scans/image/grype.json || echo '{"matches":[]}' > scans/image/grype.json
      test -f scans/image/grype.json || echo '{"matches":[]}' > scans/image/grype.json
  ```

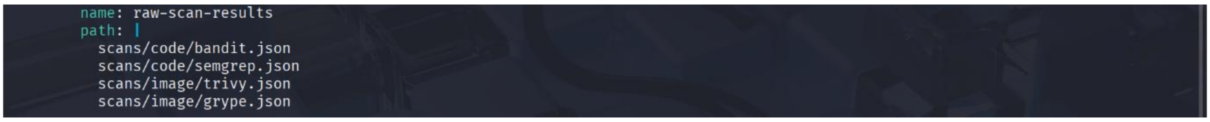
This phase integrates code and container scanning tools to automate vulnerability detection, forming the foundation of a secure CI pipeline.

5.3. Result Collection and Storage

This phase concentrates on processing JSON outputs from multiple security scanning tools. Extracted vulnerability counts are recorded in an SQLite database alongside timestamps, facilitating persistent tracking and historical analysis of scan results.

- **Organize Scan Output Files**

  Scan results from Bandit, Semgrep, Trivy, and Grype are stored as JSON files within organized directory structures:
  - scans/code/bandit.json
  - cans/code/semgrep.json
  - scans/image/trivy.json
  - scans/image/grype.json



- **Load and Parse JSON Files**

  The load_json function reads JSON files from specified paths. It handles missing files, empty contents, and parsing errors by logging warnings and errors accordingly.

```
def load_json(path):
    if os.path.exists(path):
        with open(path, 'r') as f:
            content = f.read().strip()
            if not content:
                print(f"[WARN] {path} is empty.")
                return None
            try:
                return json.loads(content)
            except json.JSONDecodeError:
                print(f"[ERROR] Failed to parse JSON from {path}")
                return None
    print(f"[WARN] {path} does not exist.")
    return None
```

```
GNU nano 8.4                    store_scan_results.py *
import json
import sqlite3
import os
from datetime import datetime, timezone

DB_PATH = os.path.abspath("scan_results.db")

def load_json(path):
    if os.path.exists(path):
        with open(path, 'r') as f:
            content = f.read().strip()
            if not content:
                print(f"[WARN] {path} is empty.")
                return None
            try:
                return json.loads(content)
            except json.JSONDecodeError:
                print(f"[ERROR] Failed to parse JSON from {path}")
                return None
    print(f"[WARN] {path} does not exist.")
    return None
```

This ensures robust loading of scan data without interruption.

- **Count Detected Issues Per Tool**

  Dedicated functions process the loaded JSON data to accurately count the detected issues:

a) The functions count_bandit_issues and count_semgrep_issues tally the number of entries within the "results" list.

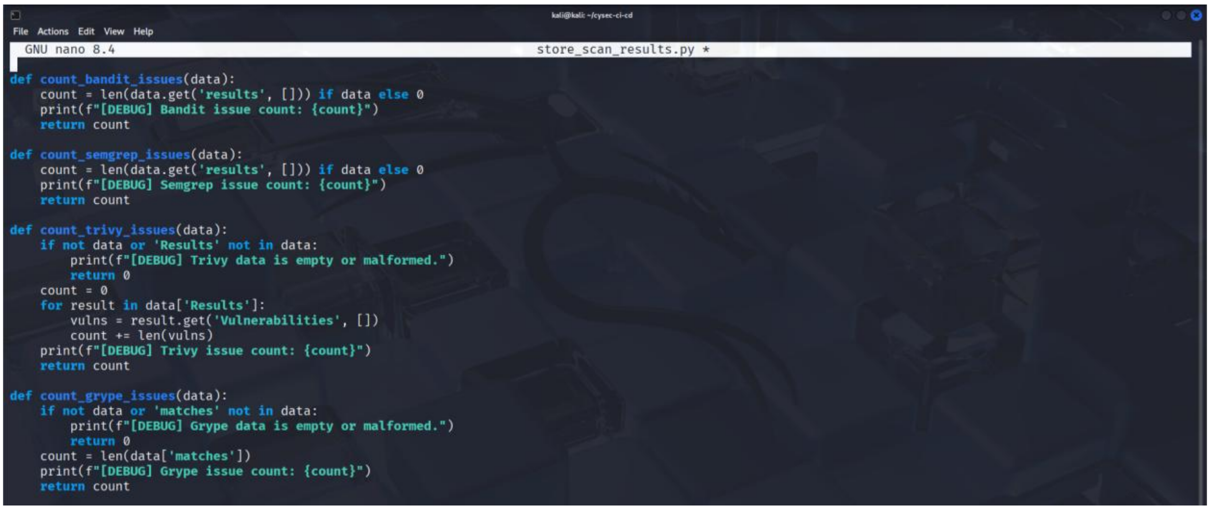
b) The count_trivy_issues function aggregates vulnerabilities found under the "Results" array.

c) The count_grype_issues function counts the entries present under the "matches" key.

Each function includes debug logging to report the respective issue counts.

```
def count_bandit_issues(data):
    count = len(data.get('results', [])) if data else 0
    print(f"[DEBUG] Bandit issue count: {count}")
    return count

def count_semgrep_issues(data):
    count = len(data.get('results', [])) if data else 0
    print(f"[DEBUG] Semgrep issue count: {count}")
    return count
```

```
def count_trivy_issues(data):
    if not data or 'Results' not in data:
        print(f"[DEBUG] Trivy data is empty or malformed.")
        return 0
    count = 0
    for result in data['Results']:
        vulns = result.get('Vulnerabilities', [])
        count += len(vulns)
    print(f"[DEBUG] Trivy issue count: {count}")
    return count

def count_grype_issues(data):
    if not data or 'matches' not in data:
        print(f"[DEBUG] Grype data is empty or malformed.")
        return 0
    count = len(data['matches'])
    print(f"[DEBUG] Grype issue count: {count}")
    return count
```

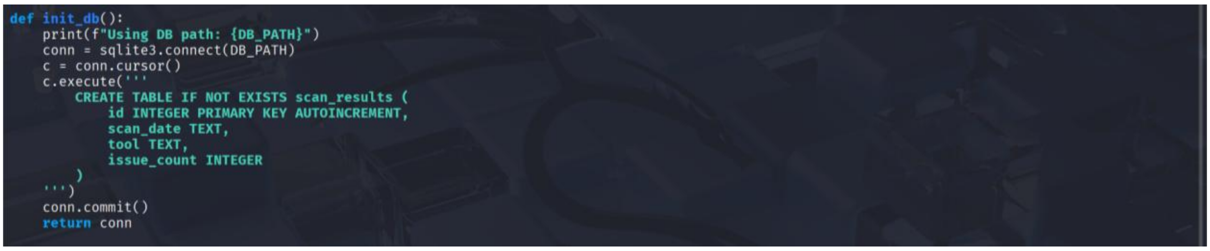


This modular approach allows accurate issue extraction per tool.

- **Initialize SQLite Database**

  The init_db function establishes a connection to the SQLite database, creating it if necessary. It also ensures the presence of the scan_results table, which includes columns for ID, scan timestamp, tool name, and issue count.

```
def init_db():
    print(f"Using DB path: {DB_PATH}")
    conn = sqlite3.connect(DB_PATH)
    c = conn.cursor()
    c.execute('''
        CREATE TABLE IF NOT EXISTS scan_results (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            scan_date TEXT,
            tool TEXT,
            issue_count INTEGER
        )
    ''')
    conn.commit()
    return conn
```



```
def init_db():
    print(f"Using DB path: {DB_PATH}")
    conn = sqlite3.connect(DB_PATH)
    c = conn.cursor()
    c.execute('''
        CREATE TABLE IF NOT EXISTS scan_results (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            scan_date TEXT,
            tool TEXT,
            issue_count INTEGER
        )
    ''')
    conn.commit()
    return conn
```

This establishes a reliable and persistent storage mechanism for scan data.

- **Insert Scan Results into Database**

  The insert_result function records the tool name, current UTC timestamp, and detected issue count into the database. It also logs detailed information about each insertion for traceability.

```
def insert_result(conn, tool, count):
    print(f"[INFO] Inserting result: {tool} = {count}")
    c = conn.cursor()
```

```
    c.execute('INSERT INTO scan_results (scan_date, tool, issue_count) VALUES (?, ?, ?)',
              (datetime.now(timezone.utc).isoformat(), tool, count))
    conn.commit()
```

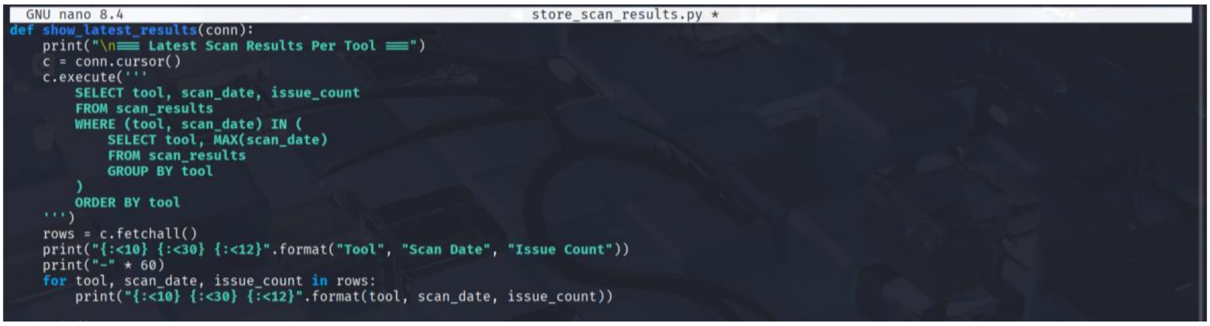
```
def insert_result(conn, tool, count):
    print(f"[INFO] Inserting result: {tool} = {count}")
    c = conn.cursor()
    c.execute('INSERT INTO scan_results (scan_date, tool, issue_count) VALUES (?, ?, ?)',
              (datetime.now(timezone.utc).isoformat(), tool, count))
    conn.commit()
```

This enables historical tracking of vulnerabilities for each scan, supporting trend analysis and auditability over time.

- **Display Latest Scan Results**

  The show_latest_results function queries the database to retrieve the most recent scan results for each tool, sorted alphabetically by tool name. It provides a clear and immediate summary of the latest security findings in the console for quick review.

```
def show_latest_results(conn):
    print("\n=== Latest Scan Results Per Tool ===")
    c = conn.cursor()
    c.execute('''
        SELECT tool, scan_date, issue_count
        FROM scan_results
        WHERE (tool, scan_date) IN (
            SELECT tool, MAX(scan_date)
            FROM scan_results
            GROUP BY tool
        )
        ORDER BY tool
    ''')
    rows = c.fetchall()
    print("{:<10} {:<30} {:<12}".format("Tool", "Scan Date", "Issue Count"))
    print("-" * 60)
    for tool, scan_date, issue_count in rows:
        print("{:<10} {:<30} {:<12}".format(tool, scan_date, issue_count))
```



```
GNU nano 8.4                                   store_scan_results.py *
def show_latest_results(conn):
    print("\n=== Latest Scan Results Per Tool ===")
    c = conn.cursor()
    c.execute('''
        SELECT tool, scan_date, issue_count
        FROM scan_results
        WHERE (tool, scan_date) IN (
            SELECT tool, MAX(scan_date)
            FROM scan_results
            GROUP BY tool
        )
        ORDER BY tool
    ''')
    rows = c.fetchall()
    print("{:<10} {:<30} {:<12}".format("Tool", "Scan Date", "Issue Count"))
    print("-" * 60)
    for tool, scan_date, issue_count in rows:
        print("{:<10} {:<30} {:<12}".format(tool, scan_date, issue_count))
```

This facilitates rapid verification of stored scan data, ensuring timely access to the most recent security insights.

- **Main Execution Flow**

  The main function orchestrates the end-to-end workflow by managing the loading of JSON scan results, initializing the database, inserting issue counts for each tool, displaying the latest results, and closing the database connection to ensure proper resource management.

```
def main():
    bandit_data = load_json("scans/code/bandit.json")
    semgrep_data = load_json("scans/code/semgrep.json")
    trivy_data = load_json("scans/image/trivy.json")
    grype_data = load_json("scans/image/grype.json")

    conn = init_db()

    insert_result(conn, "bandit", count_bandit_issues(bandit_data))
    insert_result(conn, "semgrep", count_semgrep_issues(semgrep_data))
    insert_result(conn, "trivy", count_trivy_issues(trivy_data))
    insert_result(conn, "grype", count_grype_issues(grype_data))

    show_latest_results(conn)

    conn.close()
    print("[SUCCESS] Scan results stored and latest results displayed.")
```

```
def main():
    bandit_data = load_json("scans/code/bandit.json")
    semgrep_data = load_json("scans/code/semgrep.json")
    trivy_data = load_json("scans/image/trivy.json")
    grype_data = load_json("scans/image/grype.json")

    conn = init_db()

    insert_result(conn, "bandit", count_bandit_issues(bandit_data))
    insert_result(conn, "semgrep", count_semgrep_issues(semgrep_data))
    insert_result(conn, "trivy", count_trivy_issues(trivy_data))
    insert_result(conn, "grype", count_grype_issues(grype_data))

    show_latest_results(conn)

    conn.close()
    print("[SUCCESS] Scan results stored and latest results displayed.")

if __name__ == "__main__":
    main()
```

5.4. Alerting Mechanism

This phase establishes an automated alerting mechanism that parses scan results and sends summarized vulnerability counts to a Discord channel using a webhook. It ensures real-time visibility of detected issues.

- **Load Scan JSON Files**

  The script loads JSON scan results for Bandit, Semgrep, Trivy, and Grype from their respective files using a robust load_json function, which gracefully handles missing files, empty contents, and JSON decoding errors.

```
def load_json(path):
    if os.path.exists(path):
        try:
            with open(path, 'r') as f:
                content = f.read().strip()
                if not content:
                    return None
                return json.loads(content)
        except json.JSONDecodeError:
            print(f"Warning: JSON decode error in {path}")
            return None
    return None

bandit_data = load_json("scans/code/bandit.json")
semgrep_data = load_json("scans/code/semgrep.json")
```

```
trivy_data = load_json("scans/image/trivy.json")
grype_data = load_json("scans/image/grype.json")
```

```
def load_json(path):
    if os.path.exists(path):
        try:
            with open(path, 'r') as f:
                content = f.read().strip()
                if not content:
                    return None
                return json.loads(content)
        except json.JSONDecodeError:
            print(f"Warning: JSON decode error in {path}")
            return None
    return None
```

```
bandit_data = load_json("scans/code/bandit.json")
semgrep_data = load_json("scans/code/semgrep.json")
trivy_data = load_json("scans/image/trivy.json")
grype_data = load_json("scans/image/grype.json")
```

- **Count Vulnerabilities**

  This step defines functions to count vulnerabilities reported by each scanning tool. The functions safely handle missing or malformed data and return the number of issues detected.

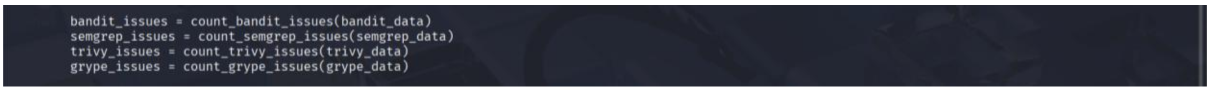
  Tool-specific functions parse the loaded JSON data to accurately count detected issues:

  - Bandit and Semgrep tally entries within the 'results' key.
  - Trivy counts vulnerabilities listed under the 'Results' array, specifically within the 'Vulnerabilities' field.
  - Grype counts entries found in the 'matches' list.

```
def count_bandit_issues(data):
    if not data or 'results' not in data:
        return 0
    return len(data['results'])

def count_semgrep_issues(data):
    if not data or 'results' not in data:
        return 0
    return len(data['results'])

def count_trivy_issues(data):
    if not data or 'Results' not in data:
```

```
        return 0
    count = 0
    for result in data['Results']:
        vulnerabilities = result.get('Vulnerabilities')
        if vulnerabilities:
            count += len(vulnerabilities)
    return count

def count_grype_issues(data):
    if not data or 'matches' not in data:
        return 0
    return len(data['matches'])

bandit_issues = count_bandit_issues(bandit_data)
semgrep_issues = count_semgrep_issues(semgrep_data)
trivy_issues = count_trivy_issues(trivy_data)
grype_issues = count_grype_issues(grype_data)
```

```
GNU nano 8.4                    .github/workflows/security_full.yml
def count_bandit_issues(data):
    if not data or 'results' not in data:
        return 0
    return len(data['results'])

def count_semgrep_issues(data):
    if not data or 'results' not in data:
        return 0
    return len(data['results'])

def count_trivy_issues(data):
    if not data or 'Results' not in data:
        return 0
    count = 0
    for result in data['Results']:
        vulnerabilities = result.get('Vulnerabilities')
        if vulnerabilities:
            count += len(vulnerabilities)
    return count

def count_grype_issues(data):
    if not data or 'matches' not in data:
        return 0
    return len(data['matches'])
```



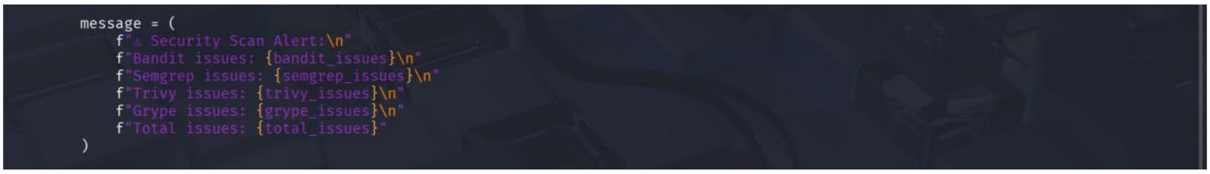
```
bandit_issues = count_bandit_issues(bandit_data)
semgrep_issues = count_semgrep_issues(semgrep_data)
trivy_issues = count_trivy_issues(trivy_data)
grype_issues = count_grype_issues(grype_data)
```

- **Compose Alert Message**

  Combine tool-wise issue counts into a single message:

```
message = (
    f"⚠ Security Scan Alert:\n"
    f"Bandit issues: {bandit_issues}\n"
    f"Semgrep issues: {semgrep_issues}\n"
```

```
    f"Trivy issues: {trivy_issues}\n"
    f"Grype issues: {grype_issues}\n"
    f"Total issues: {total_issues}"
)
```



```
message = (
    f"⚠ Security Scan Alert:\n"
    f"Bandit issues: {bandit_issues}\n"
    f"Semgrep issues: {semgrep_issues}\n"
    f"Trivy issues: {trivy_issues}\n"
    f"Grype issues: {grype_issues}\n"
    f"Total issues: {total_issues}"
)
```

- **Send Discord Notification**

  The code retrieves the Discord webhook URL from environment variables, verifies its availability, constructs a JSON payload with the alert message, and sends it using an HTTP POST request. It then evaluates the response: a 204 status code confirms successful delivery, while any other status logs an error and terminates the script.

  Storing Sensitive URL as GitHub Actions Secret.

```
DISCORD_WEBHOOK_URL=
https://discord.com/api/webhooks/1379471892611596490/mac0CgUcaZ5hevREgGo57
7djNRC3BIqTaVJXHgrEWRfR6E8CNIW1U_T2HFC52A8hRlIL
```

```
webhook_url = os.getenv("DISCORD_WEBHOOK_URL")
if not webhook_url:
    print("No Discord webhook URL set. Exiting.")
    sys.exit(1)
payload = {"content": message}
response = requests.post(webhook_url, json=payload)

if response.status_code == 204:
    print("Discord alert sent successfully.")
else:
    print(f"Failed to send Discord alert: {response.status_code}, {response.text}")
    sys.exit(1)
```

```
webhook_url = os.getenv("DISCORD_WEBHOOK_URL")
if not webhook_url:
    print("No Discord webhook URL set. Exiting.")
    sys.exit(1)

payload = {"content": message}
response = requests.post(webhook_url, json=payload)

if response.status_code == 204:
    print("Discord alert sent successfully.")
else:
    print(f"Failed to send Discord alert: {response.status_code}, {response.text}")
    sys.exit(1)
EOF
```

5.5. Visualization Dashboard

This phase focuses on developing a web-based dashboard using Flask to visualize the latest security scan results stored in the SQLite database. The dashboard offers a clear and organized interface to review scan dates, tools utilized, and issue counts, thereby improving accessibility and facilitating effective monitoring of security findings.

- **Flask Application Setup**

  The Flask framework is used to create a web application for visualizing security scan results. The database path is set using an absolute path to avoid path resolution issues.

```
from flask import Flask, render_template
import sqlite3
import os

app = Flask(__name__)
DB_PATH = os.path.abspath("scan_results.db")
```

```
kali@kali: ~/cysec-ci-cd
File  Actions  Edit  View  Help
  GNU nano 8.4                         dashboard.py *
from flask import Flask, render_template
import sqlite3
import os

app = Flask(__name__)
DB_PATH = os.path.abspath("scan_results.db")
```

- **Database Query Function**

  The get_scan_results() function connects to the SQLite database, retrieves the latest scan results grouped by tool, and returns this data for rendering. The query selects the tool name, most recent scan date, and corresponding issue count, ordering the results by descending scan date to prioritize recent findings.

```
def get_scan_results():
    print(f"Opening DB at: {DB_PATH}")
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    cursor.execute("""
        SELECT tool, MAX(scan_date) as latest_scan, issue_count
        FROM scan_results
        GROUP BY tool
        ORDER BY latest_scan DESC
    """)
    results = cursor.fetchall()
    print(f"Fetched {len(results)} latest records from DB")
    conn.close()
    return results
```

```
def get_scan_results():
    print(f"Opening DB at: {DB_PATH}")
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    cursor.execute("""
        SELECT tool, MAX(scan_date) as latest_scan, issue_count
        FROM scan_results
        GROUP BY tool
        ORDER BY latest_scan DESC
    """)
    results = cursor.fetchall()
    print(f"Fetched {len(results)} latest records from DB")
    conn.close()
    return results
```


- **Define Route and Render Template**

  The root route / invokes the get_scan_results() function to fetch the latest scan data, which is then passed to the dashboard.html template for rendering. Debug print statements log the number of records supplied to the template, facilitating verification and troubleshooting.

```
@app.route('/')
def index():
    results = get_scan_results()
    print(f"Passing {len(results)} results to template")
```

```
    return render_template('dashboard.html', results=results)
```

```
@app.route('/')
def index():
    results = get_scan_results()
    print(f"Passing {len(results)} results to template")
    return render_template('dashboard.html', results=results)
```

- **HTML Template for Display**

  The dashboard.html template formats the scan results into a clean, centered table with columns for Scan Date, Tool, and Issue Count. Jinja2 templating syntax iterates over the passed results to populate rows dynamically.

```
GNU nano 8.4                                   templates/dashboard.html
<!DOCTYPE html>
<html>
<head>
    <title>Security Scan Dashboard</title>
    <style>
        body { font-family: Arial, sans-serif; }
        table { border-collapse: collapse; width: 80%; margin: 20px auto; }
        th, td { border: 1px solid #ccc; padding: 8px; text-align: center; }
        th { background-color: #f4f4f4; }
    </style>
</head>
<body>
    <h2 style="text-align:center;">Security Scan Results</h2>
    <table>
        <thead>
            <tr>
                <th>Scan Date</th>
                <th>Tool</th>
                <th>Issue Count</th>
            </tr>
        </thead>
        <tbody>
            {% for scan_date, tool, issue_count in results %}
            <tr>
                <td>{{ scan_date }}</td>
                <td>{{ tool }}</td>
                <td>{{ issue_count }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</body>
</html>
```

- **Run the Flask Server**

  The Flask app is configured to run on all network interfaces (0.0.0.0) at port 5000 with debug mode enabled for easier troubleshooting during development.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

This phase delivers a web-based dashboard providing an organized and accessible visualization of the latest scan results stored in the SQLite database, facilitating efficient security monitoring.

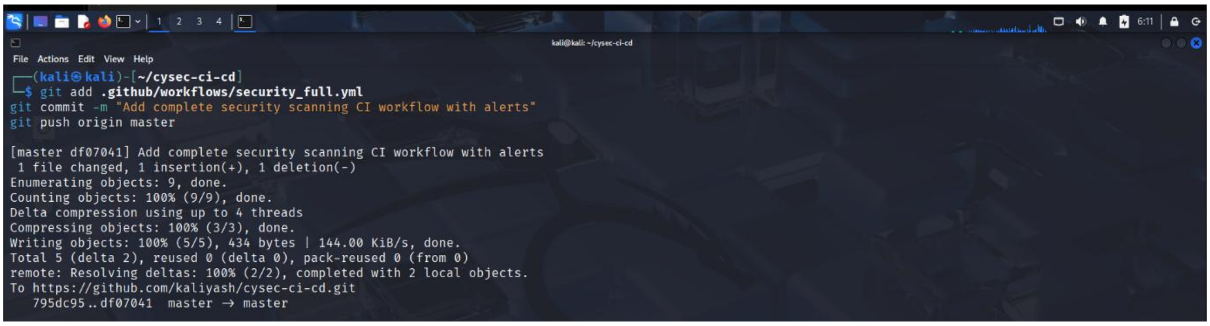
6. Test and Validation

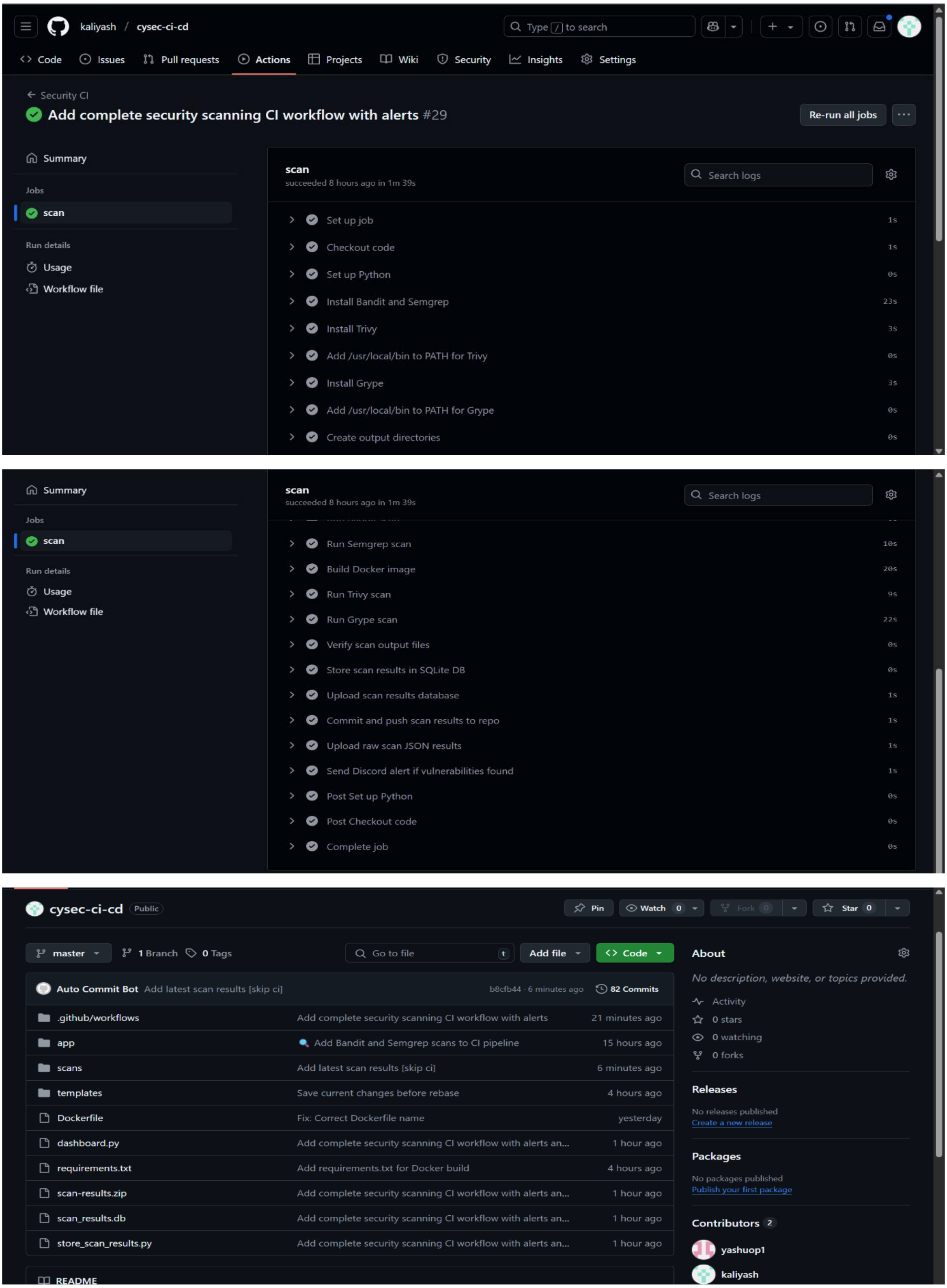
This phase is dedicated to validating the full functionality of the security CI/CD framework. It encompasses committing the CI workflow, processing and storing scan results persistently, and verifying real-time visualization via the dashboard. The primary objective is to ensure the accuracy, reliability, and seamless integration of automated scanning, result management, alerting, and reporting processes.

- **Commit CI Workflow to Repository**

  This step guarantees that the entire security scanning workflow, encompassing all scanning tools and alert configurations, is committed and pushed to the Git repository. This enables automated scans to be triggered upon code changes.

  git add .github/workflows/security_full.yml
  git commit -m "Add complete security scanning CI workflow with alerts"
  git push origin master

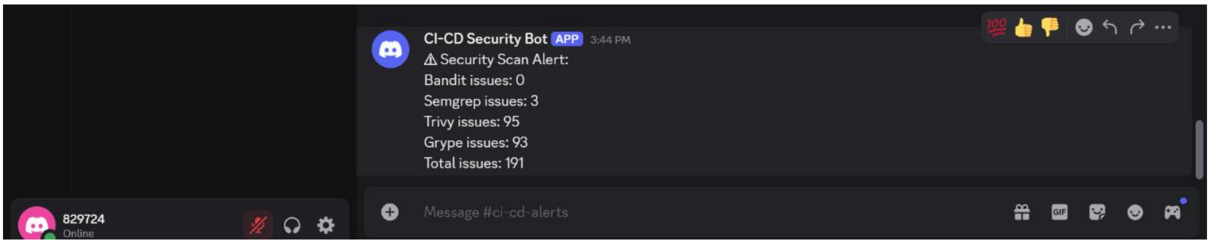




Successful commit and push of the GitHub Actions workflow to the cysec-ci-cd repository.

- **Trigger Discord Alert on Vulnerabilities**

The workflow dynamically counts total issues found by each scanning tool and sends a structured alert to a Discord webhook if any vulnerabilities are detected. The alert logic is as follows:

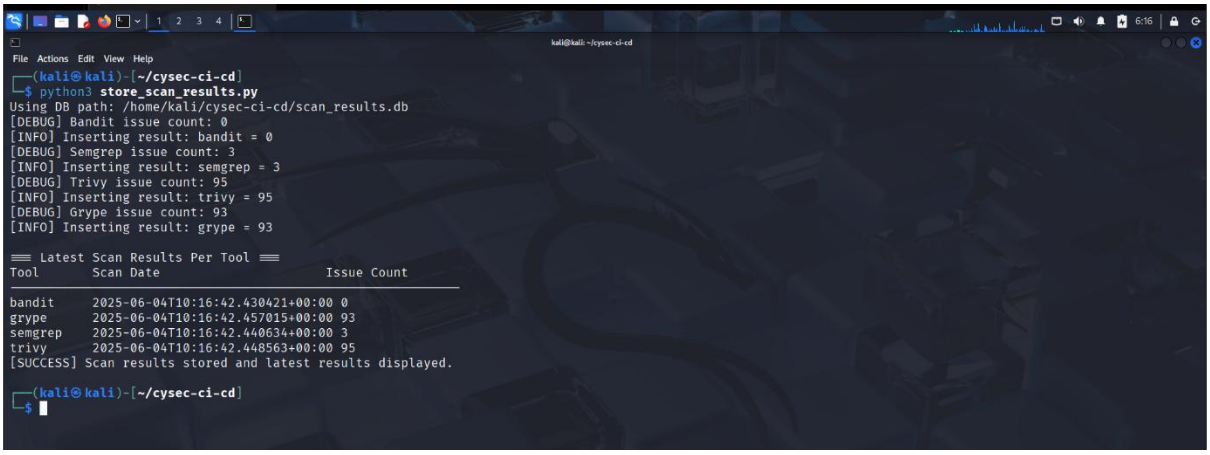


Confirmation of a Discord webhook message sent upon detection of security issues from scan outputs.

- **Process Scan Results and Store in Database**

  The script processes JSON output files from multiple security tools, enumerates detected vulnerabilities, and inserts these results into the SQLite database along with timestamps. This persistent storage facilitates ongoing tracking and historical analysis of scan data.

  python store_scan_results.py



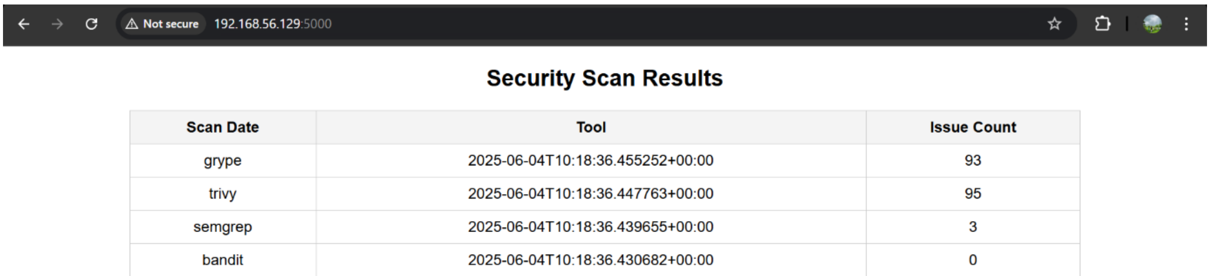
Terminal output confirming the extraction and storage of scan results into the SQLite database.

- **Launch Visualization Dashboard**

This launches the Flask web application, which queries the SQLite database for the most recent scan results and presents them in a tabular format on the dashboard. It offers an accessible, real-time interface for monitoring security scan outcomes.

python dashboard.py

Running dashboard.py to start the web server and visualize security scan results.



Scan Date	Tool	Issue Count
grype	2025-06-04T10:18:36.455252+00:00	93
trivy	2025-06-04T10:18:36.447763+00:00	95
semgrep	2025-06-04T10:18:36.439655+00:00	3
bandit	2025-06-04T10:18:36.430682+00:00	0

The live Flask-rendered dashboard showing the latest security scan results per tool, including scan date, tool name, and detected issue count.

7. Deliverables Review

The project culminated in the delivery of the following key artifacts, meticulously curated to ensure comprehensive understanding and reproducibility:

- **Public GitHub Repository:**
  A centralized and publicly accessible repository hosting the complete set of source code scripts, configuration files, and detailed documentation. This repository serves as the authoritative reference for all project components, facilitating transparency and ease of collaboration or future enhancements.

- **Technical README Documentation:**
  An extensive README file accompanies the repository, providing clear and structured explanations of the system architecture. It includes detailed diagrams that visually represent the design and data flow, alongside annotated screenshots that demonstrate key functionalities and user interfaces. This documentation is intended to guide users through deployment, usage, and maintenance.

- **Supplementary Knowledge Resources (Optional):**
  To further aid knowledge transfer, an optional recorded walkthrough or a technical blog post has been prepared. These resources summarize the project's objectives, implementation strategies, and results, offering a concise yet thorough overview suitable for presentations or educational purposes.

8. Results and Findings

This section presents the key outcomes and insights obtained through the implementation and testing phases of the project:

- **Comprehensive Security Scan Integration:**
  The system successfully integrated multiple security scanning tools (Bandit, Semgrep, Trivy, and Grype), automating their execution within a CI/CD pipeline. JSON outputs from these tools were consistently collected, parsed, and stored in a structured SQLite database, enabling persistent tracking of vulnerabilities over time.

- **Accurate Vulnerability Quantification:**

The implemented parsing scripts reliably extracted vulnerability counts from diverse JSON formats. These counts were accurately recorded with corresponding timestamps, supporting historical analysis and trend identification in scan results.

- **Real-time Alerting:**
  The alert mechanism effectively delivered vulnerability summaries to a designated Discord channel via webhooks, providing timely visibility of security issues and enabling rapid response.

- **User-friendly Visualization Dashboard:**
  The Flask-based web dashboard provided a clear and concise interface to view the latest scan results. It facilitated easy monitoring of vulnerabilities per tool, enhancing situational awareness for security teams.

- **Robust Testing and Validation:**
  End-to-end testing, including code commits, scan execution, data storage, alerting, and dashboard display, demonstrated the system's reliability and functional correctness.

9. Recommendations

Based on the implementation, testing, and findings of this project, the following professional recommendations are proposed to enhance the system further:

- **Expand Tool Coverage:**
  Integrate additional static and dynamic analysis tools such as SonarQube, OWASP ZAP, or Checkmarx to broaden vulnerability detection across application layers.

- **Severity-Based Alerting:**
  Implement logic to classify vulnerabilities by severity (e.g., Critical, High, Medium, Low) and send alerts accordingly. This prioritizes incident response and reduces alert fatigue.
- **Automated Remediation Suggestions:**

Extend the parsing engine to extract fix recommendations from tool outputs where available, and include them in Discord alerts or the dashboard interface.

- **Historical Trend Analysis:**
  Incorporate visual analytics and trend graphs in the dashboard to observe the evolution of vulnerabilities over multiple scan cycles.

- **Access Control for Dashboard:**
  Add user authentication and role-based access controls to secure the dashboard and restrict access to authorized personnel only.

- **Pipeline Integration Testing:**
  Add automated tests within the CI/CD pipeline to validate not only code security but also the integrity of the pipeline and alert mechanisms themselves.

10. Conclusion

This project successfully delivered a comprehensive cybersecurity automation framework that integrates multiple security scanning tools into a continuous integration and delivery (CI/CD) pipeline. By automating the detection, reporting, and alerting of code and container vulnerabilities, the system enhances early threat identification and fosters a secure development lifecycle. The inclusion of a real-time dashboard and Discord alerting system ensures visibility and prompt response to security issues. Overall, this solution demonstrates an effective, scalable, and practical approach to embedding security into modern DevOps workflows.

Appendix

GitHub and Blog Links

- GitHub Repository: Repository Link
- Blog: Link