Project Title: Gen-AI based Automation of Vulnerability Discovery and Testing

Short Project Description:

The system integrates code/data crawling, AI-powered vulnerability prediction, automated test generation, execution, risk analysis, and reporting to streamline security assessments.

Component	Role
Code/Data Crawler	Gathers codebases, APIs, application details
Gen-AI Vulnerability Predictor	Predicts vulnerable components in code or system configs
Automated Test Case Generator	Generates fuzzing and penetration tests based on LLM analysis
Execution and Monitoring Module	Runs generated tests against targets
Risk Analysis Engine	Assesses impact of discovered bugs
Report Builder	Writes testing results, vulnerability proofs, and recommendations

Component Details:

1. **Code/Data Crawler**:
   - Collects:
     - Source code
     - Swagger/OpenAPI specs
     - Infrastructure configurations (Terraform, Ansible, etc)
     - Etc
2. **Gen-AI Vulnerability Predictor**:
   - Uses Transformer models to predict:
     - SQLi risk in APIs
     - Insecure configurations
     - Weak authentication patterns
     - Etc
3. **Automated Test Case Generator**:
   - Creates:
     - Custom fuzzers
     - Specific payloads for injections
4. **Execution and Monitoring Module**:
   - Launches generated tests.
   - Monitors application/system responses for anomalies.
5. **Risk Analysis Engine**:
   - Measures severity of discovered issues.

    - Predicts exploitability.
6. **Report Builder**:
    - Summarizes vulnerabilities, risks, and suggested remediations.

Overall System Flow:

- Input: Source code, APIs, or system configs
- Output: Discovered vulnerabilities and risks
- Focus: **Fully-automated, AI-driven vulnerability discovery/testing**.

Internal Functioning of Each Module:

1. Code/Data Crawler

- **Purpose**:
    - Gather raw input data.
- **How it works**:
    - GitHub, GitLab API scrapers.
    - Swagger/OpenAPI parsers.
    - Terraform file parsers for infrastructure configs.
    - Etc.

---

2. Gen-AI Vulnerability Predictor

- **Purpose**:
    - Predict vulnerabilities using LLMs.
- **How it works**:
    - Input:
        - Code snippets
        - API specs
        - Config files
        - Etc
    - Model identifies:
        - SQL injections
        - Insecure deserialization
        - Open S3 buckets
        - Etc
- **Model**:
    - Fine-tuned CodeBERT/GraphCodeBERT or GPT-4 custom prompts, etc.

---

3. Automated Test Case Generator

- **Purpose**:
  - Generate test cases and payloads.
- **How it works**:
  - Input:
    - Vulnerability hypothesis (e.g., possible SQLi at /login, etc)
  - Output:
    - Fuzzing payloads
    - Specific exploit strings
    - Etc
- **Smart payloads**:
  - For example, LLM writes SQL Injection payloads tuned for target stack (e.g., MySQL vs PostgreSQL).

---

4. Execution and Monitoring Module

- **Purpose**:
  - Run generated tests automatically.
- **How it works**:
  - Executes HTTP requests, CLI commands, fuzzers, etc.
  - Monitors target responses:
    - HTTP 500s
    - Timeout behavior
    - Error messages
    - Etc

---

5. Risk Analysis Engine

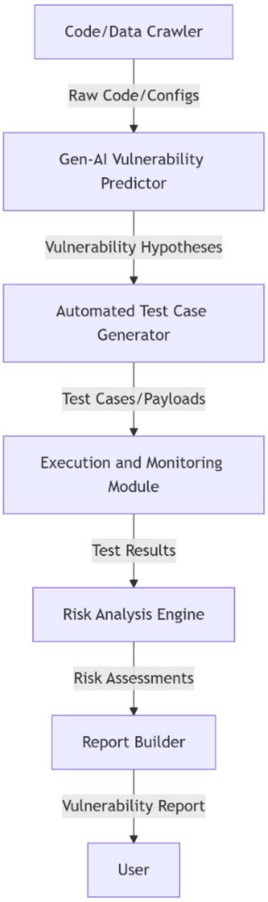
- **Purpose**:
  - Assess severity of findings.
- **How it works**:
  - LLM-based risk scoring:
    - Likelihood + Impact = Risk.

---

6. Report Builder

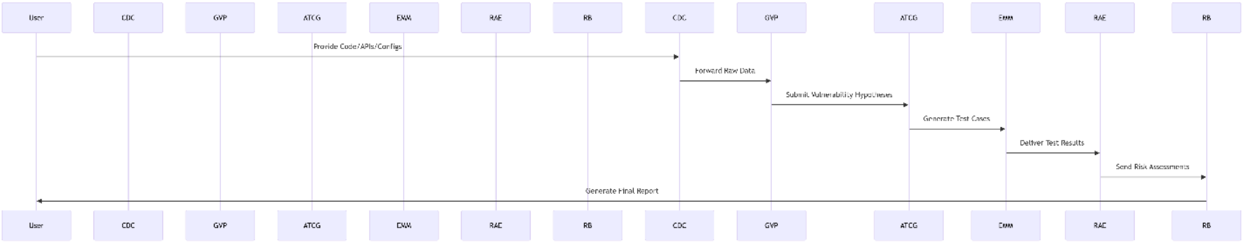
- **Purpose**:
  - Prepare structured vulnerability reports.
- **How it works**:
  - Maps:
    - Vulnerability
    - Proof of exploitability
    - Recommended fixes
    - CVSS scores

Component Diagram



- **Code/Data Crawler**: Collects source code, API specs (e.g., Swagger, etc), and infrastructure configurations (e.g., Terraform, etc).
- **Gen-AI Vulnerability Predictor**: Uses LLMs (e.g., CodeBERT, etc) to predict vulnerabilities like SQLi, insecure deserialization, or misconfigured cloud storage.
- **Automated Test Case Generator**: Generates targeted test payloads (e.g., SQLi strings, etc) based on AI hypotheses.
- **Execution and Monitoring Module**: Runs tests and monitors responses (e.g., HTTP 500 errors, timeouts, etc).
- **Risk Analysis Engine**: Evaluates exploit likelihood and impact using AI-driven scoring.
- **Report Builder**: Compiles findings into structured reports with CVSS scores and remediation steps.

Sequence Diagram



1. **User** provides code, APIs, or configurations to the **Code/Data Crawler**.
2. **Crawler** gathers data and sends it to the **Gen-AI Vulnerability Predictor** for analysis.
3. **Predictor** identifies vulnerabilities and forwards hypotheses to the **Automated Test Case Generator**.
4. **Test Generator** creates tailored tests and triggers the **Execution and Monitoring Module**.
5. **Execution Module** runs tests, captures results, and sends them to the **Risk Analysis Engine**.
6. **Risk Engine** assesses severity and passes findings to the **Report Builder**.
7. **Report Builder** generates a final report for the **User**, including proofs and fixes.

Detailed Project Description: Gen-AI Based Automation of Vulnerability Discovery and Testing

The system integrates code/data crawling, AI-powered vulnerability prediction, automated test generation, execution, risk analysis, and reporting to streamline security assessments.

---

1. System Components and Roles

1.1 Code/Data Crawler

Purpose: Collect codebases, API specifications, and infrastructure configurations.
Implementation Details (e.g.):

- **Tools**:
  - **GitHub/GitLab API**: Fetch repositories via Python libraries like `PyGithub`.
  - **Swagger/OpenAPI Parser**: Extract API endpoints and parameters using `swagger-parser`.
  - **Terraform/Ansible Parser**: Analyze infrastructure-as-code files for misconfigurations.
  - **Etc**.
- **Example Code**:

```python
from github import Github
g = Github("API_TOKEN")
repo = g.get_repo("owner/repo")
contents = repo.get_contents("")
for file in contents:
    if file.name.endswith((".tf", ".yaml")):
        print(f"Downloading {file.path}")
```

1.2 Gen-AI Vulnerability Predictor

Purpose: Predict vulnerabilities in code, APIs, and configurations using AI models.
Implementation Details (e.g.):

- **Models**:

    - **CodeBERT/GraphCodeBERT**: Fine-tuned for code vulnerability detection (e.g., SQLi, XSS, etc).
    - **GPT-4**: Generate hypotheses via prompts like:

      ```
      prompt = f"Analyze this API route: {api_spec}. Predict potential vulnerabilities."
      response = openai.ChatCompletion.create(model="gpt-4", messages=[{"role": "user", "content": prompt}])
      ```

- **Training**:
    - Use datasets like SARD (Software Assurance Reference Dataset) to fine-tune models.
    - Example vulnerability labels: `CWE-89 (SQLi)`, `CWE-79 (XSS)`.

1.3 Automated Test Case Generator

Purpose: Generate targeted test payloads and fuzzing inputs.

Implementation Details (e.g.):

- **Tools**:
    - **Radamsa**: Generate generic fuzzing inputs.
    - **LLM-Powered Payloads**:

      ```
      prompt = "Generate SQLi payloads for a MySQL backend targeting /login endpoint."
      payloads = llm.generate(prompt)  # Output: 'admin' OR 1=1 -- , UNION SELECT ...
      ```

- **Integration (optional)**:
    - Convert payloads into `Burp Suite` or `Postman` collections for HTTP testing.

1.4 Execution and Monitoring Module

Purpose: Execute tests and monitor target responses.

Implementation Details (e.g.):

- **Tools**:
    - **OWASP ZAP API**: Automate HTTP testing.
    - **Custom Scripts**:

      ```
      import requests
      for payload in payloads:
      ```

```
    response = requests.post("https://target/login", data={"user": payload})
    if "error" in response.text:
        log_vulnerability("SQLi", payload)
```

- **Monitoring**:
  - Capture logs, HTTP status codes, and error messages.
  - Use `Elasticsearch` for real-time anomaly detection.

1.5 Risk Analysis Engine

Purpose: Evaluate exploit likelihood and business impact.

Implementation Details (e.g.):

- **Scoring Algorithm**:

```
risk_score = (likelihood * 0.7) + (impact * 0.3)  # Likelihood: 0-10, Impact: 0-10
```

- **LLM Contextual Analysis**:

```
prompt = f"Assess risk of SQLi at /login. Likelihood: 8/10, Impact: 9/10. Justify score."
justification = llm.generate(prompt)
```

1.6 Report Builder

Purpose: Generate structured reports with proofs and fixes.

Implementation Details (e.g.):

- **Tools**:
  - **Jinja2**: Template HTML/PDF reports.
  - **CVSS Calculator**: Assign CVSS v3.1 scores.
  - **Etc**
- **Example Report Entry**:

```
## Vulnerability: SQL Injection (CWE-89)
**Endpoint**: `/login`
**Payload**: `admin' OR 1=1 -- `
**CVSS**: 9.8 (Critical)
**Fix**: Use parameterized queries with SQLAlchemy.
```

2. System Integration and Component Interaction

1. **Data Collection**:
   - **Crawler → Predictor**: Sends code/API/config data.
2. **Vulnerability Prediction**:
   - **Predictor → Test Generator**: Flags potential issues (e.g., SQLi).
3. **Test Execution**:
   - **Test Generator → Execution Module**: Sends payloads.
4. **Risk Assessment**:
   - **Execution Module → Risk Engine**: Logs anomalies.
5. **Reporting**:
   - **Risk Engine → Report Builder**: Sends scored findings.

---

3. Evaluation Criteria

1. **Detection Accuracy**: Compare AI-predicted vulnerabilities against manual audits.
2. **False Positives**: Percentage of flagged issues that are non-exploitable.
3. **Test Effectiveness**: Success rate of generated payloads in triggering vulnerabilities.
4. **Report Utility**: Feedback from security teams on actionability.

---

4. Ethical and Operational Considerations

- **Authorization**: Only test systems with explicit permission.
- **Data Handling**: Anonymize sensitive data (e.g., API keys) during analysis.
- **Model Bias**: Regularly audit AI predictions for false positives/negatives.

---

6. Tools and Resources (e.g.)

- **Crawling**: PyGithub, Swagger-Parser, etc.

- **AI Models**: Hugging Face Transformers, OpenAI API, etc.
- **Testing**: OWASP ZAP, Burp Suite, etc.
- **Reporting**: Jinja2, Pandas, etc.