Project Title: Autonomous Adaptive Exploit Generation System (AAEGS) Using LLMs

---

1. Project Overview

The **Autonomous Adaptive Exploit Generation System (AAEGS)** aims to automate the creation of proof-of-concept (PoC) exploits for identified vulnerabilities in networked assets. Leveraging Large Language Models (LLMs), the system integrates asset discovery, vulnerability aggregation, and adaptive exploit generation, iterating until successful exploitation. This project addresses the challenges of rapid vulnerability response and ethical penetration testing.

---

2. Objectives

- **Automated Asset-to-Exploit Pipeline**: Transform raw asset data (i.e., IPs) into validated proof-of-concept (PoC) exploits.
- **Adaptive Exploit Generation**: Use LLMs to iteratively refine exploit strategies based on failure feedback.
- **Comprehensive Vulnerability Mapping**: Correlate network services with vulnerabilities (CVEs, CWEs, KEV) and attack patterns (CAPEC).
- **Validate exploits** in **controlled environments** to ensure safety and efficacy.
- **Ethical Automation**: Ensure compliance with legal and safety standards for penetration testing.

---

3. System Components

Component 1: Asset Discovery & Service Profiler

- **Input**: IP addresses/ranges.
- **Process**:
    - **Port Scanning**: Use *Nmap/Masscan* to detect open ports.
    - **Service Fingerprinting**: Identify service types (e.g., HTTP, FTP, etc) and versions (e.g., OpenSSH 8.2p1).
- **Output**: Structured list of services per IP (e.g., `{IP: 10.0.0.1, Service: Apache 2.4.49}`).

Component 2: Vulnerability Intelligence Aggregator

- **Input**: Service profiles (from Component 1).
- **Process**:
    - Query **NVD**, **MITRE CVE/KEV**, and **CAPEC** databases for vulnerabilities tied to service versions.
    - Prioritize CVEs using **CVSS scores** and **KEV exploitation likelihood**.
- **Output**: Curated list of vulnerabilities (CVE IDs, descriptions, exploit prerequisites).

Component 3: LLM-Based Exploit Generator

- **Input**: Vulnerability data (from Component 2).
- **Process**:
    - **Prompt Engineering**: Feed LLMs (e.g., GPT-4, CodeLlama) with structured prompts:
        - *"Generate Python code to exploit CVE-2023-XXXX (buffer overflow in Service Y v2.1). Use stack overflow with ASLR bypass."*
        - Include code examples from similar CVEs for context.
    - **Code Validation**: Parse LLM output into executable scripts; flag syntax errors.
- **Output**: Candidate exploit code (e.g., Python, Ruby, or C payloads).

Component 4: Exploit Validator & Sandbox

- **Input**: Candidate exploit code (from Component 3).

- **Process**:
    - **Environment Replication**: Deploy Docker containers mirroring the target service (e.g., Apache 2.4.49).
    - **Execution & Monitoring**: Run the exploit and observe outcomes (e.g., shell access, service crash, etc).
    - **Static/Dynamic Analysis**: Use *Semgrep* (static) and *GDB/Strace* (dynamic) to detect flaws.
- **Output**: Exploit success/failure status; debug logs.

Component 5: Adaptive Feedback Engine

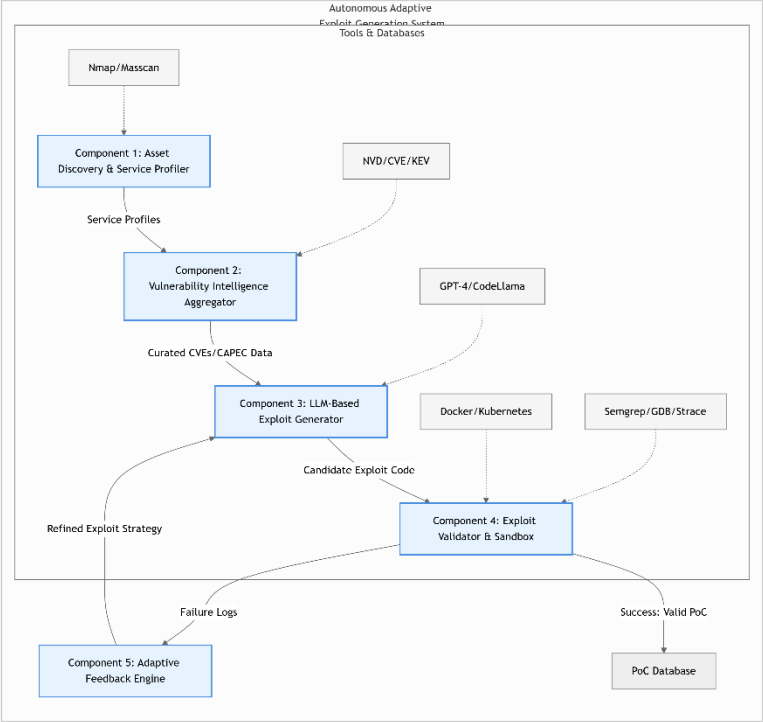
- **Input**: Exploit failure logs (from Component 4).
- **Process**:
    - **Root Cause Analysis**: Identify failure reasons (e.g., incorrect offset, WAF detection).
    - **Strategy Adjustment**:
        - Modify payloads (e.g., encode shellcode, adjust buffer size, etc).
        - Switch techniques (e.g., replace ROP chain with ret2libc).
    - **LLM Retraining**: Update prompts with failure context (e.g., "*Previous payload triggered IDS; suggest obfuscation*").
- **Output**: Refined exploit code for re-testing.

---

4. Workflow Cycle

1. **Scan IPs** → **Identify Services** (Component 1).
2. **Map Services** → **CVEs** (Component 2).
3. **Generate Exploit Code** (Component 3).
4. **Test Exploit in Sandbox** (Component 4).
    - **If Success**: Store PoC; report.
    - **If Failure**: Pass logs to Component 5 → Revise exploit → Retest.
5. Repeat steps 3–4 until exploitation succeeds or a termination threshold (e.g., 5 iterations) is met.

5. Component Diagram



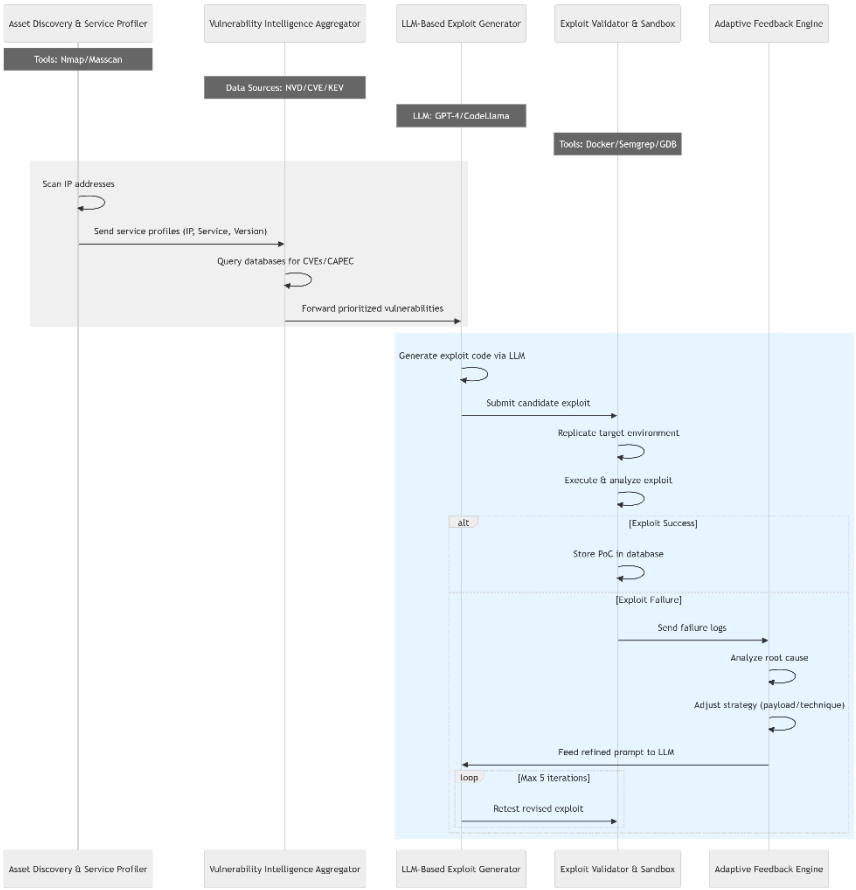
Key Elements Explained:

1. **Core Components (Blue Nodes):**
    - Component 1: IP scanning → service fingerprinting
    - Component 2: Vulnerability prioritization using CVSS/KEV
    - Component 3: LLM-driven code generation with security prompts
    - Component 4: Sandboxed exploit validation with analysis tools
    - Component 5: Feedback loop for exploit refinement
2. **External Tools/Databases (Gray Nodes):**
    - Scanning tools (Nmap), vulnerability databases (NVD), LLMs (GPT-4), and analysis tools (Semgrep)
3. **Workflow Logic:**
    - Solid arrows: Primary data flow between components
    - Dashed arrows: Tool/database dependencies
    - Feedback loop: Failed exploits trigger adaptive strategy updates
4. **Output:**

- Successful PoCs stored in a database for reporting
- Continuous iteration until exploit succeeds or threshold reached

---

6. Sequence Diagram



Key Annotations:

1. **Preparation Phase (Gray Section):**
   - Component 1 (`Asset Discovery`) scans IPs and shares service profiles.
   - Component 2 (`Vulnerability Aggregator`) maps services to CVEs using external databases.
2. **Exploit Lifecycle (Blue Section):**
   - Component 3 (`LLM Generator`) creates initial exploit code.
   - Component 4 (`Validator`) tests the exploit in an isolated sandbox.
   - **Success**: PoC is stored; process terminates.
   - **Failure**: Component 5 (`Feedback Engine`) refines the strategy and triggers retries (max 5 iterations).

7. Technical Specifications (e.g.)

- **LLM Configuration**:
  - Model: Fine-tuned GPT-4 with cybersecurity corpus (CVE descriptions, exploit-db scripts).
  - Constraints: Code output restricted to safe functions (no real-world harm).
- **Sandbox Environment**:
  - Docker/Kubernetes cluster with pre-built vulnerable service images.
  - Network isolation to prevent accidental external exposure.

---

8. Evaluation Framework

Metric	Measurement
Exploit Success Rate	% of CVEs with working PoCs after ≤5 cycles.
Time-to-Exploit	Avg. minutes from CVE identification to PoC.
LLM Accuracy	% of syntactically valid, executable code.
Adaptation Efficiency	Avg. iterations needed per successful exploit.

---

9. Ethical & Safety Considerations

- **Controlled Testing**: Restrict exploits to sandboxed environments.
- **Compliance**: Adhere to penetration testing laws (e.g., written consent for target IPs).
- **Bias Mitigation**: Audit LLM outputs to prevent harmful code generation.

---

10. Challenges & Mitigations (optional)

- **LLM Hallucinations**: Validate code with SAST tools.
- **Resource Intensity**: Optimize parallel testing via Kubernetes clusters.
- **False Positives**: Pre-scan assets to confirm vulnerability presence.
