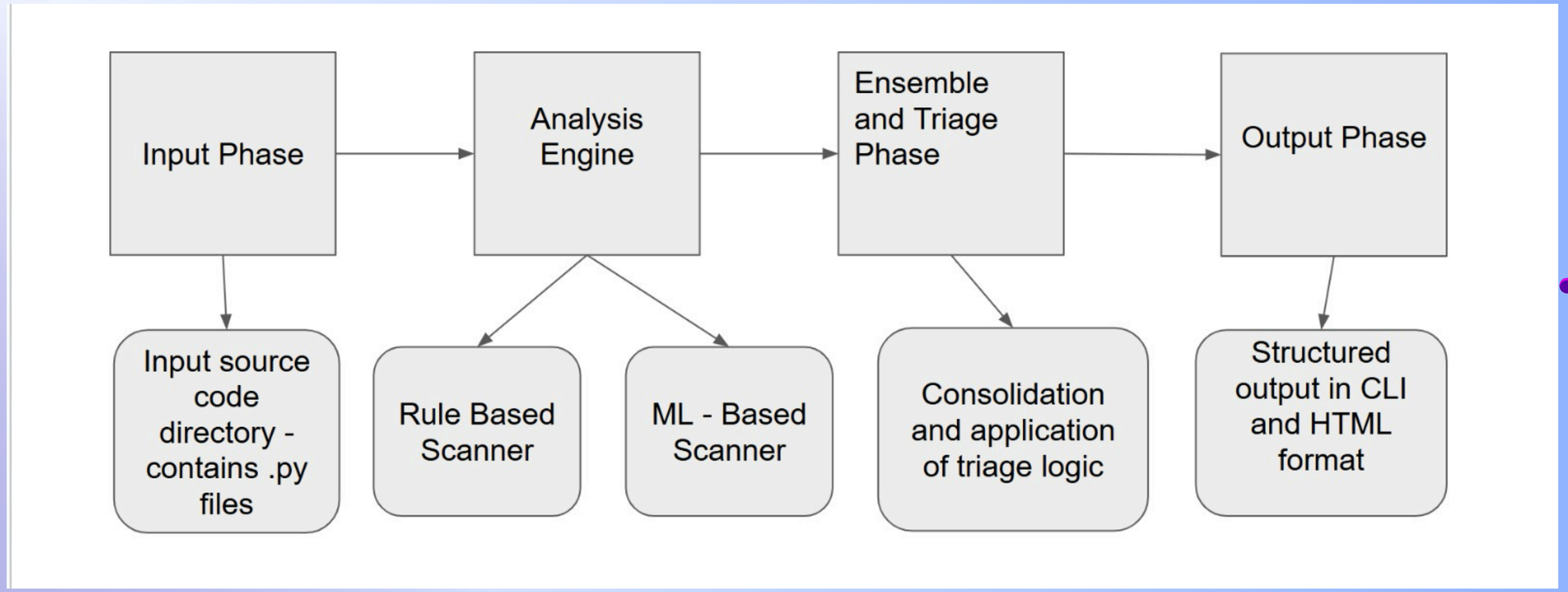
SMART CODE VULNARABILITY TRIAGE SYSTEM

TEAM NO: 53 - CODE HUNTERS

ARCHITECTURE FLOW DIAGRAM



INPUT PHASE

- Users provide a directory containing source code files, primarily .py files for Python.
- The system automatically traverses these directories and prepares files for scanning.
- Highlight challenges in manual code review versus automation.
- State that this phase is critical for organizing raw data to streamline analysis.

ANALYSIS ENGINE

- Rule-Based Scanner: Utilizes regular expressions and known vulnerability patterns (e.g., SQL injection, cross-site scripting) to flag potential issues fast and reliably.
- Machine Learning-Based Scanner: Employs AI models trained on labeled datasets to detect subtle, complex vulnerabilities that pattern matching might miss.
- AST (Abstract Syntax Tree) analysis enhances context understanding beyond simple text patterns, reducing false positives.

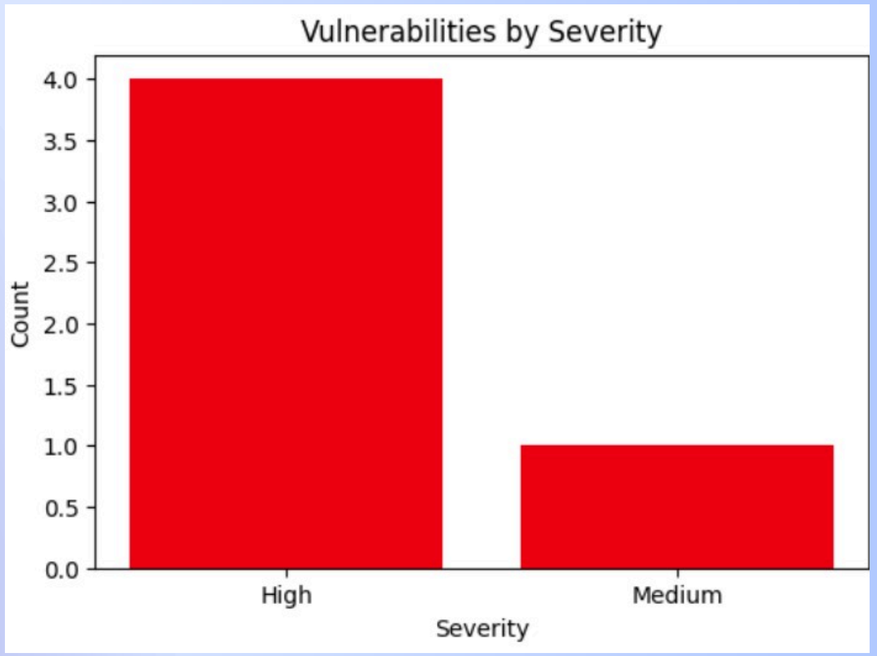
OUTPUT PHASE

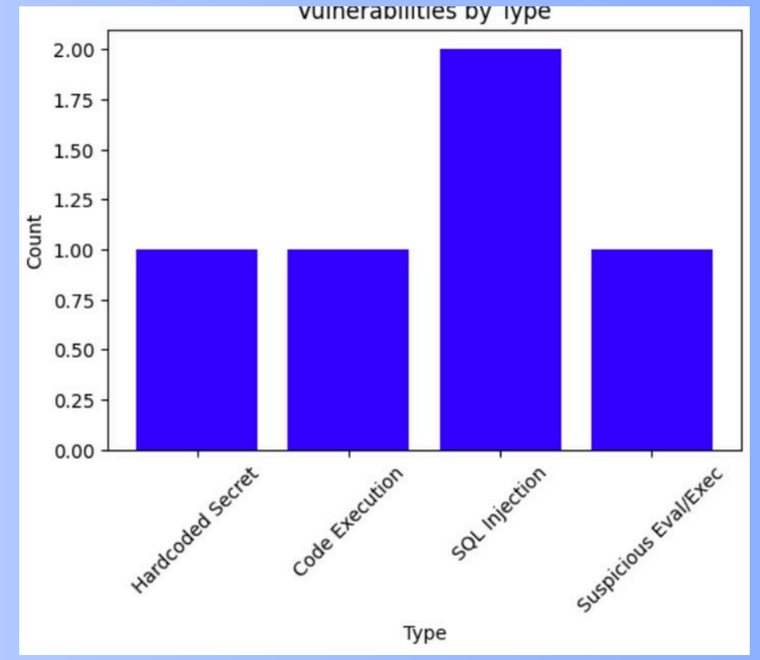
- Command Line Interface (CLI) reports for quick developer feedback during coding or CI/CD.
- HTML reports with a user-friendly interface for detailed analysis, sharing, and audit trails.
- Mention the structured data formats (JSON, HTML) facilitate integration with other security tools and dashboards.
- Discuss how outputs help teams remediate vulnerabilities efficiently, reducing security debt.

RESULTS

report.json


```
{
    "High": [
        {
            "file": "./test_code.py",
            "line": 7,
            "issue": "Hardcoded API Key",
            "severity": "High",
            "code": "api_key = \"12345-abcde\""
        },
        {
            "file": "./test_code.py",
            "line": 14,
            "issue": "Hardcoded Password",
            "severity": "High",
            "code": "database_password = \"my_db_password\""
        }
    ],
    "Medium": [
        {
            "file": "./my_project/src/utils.py",
            "line": 4,
            "issue": "Suspicious Eval/Exec",
            "severity": "Medium",
            "code": "exec(command)"
        }
    ],
    "Low": []
}
```



WEBSITE INTERFACE

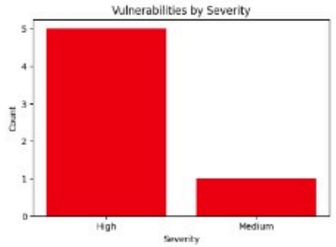
Vulnerability Scan Report

Total Findings: 6

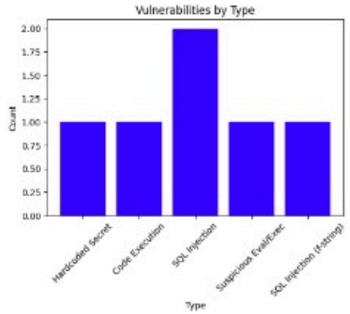
Files Scanned: main.py, utils.py

Summary Charts

By Severity



By Type

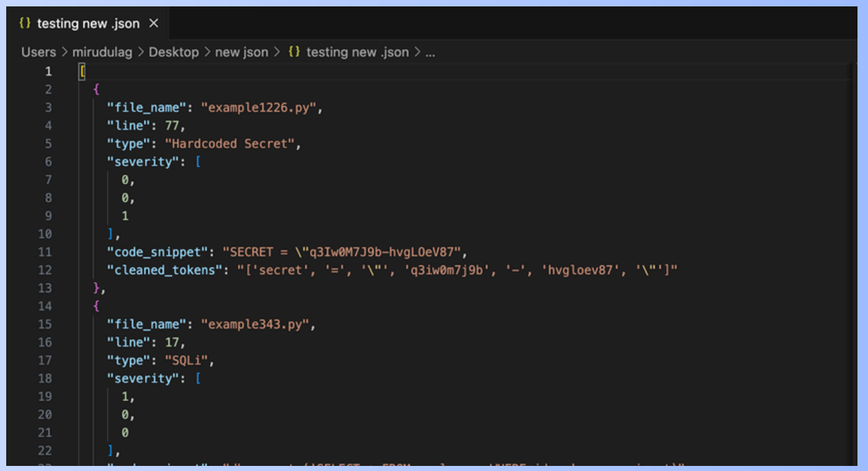


Findings

File	Line	Type	Severity	Detected By
utils.py	6	Hardcoded Secret	High	AST
utils.py	4	Code Execution	High	AST
main.py	4	SQL Injection	High	AST
main.py	11	SQL Injection	High	AST
utils.py	4	Suspicious Eval/Exec	Medium	Regex
main.py	11	SQL Injection (f-string)	High	Regex

- Shows vulnerabilities detected using rule-based scans with logic and pattern matching.
- JSON output and charts summarize severity and type of issues found, highlighting prominent risks like hardcoded secrets and SQL injection.

DATASETS



- Dataset collection
- Synthetic data generation for missing values
- Data preprocessing (cleaning, tokenization, normalization)
- Split into training (70%), validation (15%), and testing (15%) sets

DEEP LEARNING MODELS USED

- CNN (Convolutional Neural Network):
  - Extracts structural features from code tokens/snippets.
  - Detects localized vulnerability patterns (e.g., suspicious function calls).
- RNN (Recurrent Neural Network):
  - Processes source code as a sequence.
  - Captures contextual dependencies across multiple lines/functions.
- Random Forest (Ensemble Classifier):
  - Traditional ML model, robust to noisy features.
  - Provides strong baseline classification with interpretability.
- Why Combine CNN, RNN & Random Forest?
  - CNN → Detects local features in code.
  - RNN → Captures sequential flow & context.
  - Random Forest → Adds stability and reduces overfitting.
  - Together → Hybrid ensemble leverages strengths of DL + ML → higher detection accuracy, lower false positives.

OUTPUTS

MODEL TRAIINING

```
Counts: 1050 225 225

Training CNN…
Epoch 1/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 8s 219ms/step - accuracy: 0.6042 - loss: 1.0041 - val_accuracy: 0.8622 - val_loss: 0.5611 - learning_rate: 0.0010
Epoch 2/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 4s 253ms/step - accuracy: 0.8644 - loss: 0.4948 - val_accuracy: 0.8622 - val_loss: 0.4616 - learning_rate: 0.0010
Epoch 3/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 5s 277ms/step - accuracy: 0.8554 - loss: 0.4106 - val_accuracy: 0.8622 - val_loss: 0.5047 - learning_rate: 0.0010
Epoch 4/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 3s 176ms/step - accuracy: 0.8490 - loss: 0.4070 - val_accuracy: 0.8622 - val_loss: 0.4277 - learning_rate: 5.0000e-04
Epoch 5/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 3s 167ms/step - accuracy: 0.8661 - loss: 0.3784 - val_accuracy: 0.8622 - val_loss: 0.4444 - learning_rate: 5.0000e-04
Epoch 6/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 7s 296ms/step - accuracy: 0.8653 - loss: 0.3700 - val_accuracy: 0.8622 - val_loss: 0.4354 - learning_rate: 2.5000e-04

Training RNN…
Epoch 1/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 25s 1s/step - accuracy: 0.5177 - loss: 1.0399 - val_accuracy: 0.8622 - val_loss: 0.7072 - learning_rate: 0.0010
Epoch 2/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 22s 1s/step - accuracy: 0.8660 - loss: 0.5805 - val_accuracy: 0.8622 - val_loss: 0.4941 - learning_rate: 0.0010
Epoch 3/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 43s 1s/step - accuracy: 0.8651 - loss: 0.3942 - val_accuracy: 0.8622 - val_loss: 0.4615 - learning_rate: 0.0010
Epoch 4/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 19s 1s/step - accuracy: 0.8654 - loss: 0.3824 - val_accuracy: 0.8622 - val_loss: 0.4425 - learning_rate: 0.0010
Epoch 5/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 21s 1s/step - accuracy: 0.8703 - loss: 0.3699 - val_accuracy: 0.8622 - val_loss: 0.4366 - learning_rate: 0.0010
Epoch 6/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 24s 1s/step - accuracy: 0.8803 - loss: 0.3323 - val_accuracy: 0.8578 - val_loss: 0.4495 - learning_rate: 0.0010
Epoch 7/12
17/17 ━━━━━━━━━━━━━━━━━━━━ 19s 1s/step - accuracy: 0.8968 - loss: 0.2876 - val_accuracy: 0.8489 - val_loss: 0.4405 - learning_rate: 5.0000e-04
Feature shapes: (1050, 384) (225, 384) (225, 384)
Picked ExtraTrees (val macro-F1 RF=0.781, ET=0.806)
```

- **Epoch-wise training logs display how accuracy and loss improve for both CNN and RNN models, achieving strong validation performance and confirming effective learning**

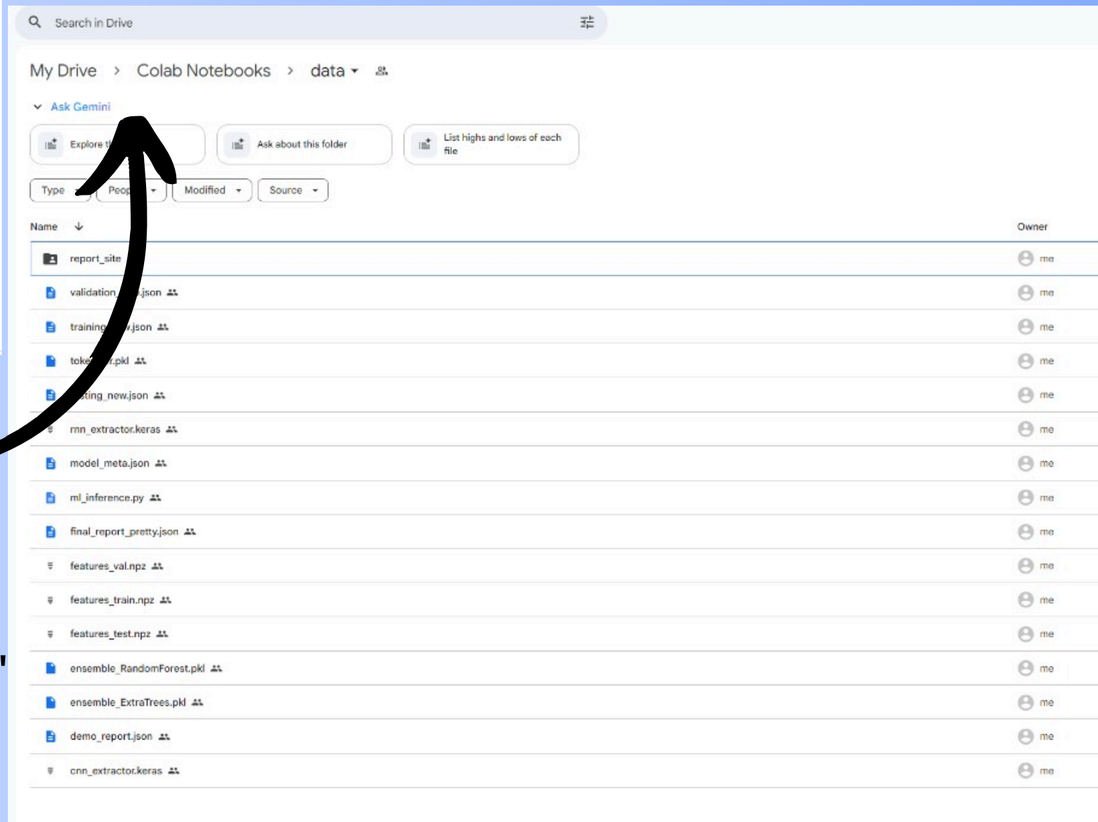
OUTPUTS

```
[Ensemble] TEST REPORT
                  precision    recall  f1-score   support

            SQLi       0.91      0.79      0.85        95
             XSS       0.82      0.90      0.86        70
Hardcoded Secret       0.82      0.90      0.86        60

        accuracy                           0.85       225
       macro avg       0.85      0.86      0.85       225
    weighted avg       0.86      0.85      0.85       225

Confusion matrix:
 [[75 12  8]
 [ 3 63  4]
 [ 4  2 54]]

Saved to: /content/drive/MyDrive/Colab Notebooks/data
```

```
Feature shapes: (1050, 384) (225, 384) (225, 384)
Chosen: RandomForest  (val macro-F1  RF=0.820, ET=0.785)

[Ensemble] TEST REPORT
                  precision    recall  f1-score   support

            SQLi       0.90      0.80      0.85        95
             XSS       0.86      0.90      0.88        70
Hardcoded Secret       0.81      0.92      0.86        60

        accuracy                           0.86       225
       macro avg       0.86      0.87      0.86       225
    weighted avg       0.87      0.86      0.86       225

Confusion matrix:
 [[76 10  9]
 [ 3 63  4]
 [ 5  0 55]]

Saved to: /content/drive/MyDrive/Colab Notebooks/data
 - cnn_extractor.keras
 - rnn_extractor.keras
 - tokenizer.pkl
 - ensemble_RandomForest.pkl
 - model_meta.json
```

Training the model with CNN and RNN sequenced and maxpooled

```
Saved to: /content/drive/MyDrive/Colab Notebooks/data
```



- **Evaluation Results saved in organized folder for reproducibility and easy access"**
- **Ensures transparent documentation of model evaluation.**

OUTPUTS

CNN EXTRACTOR

Model: "cnn_extractor"

Layer (type)	Output Shape	Param #
input_layer_11 (InputLayer)	(None, 200)	0
embedding_11 (Embedding)	(None, 200, 64)	768,000
conv1d_18 (Conv1D)	(None, 200, 128)	41,088
max_pooling1d_9 (MaxPooling1D)	(None, 100, 128)	0
conv1d_19 (Conv1D)	(None, 100, 128)	82,048
cnn_features (GlobalMaxPooling1D)	(None, 128)	0
dropout_11 (Dropout)	(None, 128)	0
dense_13 (Dense)	(None, 128)	16,512
dropout_12 (Dropout)	(None, 128)	0
dense_14 (Dense)	(None, 3)	387

Total params: 908,035 (3.46 MB)
Trainable params: 908,035 (3.46 MB)
Non-trainable params: 0 (0.00 B)
Model: "rnn_extractor"

- Table shows the architecture of the CNN feature extractor model layer-by-layer.
- Includes layer types, output dimensions, and trainable parameter counts per layer.
- Highlights how data is transformed from inputs to compact learned feature vectors.
- Total trainable params (~908k) indicate model complexity and learning capacity.
- Ends with a small dense output layer for 3-class classification.
- Summary lines provide concise stats: total params, trainable vs. non-trainable parameters.
- Model ready for training on tokenized sequence data for multi-class classification.

OUTPUTS

RNN EXTRACTOR

Model: "rnn_extractor"

Layer (type)	Output Shape	Param #
input_layer_12 (InputLayer)	(None, 200)	0
embedding_12 (Embedding)	(None, 200, 64)	768,000
bidirectional_1 (Bidirectional)	(None, 200, 256)	148,992
rnn_features (GlobalMaxPooling1D)	(None, 256)	0
dropout_13 (Dropout)	(None, 256)	0
dense_15 (Dense)	(None, 128)	32,896
dropout_14 (Dropout)	(None, 128)	0
dense_16 (Dense)	(None, 3)	387

Total params: 950,275 (3.63 MB)
Trainable params: 950,275 (3.63 MB)
Non-trainable params: 0 (0.00 B)

- The RNN model processes sequences step-by-step using recurrent units capable of capturing temporal dependencies.
- Includes embedded input layer followed by bidirectional GRU layers which extract context-aware sequential features.
- Final features are condensed (e.g., via pooling or using last hidden state) before classification layers.
- Contains dropout layers to prevent overfitting and dense layers for decision making.
- Total trainable parameters indicate model complexity and learning capacity.
- Designed for sequential tokenized data, enabling learning from temporal patterns.
- Output layer predicts among 3 classes, completing the classification pipeline.

FINAL OUTPUTS

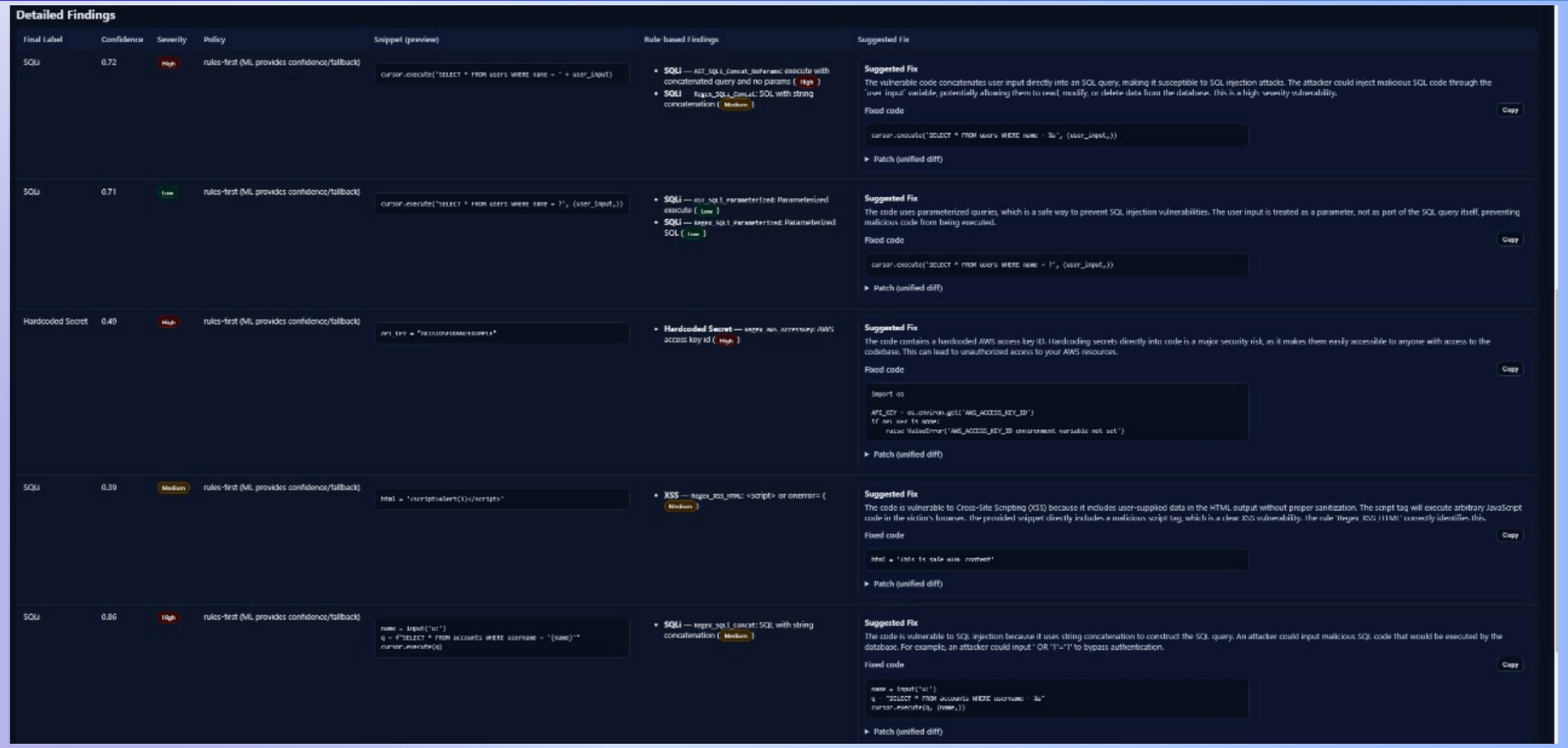
```
Mounted at /content/drive
[LOAD] loading artifacts…
✅ Main ML report written:
JSON: /content/drive/MyDrive/Colab Notebooks/data/report_site/final_report_pretty.json
SITE: /content/drive/MyDrive/Colab Notebooks/data/report_site/index.html
[+] Fetching http://testphp.vulnweb.com ...
[+] Gathering frontend sources …
[+] Saved 3 source files into /content/drive/MyDrive/Colab Notebooks/data/report_site/webscan/testphp.vulnweb.com_root_20250917-215617
[+] Scanning …
[+] JSON report: /content/drive/MyDrive/Colab Notebooks/data/report_site/webscan/testphp.vulnweb.com_root_20250917-215617/site_report.json
[+] HTML report: /content/drive/MyDrive/Colab Notebooks/data/report_site/webscan/testphp.vulnweb.com_root_20250917-215617/site_report.html
[+] Linked site scan from: /content/drive/MyDrive/Colab Notebooks/data/report_site/index.html

✅ Website scan saved under: /content/drive/MyDrive/Colab Notebooks/data/report_site/webscan/testphp.vulnweb.com_root_20250917-215617

🎉 Pipeline complete.
Main JSON: /content/drive/MyDrive/Colab Notebooks/data/report_site/final_report_pretty.json
Main Site: /content/drive/MyDrive/Colab Notebooks/data/report_site/index.html
Report site folder: /content/drive/MyDrive/Colab Notebooks/data/report_site
```


```
{
  "scanner": "Rules + ML (CNN/RNN\u2192Ensemble) + Suggestions",
  "labels": [
    "SQLi",
    "XSS",
    "Hardcoded Secret"
  ],
  "count": 6,
  "items": [
    {
      "id": "8cfb0e65-d484-4fa2-a4e1-4e2a85caa8bc",
      "timestamp": "2025-09-17T21:56:052",
      "language": "python",
      "snippet_preview": "cursor.execute('SELECT * FROM users WHERE name = ' + user_input)",
      "snippet_full": "cursor.execute('SELECT * FROM users WHERE name = ' + user_input)",
      "ensemble_policy": "rules-first (ML provides confidence/fallback)",
      "rule_based_findings": [
        {
          "type": "SQLi",
          "evidence": "execute with concatenated query and no params",
          "severity": "high",
          "rule": "AST_SQLi_Concat_NoParams"
        },
        {
          "type": "SQLi",
          "evidence": "SQL with string concatenation",
          "severity": "medium",
          "rule": "Regex_SQLi_Concat"
        }
      ],
      "ml_prediction": {
        "label": "SQLi",
        "confidence": 0.7239320846885485,
        "probabilities": {
          "SQLi": 0.7239320846885485,
          "XSS": 0.271067915311451,
          "Hardcoded Secret": 0.005
```


- **Console output confirms that all ML and scan reports are saved systematically for audit and future review.**
- **Shows completeness of the pipeline from scanning, reporting, to organized storage in the workspace.**

- **JSON output demonstrates how findings, rule evidence, and ML predictions are combined for each code item.**
- **Provides transparency in reporting, showing multiple evidence types and confidence scores for detected vulnerabilities.**

FINAL OUTPUTS



Detailed Findings

Final Label	Confidence	Severity	Policy	Snippet (preview)	Rule-based Findings	Suggested Fix
SQLi	0.72	High	rules-first (ML provides confidence/fallback)	`cursor.execute("SELECT * FROM users WHERE name = " + user_input)`	• **SQLi** — AST_SQLi_Concat_NoParams: execute with concatenated query and no params (High) • **SQLi** — Regex_SQLi_Concat: SQL with string concatenation (Medium)	**Suggested Fix** The vulnerable code concatenates user input directly into an SQL query, making it susceptible to SQL injection attacks. The attacker could inject malicious SQL code through the 'user_input' variable, potentially allowing them to read, modify, or delete data from the database. This is a high-severity vulnerability. Fixed code Copy `cursor.execute('SELECT * FROM users WHERE name = %s', (user_input,))` ▸ Patch (unified diff)
SQLi	0.71	Low	rules-first (ML provides confidence/fallback)	`cursor.execute("SELECT * FROM users WHERE name = ?", (user_input,))`	• **SQLi** — AST_SQLi_Parameterized: Parameterized execute (Low) • **SQLi** — Regex_SQLi_Parameterized: Parameterized SQL (Low)	**Suggested Fix** The code uses parameterized queries, which is a safe way to prevent SQL injection vulnerabilities. The user input is treated as a parameter, not as part of the SQL query itself, preventing malicious code from being executed. Fixed code Copy `cursor.execute("SELECT * FROM users WHERE name = ?", (user_input,))` ▸ Patch (unified diff)

Hardcoded Secret	0.49	High	rules-first (ML provides confidence/fallback)	`API_KEY = "AKIAIOSFODNN7EXAMPLE"`	• **Hardcoded Secret** — Regex AWS Access Key: AWS access key id (High)	**Suggested Fix**
						The code contains a hardcoded AWS access key ID. Hardcoding secrets directly into code is a major security risk, as it makes them easily accessible to anyone with access to the codebase. This can lead to unauthorized access to your AWS resources.
						Fixed code Copy
						`import os`
						`API_KEY = os.environ.get('AWS_ACCESS_KEY_ID')`
						`if API_KEY is None:`
						`  raise ValueError('AWS_ACCESS_KEY_ID environment variable not set')`
						▸ Patch (unified diff)

SQLi	0.39	Medium	rules-first (ML provides confidence/fallback)	`html = '<script>alert(1)</script>'`	• **XSS** — Regex_XSS_HTML: <script> or onerror= (Medium)	**Suggested Fix** The code is vulnerable to Cross-Site Scripting (XSS) because it includes user-supplied data in the HTML output without proper sanitization. The script tag will execute arbitrary JavaScript code in the victim's browser. The provided snippet directly includes a malicious script tag, which is a clear XSS vulnerability. The rule 'Regex XSS HTML' correctly identifies this. Fixed code Copy `html = 'this is safe now content'` ▸ Patch (unified diff)

SQLi	0.86	High	rules-first (ML provides confidence/fallback)	`name = input('u:')`
				`q = f"SELECT * FROM accounts WHERE username = '{name}'"`
				`cursor.execute(q)`
					• **SQLi** — Regex_SQLi_Concat: SQL with string concatenation (Medium)	**Suggested Fix**
						The code is vulnerable to SQL injection because it uses string concatenation to construct the SQL query. An attacker could input malicious SQL code that would be executed by the database. For example, an attacker could input ' OR '1'='1' to bypass authentication.
						Fixed code Copy
						`name = input('u:')`
						`q = "SELECT * FROM accounts WHERE username = %s"`
						`cursor.execute(q, (name,))`
						▸ Patch (unified diff)

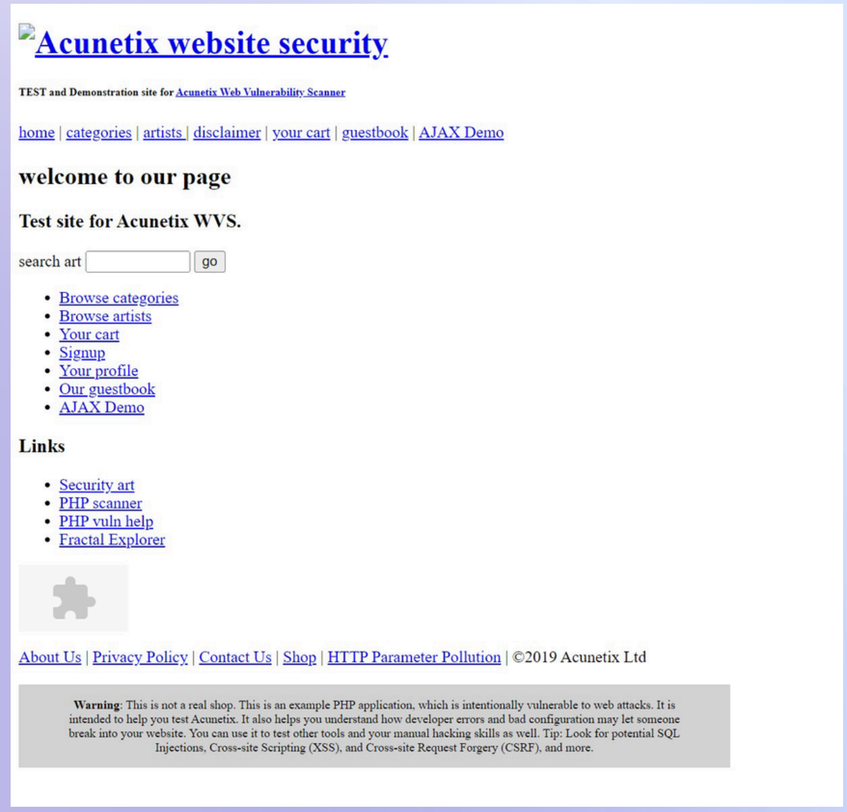
- **Detailed findings table links each detection to its evidence, severity, rule, and suggested fixes.**
- **Helps users understand vulnerabilities and apply fixes directly, improving code safety via actionable insights.**

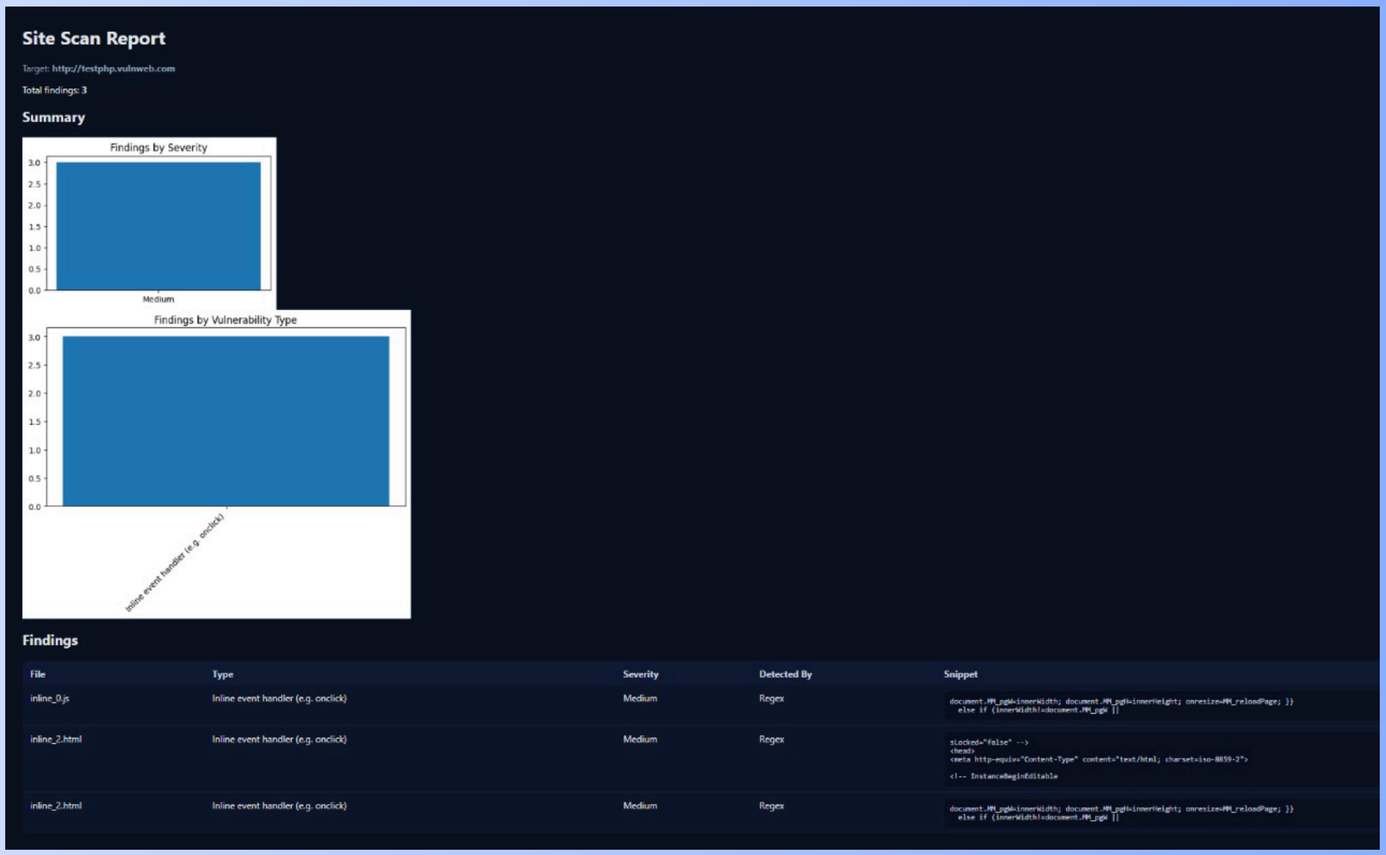
FINAL OUTPUTS

- Bar chart visualizes the vulnerability findings per label detected by the integrated ML and rule-based system.
- Shows that most findings are SQLi, with results for XSS and Hardcoded Secret also captured for comprehensive security analysis.

FINAL OUTPUTS

INPUT WEBSITE





- **Website URL is input, and the tool automatically fetches and analyzes its source code.**
- **The integrated ML and rule-based model detects and reports all vulnerabilities found in the website's code for fast, automated security assessment.**

SUMMARY & IMPACT

- The integrated system combines rule-based approaches with advanced ML models (CNN and RNN) to detect code vulnerabilities quickly and accurately.
- It can scan source files or website URLs, auto-extract code, and report all identified security risks in a structured, actionable format.

Impact

- This automated tool greatly reduces manual effort and error in vulnerability detection, streamlining secure coding practices for developers.
- It ensures faster remediation of security flaws in both offline code and live websites, raising the bar for real-world application security.

THANK YOU!