Symmetric Searchable Encryption (SSE) Project Report

General Description

The SSE web application enables secure storage and search over encrypted textual files. Users can upload plain text files via a simple web interface. Upon upload, the file is encrypted using AES encryption, indexed by a file hash, and stored in an SQLite database. Users can then perform keyword searches; the application decrypts stored files on the fly, matches the search term, and returns filenames of matching documents. A "Clear Database" button allows resetting the stored data. This project aims to demonstrate searchable encryption while maintaining confidentiality.

User Interface Overview:

- File Upload: Allows users to upload .txt files.
- Search Bar: Users can enter keywords to search over encrypted content.
- Search Results: Displays matching file names.
- Clear Button: Wipes all database entries for testing/demo purposes.

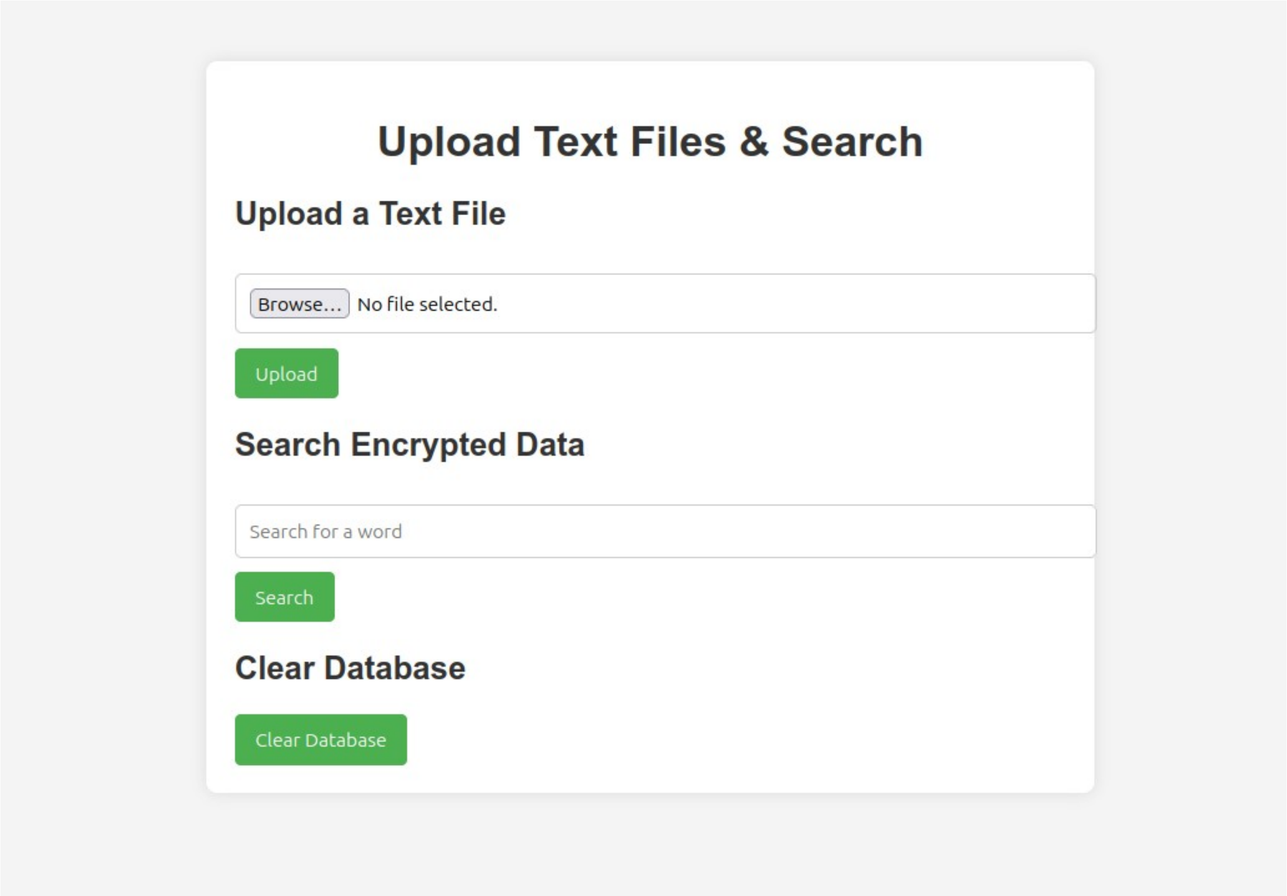
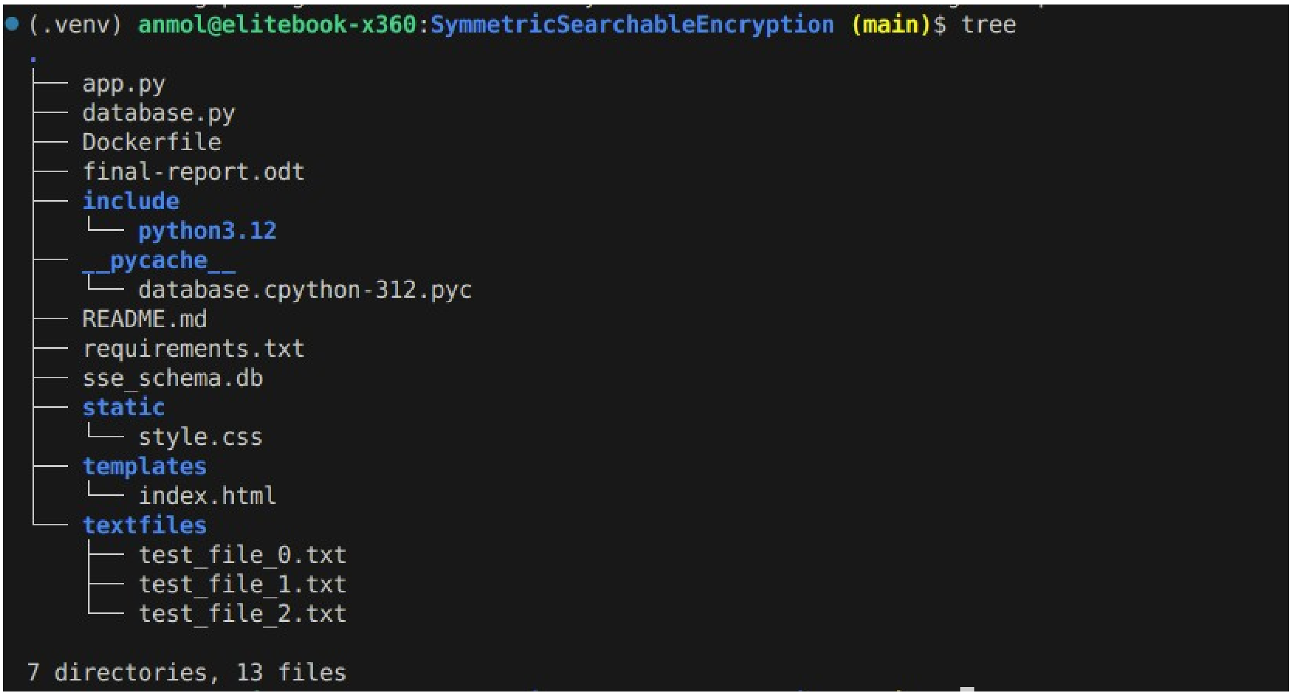


Image 1: Landing webpage

Structure of the program

- app.py: Main Flask app handling HTTP routes, search logic, and file processing.
- database.py: Handles encryption (AES), decryption, hashing, and database interactions.
- Dockerfile: Containerizes the application using a minimal Ubuntu base.
- README.md: GitHub documentation.
- requirements.txt: Lists Python dependencies.
- sse_schema.db: SQLite database.
- static/: Contains basic CSS style sheet
- templates/: Contains HTML templates rendered by Flask.
- textfiles/: Contains textfiles uploaded to database.

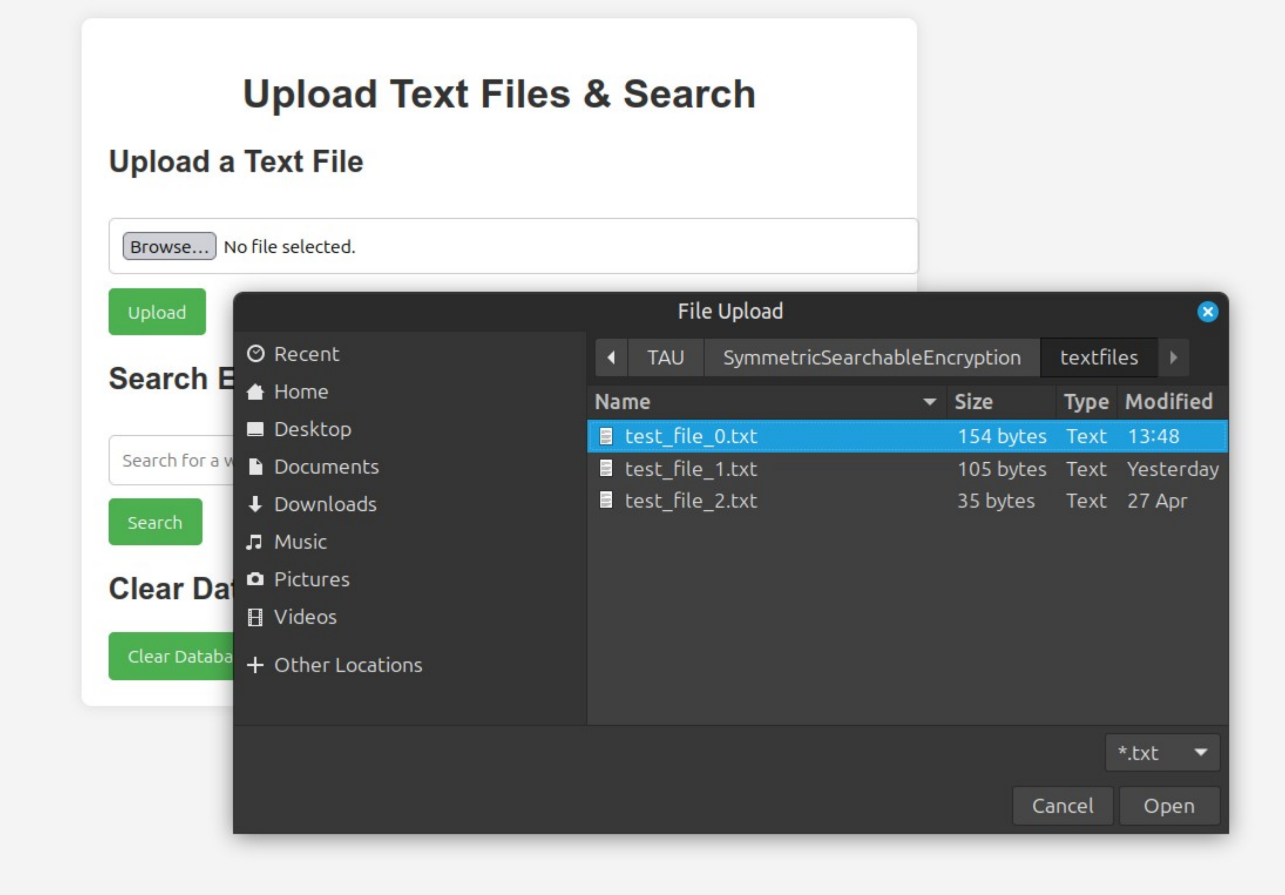


```
● (.venv) anmol@elitebook-x360:SymmetricSearchableEncryption (main)$ tree
 .
 ├── app.py
 ├── database.py
 ├── Dockerfile
 ├── final-report.odt
 ├── include
 │   └── python3.12
 ├── __pycache__
 │   └── database.cpython-312.pyc
 ├── README.md
 ├── requirements.txt
 ├── sse_schema.db
 ├── static
 │   └── style.css
 ├── templates
 │   └── index.html
 └── textfiles
     ├── test_file_0.txt
     ├── test_file_1.txt
     └── test_file_2.txt

 7 directories, 13 files
```

Image 2: Project structure

Working demonstration

1. Upload text file to database

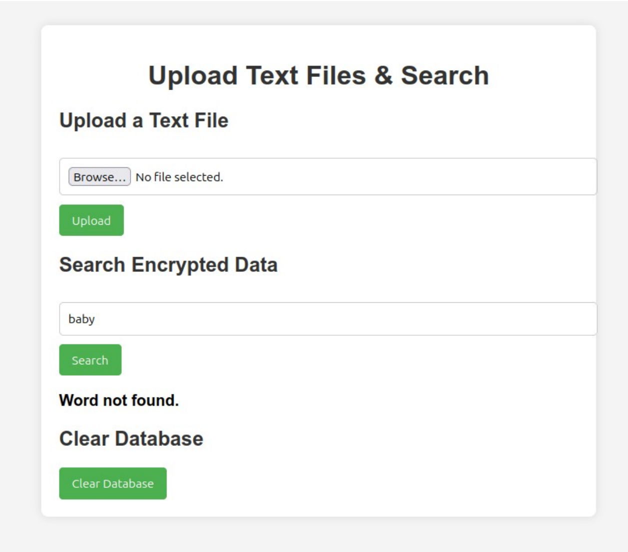


Name	Size	Type	Modified
test_file_0.txt	154 bytes	Text	13:48
test_file_1.txt	105 bytes	Text	Yesterday
test_file_2.txt	35 bytes	Text	27 Apr

Image 3: Uploading image

2. Click upload

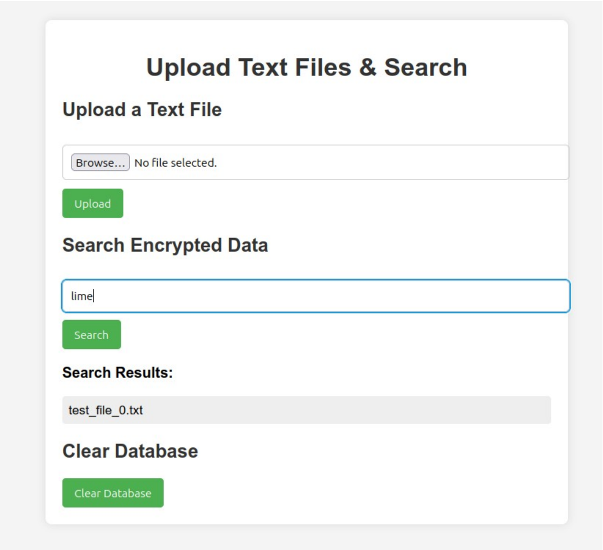
3. Search in encrypted data by words directly



baby

Word not found.

Image 4: Word not found



lime

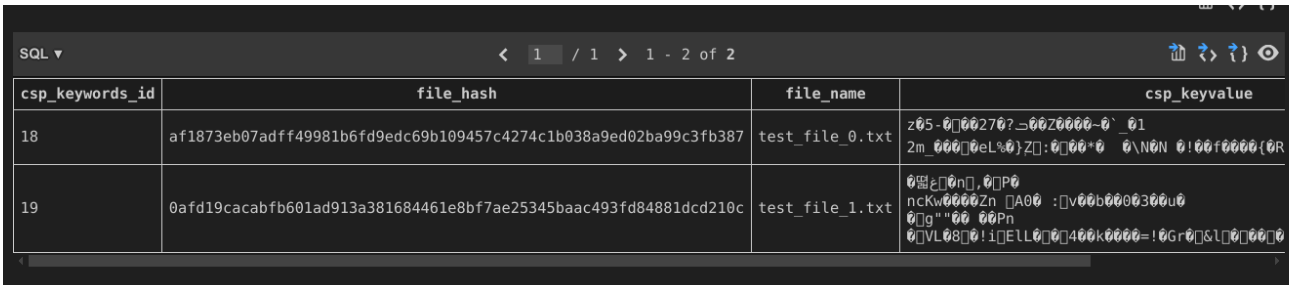
Search Results:

test_file_0.txt

Image 5: Word found

4. Clear database to reset or upload more files and repeat the same

If more than one uploaded files have the keyword then all file names are shown. Re-upload of the same file is prevented by hashing the file contents and storing in the database, in event of potential re-upload, there is error shown in the http server logs but not on the web UI.



csp_keywords_id	file_hash	file_name	csp_keyvalue
18	af1873eb07adff49981b6fd9edc69b109457c4274c1b038a9ed02ba99c3fb387	test_file_0.txt	z�5-�□��27�?⊐��Z����~�`_�1 2m_���□�eL%�}Z□:�□��*� �\N�N �!��f����{�R
19	0afd19cacabfb601ad913a381684461e8bf7ae25345baac493fd84881dcd210c	test_file_1.txt	�띪غ□�n□,�□P� ncKw����Zn □A0� :□v��b��0�3��u� �□g""�� ��Pn �□VL�8□�!i□ElL�□�□4��k����=!�Gr�□&l□�□��□�

Image 6: SQL table

Secure Programming Solutions

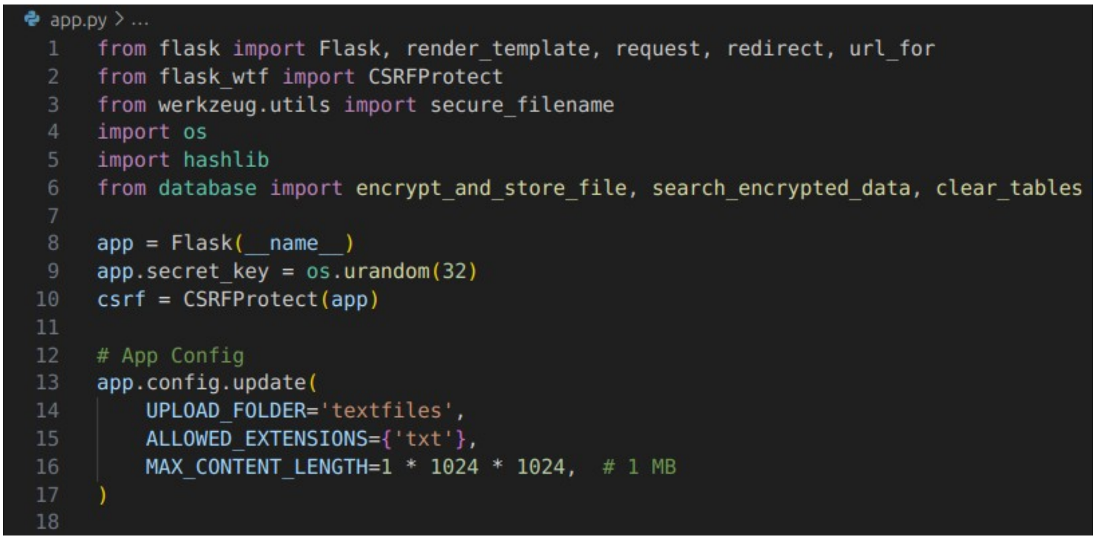
Checklist Used: OWASP Top 10

Table 1: OWASP Top 10 checklist

OWASP Top 10 Issue	Mitigation
Injection	No raw SQL input; all database queries use parameterized SQL.
Broken Auth	No login system; access is local or restricted in deployment.
Sensitive Data Exposure	AES-256 encryption used with secure IVs and padding for file content.
Security Misconfiguration	Docker image is minimal; ports restricted.
Vulnerable Components	requirements.txt uses pinned versions to reduce supply chain risk.
Logging/Monitoring	Minimal logging; prints avoid leaking sensitive data.
Insecure Deserialization	No deserialization used.
CSRF/XSS	No user input rendered directly to HTML; simple form structure, CSRF implemented.
Insufficient Logging	Only basic print logs included, suitable for prototype/testing phase.
Access Control	Currently open; can be containerized and protected at network level.

Code Examples in implementation

File app.py

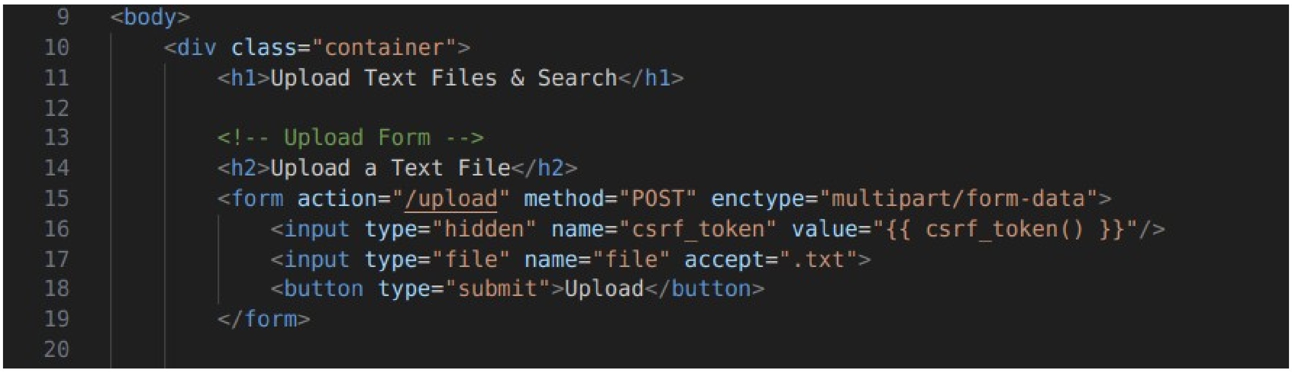


```python
from flask import Flask, render_template, request, redirect, url_for
from flask_wtf import CSRFProtect
from werkzeug.utils import secure_filename
import os
import hashlib
from database import encrypt_and_store_file, search_encrypted_data, clear_tables

app = Flask(__name__)
app.secret_key = os.urandom(32)
csrf = CSRFProtect(app)

# App Config
app.config.update(
    UPLOAD_FOLDER='textfiles',
    ALLOWED_EXTENSIONS={'txt'},
    MAX_CONTENT_LENGTH=1 * 1024 * 1024,  # 1 MB
)

```

Image 7: Secure code examples

- CSRFProtect in Flask (from flask-wtf) adds protection against Cross-Site Request Forgery attacks by embedding a secure token in forms. When a user submits a form, the token is checked to ensure the request is legit and not forged by another site.
- It requires secret key which we set up using os.urandom
- Specify UPLOAD_FOLDER, ALLOWED_EXTENSIONS and MAX_CONTENT_LENGTH

File index.html

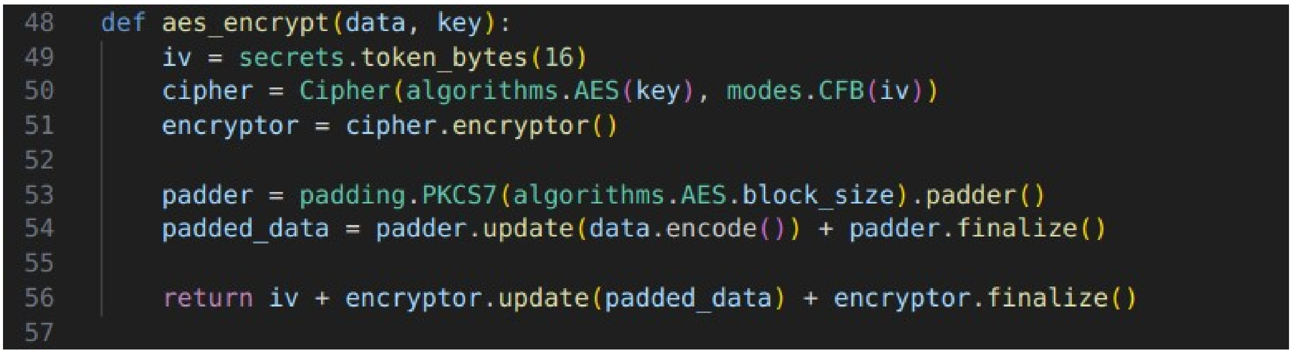


```html
<body>
    <div class="container">
        <h1>Upload Text Files & Search</h1>

        <!-- Upload Form -->
        <h2>Upload a Text File</h2>
        <form action="/upload" method="POST" enctype="multipart/form-data">
            <input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>
            <input type="file" name="file" accept=".txt">
            <button type="submit">Upload</button>
        </form>

```

Image 8: CSRF token in html file

- Include csrf_token in html form where method POST is used

File database.py



```python
def aes_encrypt(data, key):
    iv = secrets.token_bytes(16)
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv))
    encryptor = cipher.encryptor()

    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    padded_data = padder.update(data.encode()) + padder.finalize()

    return iv + encryptor.update(padded_data) + encryptor.finalize()

```

Image 9: AES encryption setup

- Encrypts file content using AES-CFB with random IV
- Uses secrets for cryptographic randomness.
- AES encryption is applied correctly with proper padding and IVs.

Changes from Previous Work

This implementation removes deprecated sse_keywords table and switches from file-based keyword tokenization to full-content encryption. Duplicate uploads are now prevented using SHA-256 file hashes. Database now also includes original filename for clarity.

Previous implementation was a CLI tool which had basic SSE usage strictly for demonstration purposes and had no cleanup or any UI.

Security Testing Performed

- Manual Testing: Attempted SQL injection in search and upload fields: protected by param queries.
- Replay Attacks: Tried re-uploading same file — rejected via hash match.
- File Type Attack: Non-txt files rejected by extension filter.
- Insecure File Name: Sanitized using secure_filename utility.
- Encryption Testing: Verified output not readable without KSKE key.
- Docker Security: Used Ubuntu minimal base, added no unnecessary packages.

Findings & Fixes:

- Duplicate file upload allowed previously – now fixed using file hash.
- Filename was not stored – added to database for user clarity.
- Database wasn't cleared – created button which cleans database.
- Webpage didn't run in Docker initially — fixed by exposing port & using host='0.0.0.0'.
- CSRF token – application had no check for CSRF, added using flask_wtf.

DevSecOps Integration

CI/CD handled via Jenkins pipeline. SSH credentials securely managed via Jenkins credentials manager. Docker image is built after pulling git repo, uploaded to docker hub in pipeline as well. Docker scout enabled for this repo. Sonarqube, Trivy and OWASP DAST scanning added to Jenkins stages.

Jenkins File

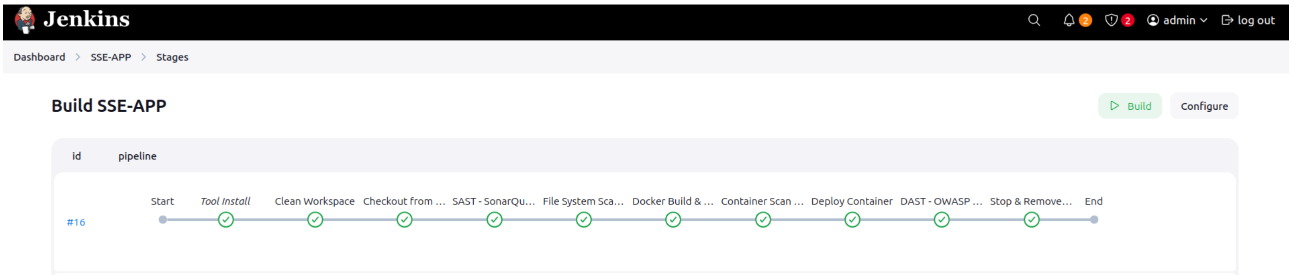
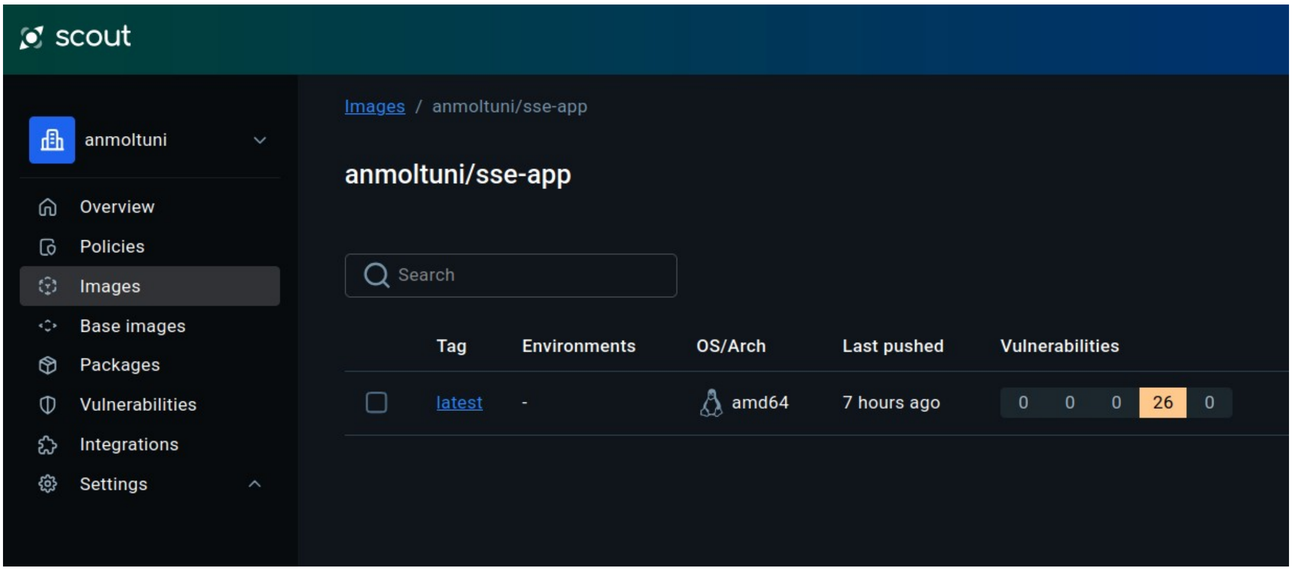


Image 10: Jenkins pipeline stages

Docker Scout



Images / anmoltuni/sse-app

anmoltuni/sse-app

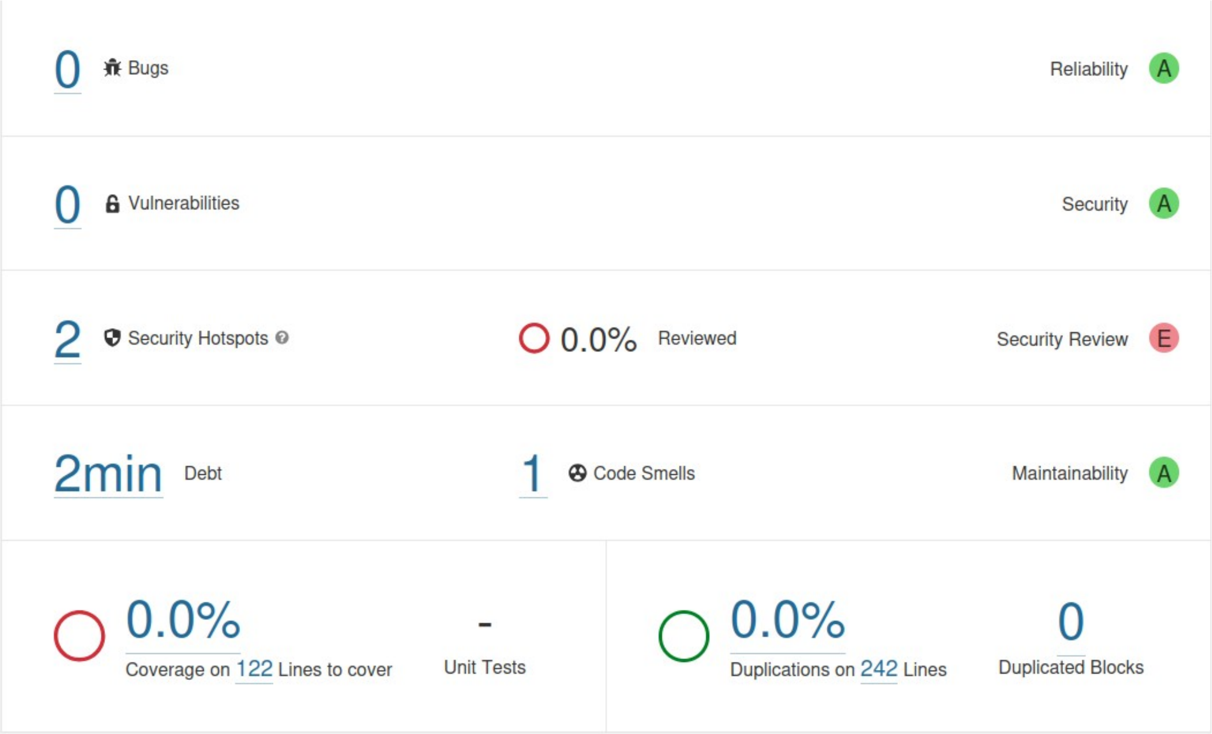
	Tag	Environments	OS/Arch	Last pushed	Vulnerabilities
□	latest	-	amd64	7 hours ago	0 0 0 26 0

Image 11: Docker scount enabled on repo

Docker scout did show 1 CRITICAL issue with the image I started out with, python3-11:alpine and then I bumped it up to python3-13:alpine which has no high level issues.

SonarQube Scan

From sonarqube output I got the advise of adding CSRF to Flask app, declaring 'index.html' as a variable instead of hardcoding it in multiple places, renaming 'file' to 'up_file' since file is similar to in built variable name in python.



0 Bugs — Reliability A

0 Vulnerabilities — Security A

2 Security Hotspots — 0.0% Reviewed — Security Review E

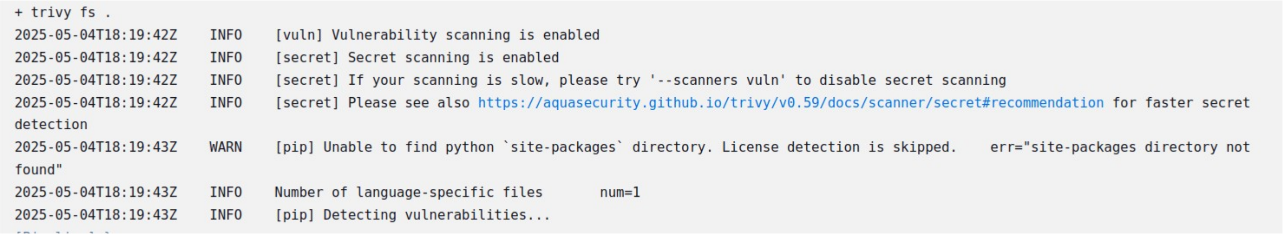
2min Debt — 1 Code Smells — Maintainability A

0.0% Coverage on 122 Lines to cover — - Unit Tests

0.0% Duplications on 242 Lines — 0 Duplicated Blocks

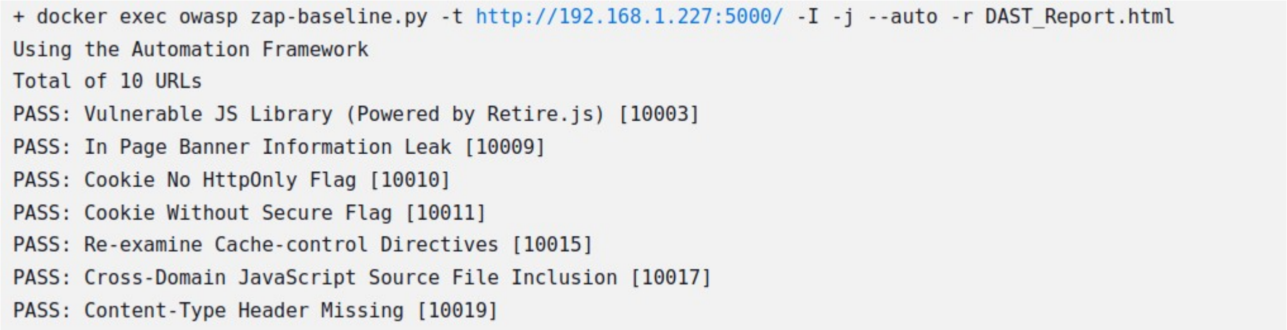
Image 11: Sonarqube dashboard

Trivy and OWASP DAST output



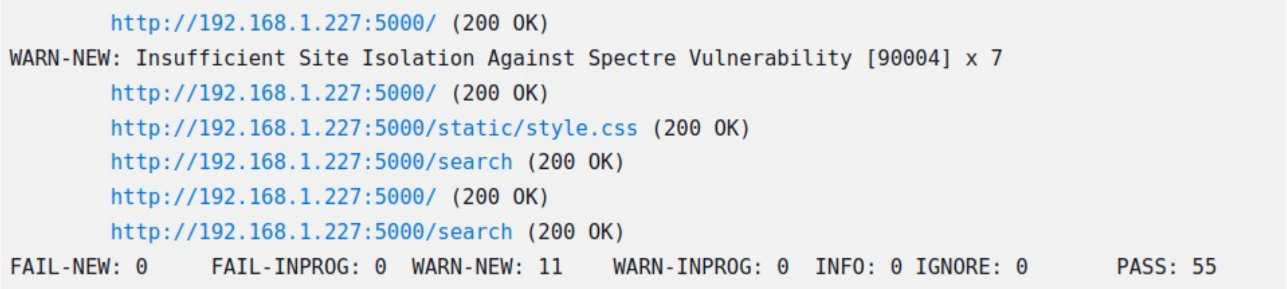
```
+ trivy fs .
2025-05-04T18:19:42Z    INFO    [vuln] Vulnerability scanning is enabled
2025-05-04T18:19:42Z    INFO    [secret] Secret scanning is enabled
2025-05-04T18:19:42Z    INFO    [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2025-05-04T18:19:42Z    INFO    [secret] Please see also https://aquasecurity.github.io/trivy/v0.59/docs/scanner/secret#recommendation for faster secret
detection
2025-05-04T18:19:43Z    WARN    [pip] Unable to find python `site-packages` directory. License detection is skipped.    err="site-packages directory not
found"
2025-05-04T18:19:43Z    INFO    Number of language-specific files       num=1
2025-05-04T18:19:43Z    INFO    [pip] Detecting vulnerabilities...
```

Image 12: Trivy filesystem scan



```
+ docker exec owasp zap-baseline.py -t http://192.168.1.227:5000/ -I -j --auto -r DAST_Report.html
Using the Automation Framework
Total of 10 URLs
PASS: Vulnerable JS Library (Powered by Retire.js) [10003]
PASS: In Page Banner Information Leak [10009]
PASS: Cookie No HttpOnly Flag [10010]
PASS: Cookie Without Secure Flag [10011]
PASS: Re-examine Cache-control Directives [10015]
PASS: Cross-Domain JavaScript Source File Inclusion [10017]
PASS: Content-Type Header Missing [10019]
```

…



```
        http://192.168.1.227:5000/ (200 OK)
WARN-NEW: Insufficient Site Isolation Against Spectre Vulnerability [90004] x 7
        http://192.168.1.227:5000/ (200 OK)
        http://192.168.1.227:5000/static/style.css (200 OK)
        http://192.168.1.227:5000/search (200 OK)
        http://192.168.1.227:5000/ (200 OK)
        http://192.168.1.227:5000/search (200 OK)
FAIL-NEW: 0     FAIL-INPROG: 0  WARN-NEW: 11    WARN-INPROG: 0  INFO: 0 IGNORE: 0       PASS: 55
```

Image 13: OWASP baseline output

Missing Features / Known Issues

- No user authentication
- HTTPS not enforced (to be added in real deployment).
- Search performance degrades with large number of files (no indexing).

Suggestions for Improvement

- Add login/authentication with role-based access.
- Use full-text search over encrypted indexes.
- Store encrypted files in filesystem instead of DB for scalability.
- Add unit tests and pytest coverage in CI.
- Migrate from SQLite to PostgreSQL.
- Add JWT token auth if making API endpoints.

Use of AI

ChatGPT has been used while creating this project for various reasons, but mainly as a guide while coding. The version used is GPT-4-turbo (OpenAI) — same model powering ChatGPT as of May 2025.

GPT has also been used for its security recommendations and to help with DevSecOps related tools.