Project Title: Automated Fuzzing for Vulnerability Discovery

Short Project Description:

An automated fuzzing system for discovering software vulnerabilities. The system combines input generation, execution monitoring, crash analysis, and reporting to identify security flaws in target applications.

Component	Role
Input Generator (Mutator)	Generates or mutates test inputs
Instrumented Execution Monitor	Runs app and watches for crashes/bugs
Crash Analyzer	Categorizes crashes (e.g., heap overflow, use-after-free)
Test Case Minimizer	Shrinks crash cases to minimal reproducer
Reporting Engine	Summarizes fuzzing outcomes and discovered vulnerabilities

---

Component Details:

1. **Input Generator (Mutator)**:
    - Generates test inputs via:
        - Random mutations
        - Structure-aware mutations (e.g., JSON format, etc).
        - Etc
2. **Instrumented Execution Monitor**:
    - Runs program with coverage-guided instrumentation:
        - Tools like **AFL++**, **libFuzzer**, **Honggfuzz**, etc.
3. **Crash Analyzer**:
    - Diagnoses crashes:
        - Buffer overflow
        - Invalid memory access
        - Undefined behavior
        - Etc
4. **Test Case Minimizer**:
    - Trims fuzzed input to minimum bytes causing crash (for easier debugging).
5. **Reporting Engine**:
    - Maps crashes to input files and root cause.

Overall System Flow:

- Input: Target executable or library
- Output: Crash reports and bug classification
- Focus: **Automated discovery via fuzzing**.

Internal Functioning of Each Module:

1. Input Generator (Mutator)

- **Random Mutations**:
  - Flip random bits, insert/delete bytes, reorder bytes, etc.
- **Structure-Aware Mutations**:
  - For known formats (like JSON, XML, PNG, etc):
    - Maintain syntactic validity while mutating fields.
- **Grammar-Based Fuzzing**:
  - Use input grammars (e.g., for a proprietary protocol) to generate valid inputs.

---

2. Instrumented Execution Monitor

- **Coverage-Guided Fuzzing**:
  - Tools like **AFL++**:
    - Instrument code at compile-time or runtime.
    - Measure code coverage (basic block/edge coverage).
    - Prioritize inputs that explore new code paths.
- **Forkserver Mode**:
  - Fast cloning of process to fuzz without heavy startup costs.

---

3. Crash Analyzer

- **Crash triaging**:
  - Use **gdb**, **coredumps**, or sanitizers to:
    - Identify crash types:
      - Segfaults (null pointer dereference)
      - Heap corruption
      - Stack smashing
      - Etc.
- **Heuristic ranking**:
  - Prioritize crashes likely to lead to code execution.

---

4. Test Case Minimizer

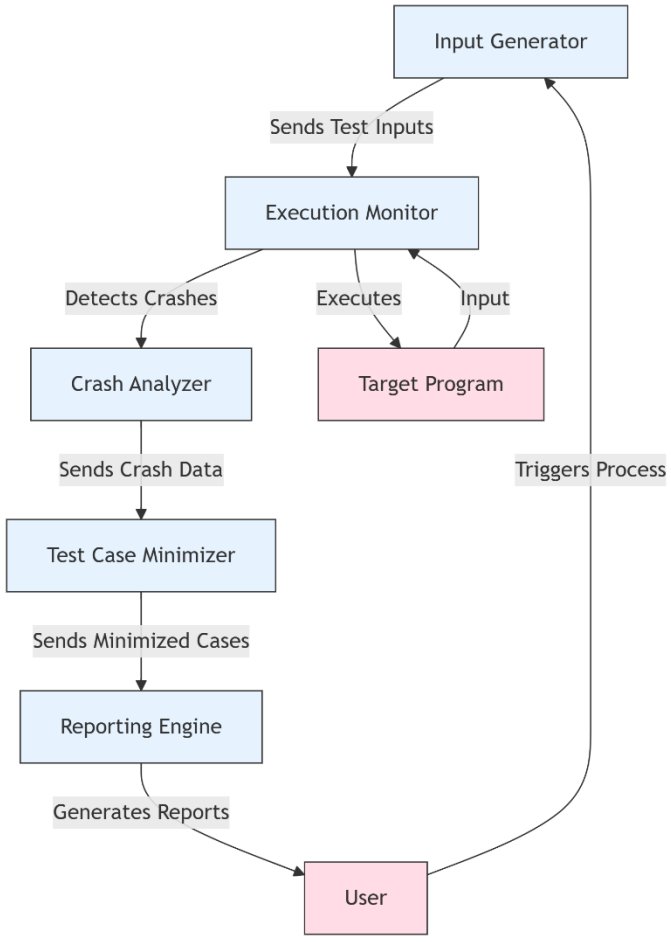
- **Delta Debugging**:
  - Minimize fuzzing input by:
    - Iteratively removing parts and testing if the crash still occurs.
- **Minimization tools**:
  - AFL-tmin
  - libFuzzer's minimization mode

---

5. Reporting Engine

- **Detailed crash reports**:
  - Input file
  - Crash reason
  - Exploitability score
  - Suggested code fix

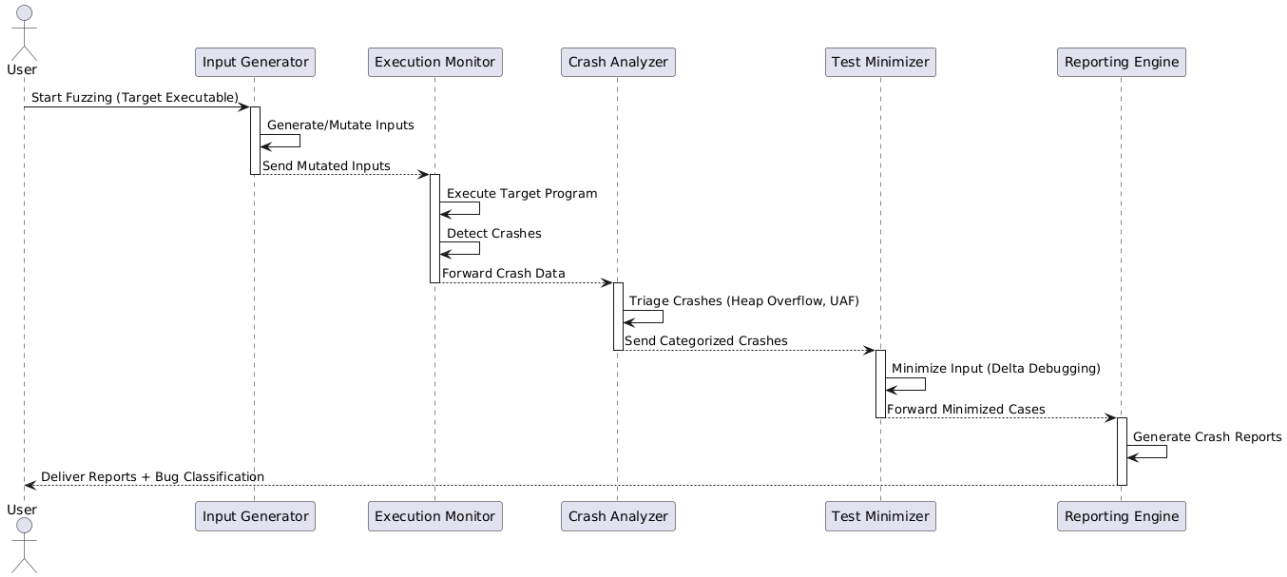
Component Diagram



- **Input Generator** mutates inputs (random/structure-aware) and sends them to the **Monitor**.
- The **Monitor** runs the target program, detects crashes, and forwards data to the **Crash Analyzer**.
- The **Analyzer** triages crashes and sends them to the **Minimizer** for reduction.

- The **Minimizer** shrinks test cases and passes them to the **Reporting Engine** for final reports.

Sequence Diagram



1. **User** triggers the fuzzing process.
2. **Input Generator** creates test cases and sends them to the **Monitor**.
3. **Monitor** executes the target program and forwards crashes to the **Analyzer**.
4. **Analyzer** categorizes crashes (e.g., heap overflow) and sends them to the **Minimizer**.
5. **Minimizer** reduces inputs and sends them to the **Reporter**.
6. **Reporter** generates reports and delivers them to the user.

Detailed Project Description: Automated Fuzzing for Vulnerability Discovery

An automated fuzzing system for discovering software vulnerabilities. The system combines input generation, execution monitoring, crash analysis, and reporting to identify security flaws in target applications.

---

1. System Components and Roles

1.1 Input Generator (Mutator)

Purpose: Generate or mutate test inputs to explore edge cases in the target application.
Implementation Details (e.g.):

- **Mutation Strategies**:
  - **Random Mutations**: Flip bits, insert/delete bytes, shuffle input segments.
  - **Structure-Aware Mutations**: Preserve syntax for formats like JSON, XML, or PNG.
  - **Grammar-Based Fuzzing**: Use protocol specifications (e.g., HTTP, TLS, etc) to generate valid inputs.
- **Tools**:
  - **AFL++** (custom mutators via `afl-fuzz`).
  - **libFuzzer** (in-process fuzzing with seed corpus).
  - **radamsa** (generic input mutator).
  - **Etc**
- **Example Command**:
  ```
  # Generate mutated inputs with AFL++
  afl-fuzz -i input_seeds -o output_dir -- ./target @@
  ```

1.2 Instrumented Execution Monitor

Purpose: Run the target program with instrumentation to detect crashes and track code coverage.
Implementation Details (e.g.):

- **Coverage-Guided Fuzzing**:
  - **AFL++**: Compile target with instrumentation:
    ```
    export AFL_USE_ASAN=1
    ./configure --disable-shared CC=afl-clang-fast
    make
    ```
  - **libFuzzer**: Link target with libFuzzer library:
    ```
    clang -fsanitize=fuzzer,address -o fuzzer fuzz_target.c
    ```
- **Execution Modes**:
  - **Forkserver**: Speeds up fuzzing by reusing process state (default in AFL++).
  - **Persistent Mode**: Reuse a single process for multiple inputs (libFuzzer).

1.3 Crash Analyzer

Purpose: Diagnose crash root causes (e.g., buffer overflow, use-after-free, etc).
Implementation Details (e.g.):

- **Tools**:
  - **GDB**: Debug crashes using core dumps:
    ```
    gdb ./target core
    ```
  - **AddressSanitizer (ASan)**: Compile target with ASan to detect memory errors:
    ```
    clang -fsanitize=address -g -o target target.c
    ```
- **Crash Triage**:
  - Classify crashes by type (e.g., SIGSEGV, SIGABRT, etc).
  - Prioritize exploitable crashes using **Exploitable** (GDB plugin).

1.4 Test Case Minimizer

Purpose: Reduce crash-inducing inputs to minimal reproducible examples.
Implementation Details (e.g.):

- **Tools**:
  - **AFL-tmin**: Minimize inputs while preserving crash behavior:
    ```
    afl-tmin -i crash_input -o minimized_input -- ./target @@
    ```
  - **libFuzzer's** `-minimize_crash`: Automatically reduce test cases.
- **Delta Debugging**: Iteratively remove non-essential bytes from input.

1.5 Reporting Engine

Purpose: Generate actionable reports summarizing vulnerabilities.
Implementation Details (e.g.):

- **Tools**:
    - **Python scripts** to parse crash logs and generate HTML/PDF reports.
    - **Elastic Stack** (Elasticsearch, Kibana) for dashboard visualization.
    - **Etc**
- **Report Metrics**:
    - Crash type (e.g., heap overflow, etc).
    - Exploitability score (High/Medium/Low, etc).
    - Affected code location (file, line number, etc).

---

2. System Integration and Workflow

2.1 Component Interaction

1. **Input Generation**:
    - The **Mutator** generates test cases (random/structured) and feeds them to the **Monitor**.
2. **Execution Monitoring**:
    - The **Monitor** runs the target program, detects crashes, and logs coverage data.
3. **Crash Analysis**:
    - The **Analyzer** triages crashes using GDB/ASan and forwards them to the **Minimizer**.
4. **Test Case Minimization**:
    - The **Minimizer** reduces inputs to minimal reproducers.
5. **Reporting**:
    - The **Reporting Engine** aggregates data into a structured report.

---

3. Implementation Steps (e.g.)

3.1 Environment Setup

- **OS**: Ubuntu 22.04 LTS (recommended for tool compatibility).
- **Dependencies**:

```
sudo apt install afl++ clang gdb python3-pip
pip install exploitable jinja2
```

3.2 Target Instrumentation

- **AFL++**:

```
export AFL_USE_ASAN=1
afl-clang-fast -o target target.c
```

- **libFuzzer**:

```
clang -fsanitize=fuzzer,address -o fuzzer fuzz_target.c
```

3.3 Fuzzing Execution

- **AFL++ Command**:

```
afl-fuzz -i input_seeds -o findings -- ./target @@
```

- **libFuzzer Command**:

```
./fuzzer -artifact_prefix=./crashes/
```

3.4 Crash Minimization

- **AFL-tmin**:

```
afl-tmin -i findings/crashes/id:000000 -o minimized -- ./target @@
```

3.5 Report Generation

- **Python Script Example**:

```
import json
from jinja2 import Template
crash_data = {
    "type": "heap-overflow",
    "file": "target.c:42",
    "exploitability": "High"
}
```

```
template = Template(open("report_template.html").read())
with open("report.html", "w") as f:
    f.write(template.render(crash=crash_data))
```

---

4. Evaluation Criteria

1. **Crash Detection Rate**: Percentage of known vulnerabilities (CVEs) detected.
2. **Code Coverage**: Measure edge/basic block coverage (e.g., using `afl-cov`).
3. **False Positives**: Manual review of reported crashes.
4. **Minimization Efficiency**: Reduction ratio of crash inputs (e.g., 10 MB → 100 bytes).

---

5. Ethical Considerations

- **Authorization**: Only fuzz software you own or have explicit permission to test.
- **Safe Environment**: Run fuzzing in isolated VMs/containers to prevent system instability.

---

6. Tools and Resources (e.g.)

- **Fuzzing Frameworks**: AFL++, libFuzzer, Honggfuzz, etc.
- **Crash Analysis**: GDB, AddressSanitizer, Exploitable, etc.
- **Minimization**: AFL-tmin, libFuzzer's minimization mode, etc.
- **Reporting**: Jinja2, Elastic Stack, etc.
