Project Title: Software Vulnerability Discovery via Static and Dynamic Analysis

---

Short Project Description:

A hybrid vulnerability discovery system combining static and dynamic analysis. The system identifies coding flaws, unsafe patterns, and runtime vulnerabilities in both source code and compiled binaries.

Component	Role
Source Code Static Analyzer	Finds potential coding flaws in source
Binary Static Analyzer	Finds flaws even without source code
Dynamic Instrumentation Module	Monitors program behavior during runtime
Vulnerability Pattern Matcher	Detects known vulnerability patterns
Reporting Engine	Generates vulnerability discovery reports

Component Details:

1. **Source Code Static Analyzer**:
   - Parses code ASTs, CFGs.
   - Flags risky functions (`strcpy`, `system`, `malloc` misuse, etc).
2. **Binary Static Analyzer**:
   - Disassembles binaries.
   - Looks for unsafe code patterns without needing source.
3. **Dynamic Instrumentation Module**:
   - Runs target under tools like:
     - **Valgrind** (memory errors)
     - **AddressSanitizer** (heap/stack violations)
     - **Etc**
4. **Vulnerability Pattern Matcher**:
   - Uses rule engines (YARA, custom regex, etc) to match vulnerabilities.
5. **Reporting Engine**:
   - Lists suspicious functions/areas for human review.

Overall System Flow:

- Input: Source or compiled binary
- Output: Potential vulnerability locations
- Focus: **Hybrid static and dynamic vulnerability discovery**.

Internal Functioning of Each Module:

1. Source Code Static Analyzer

- **Parsing source**:
  - Builds **Abstract Syntax Tree (AST)**:
    - Nodes = programming constructs (functions, loops, if-else, etc.)
  - Builds **Control Flow Graph (CFG)**:
    - Paths through code execution.
- **Analysis techniques**:
  - **Taint Analysis**:
    - Track input data (e.g., user input) through the code.
    - Flag unsafe flows (e.g., user input → system command).
  - **Pattern Matching**:
    - Regex over source code for dangerous APIs:
      - strcpy(), gets(), sprintf(), etc.
- **Risk prioritization**:
  - Rank findings based on function criticality and input control.

---

2. Binary Static Analyzer

- **Binary disassembly**:
  - Use **Ghidra**, **Radare2**, **IDA Pro**, etc to:
    - Convert machine code into assembly or pseudo-code.
- **Signature Detection**:
  - Look for:
    - Shellcode patterns
    - Stack-based buffer operations (e.g., mov [ebp-4], eax)
    - Etc
- **Function boundary detection**:
  - Heuristics to separate functions without debug symbols.

---

3. Dynamic Instrumentation Module

- **Runtime monitoring tools**:
    - **Valgrind (Memcheck)**:
        - Detects:
            - Invalid memory reads/writes
            - Use-after-free
            - Memory leaks
    - **AddressSanitizer (ASan)**:
        - Compile-time instrumentation for runtime error detection.
- **Hooks**:
    - Intercept function calls (especially malloc/free, file I/O, etc).

---

4. Vulnerability Pattern Matcher

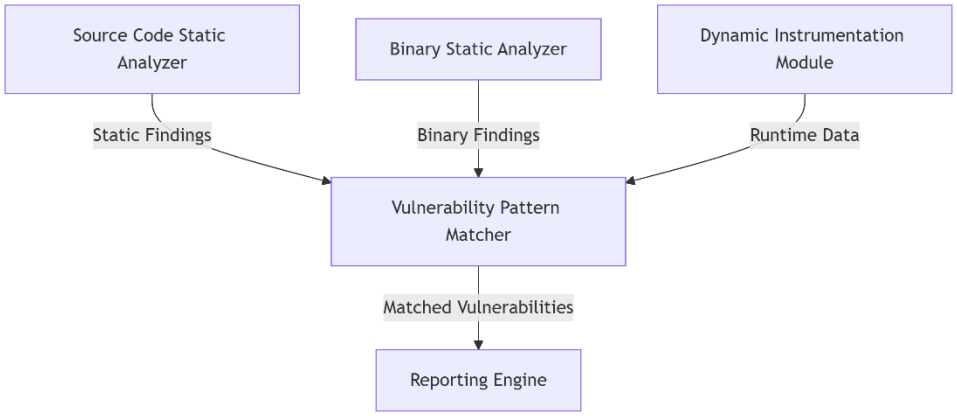
- **Pattern Database**:
    - YARA rules for memory vulnerabilities.
    - Custom regex for identifying vulnerability behaviors.
- **Advanced**:
    - Use ML-based classifiers for novel vulnerability pattern detection.

---

5. Reporting Engine

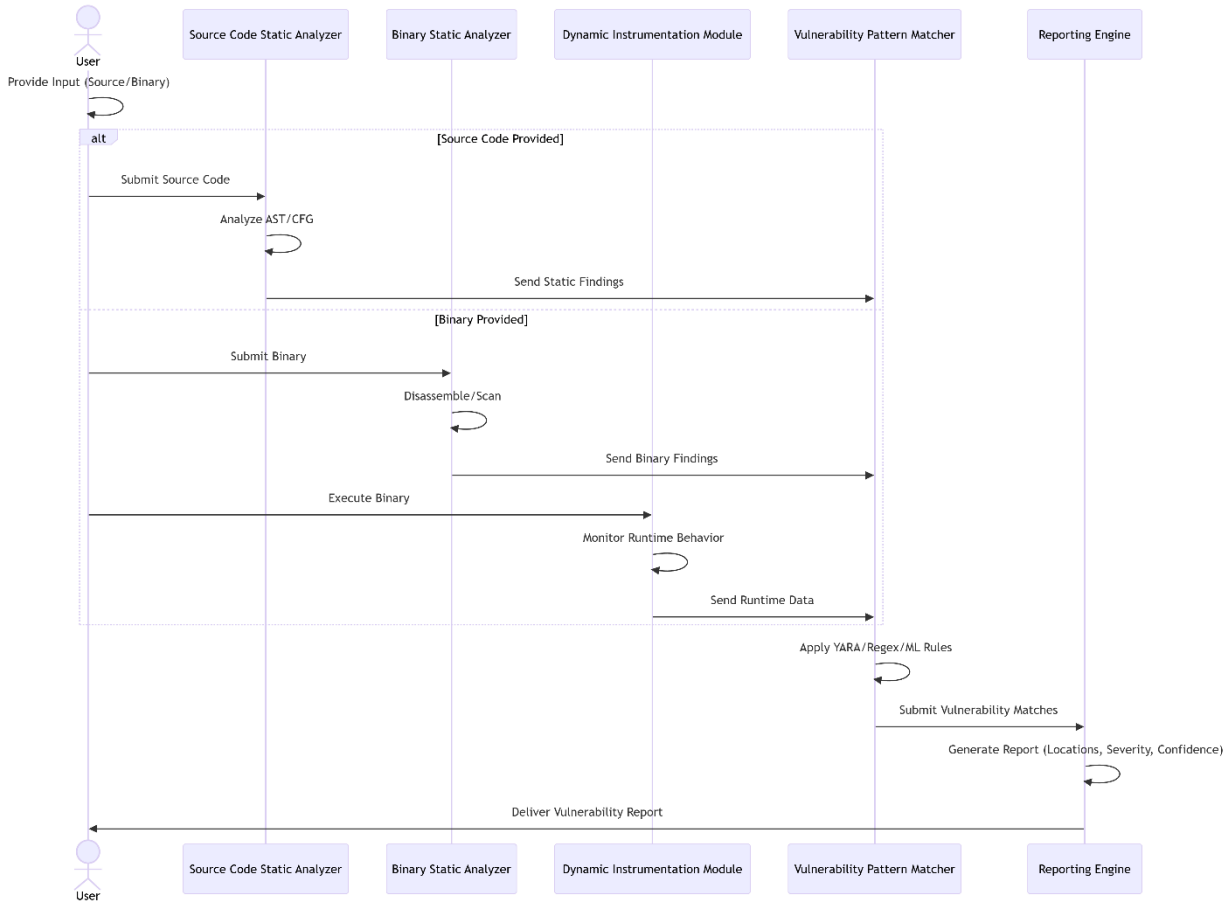
- **Report contents**:
    - Function locations
    - Vulnerability type
    - Severity rating
    - Confidence score (based on static vs dynamic evidence)
    - Etc

Component Diagram



- **Source Code Static Analyzer** and **Binary Static Analyzer** feed static analysis results (e.g., risky functions, disassembly insights, etc) into the **Vulnerability Pattern Matcher**.
- **Dynamic Instrumentation Module** provides runtime data (e.g., memory leaks, heap violations, etc) to the **Vulnerability Pattern Matcher**.
- **Vulnerability Pattern Matcher** correlates findings from all sources using rules (YARA, regex, etc) or ML, then sends confirmed vulnerabilities to the **Reporting Engine**.
- **Reporting Engine** compiles all findings into a structured report.

Sequence Diagram



- **User** provides either source code or a compiled binary.
    - **Source Code Path**:
        - Source Code Static Analyzer parses AST/CFG and flags vulnerabilities (e.g., `strcpy` misuse, etc).
        - Findings are sent directly to the Vulnerability Pattern Matcher.
    - **Binary Path**:
        - Binary Static Analyzer disassembles the binary and identifies unsafe patterns.
        - Dynamic Instrumentation Module runs the binary with tools like Valgrind/ASan to capture runtime issues.
        - Both modules send results to the Vulnerability Pattern Matcher.
- **Vulnerability Pattern Matcher** applies detection rules (e.g., YARA for shellcode, etc) to prioritize and validate findings.
- **Reporting Engine** aggregates results, assigns severity/confidence scores, and delivers the final report to the user.

Detailed Project Description: Software Vulnerability Discovery via Static and Dynamic Analysis

A hybrid vulnerability discovery system combining static and dynamic analysis. The system identifies coding flaws, unsafe patterns, and runtime vulnerabilities in both source code and compiled binaries.

---

1. System Components and Roles

1.1 Source Code Static Analyzer

Purpose: Analyze source code for coding flaws using AST and CFG analysis.
Implementation Details (e.g.):

- **Tools**:
  - **Clang Static Analyzer** (C/C++) or **Bandit** (Python) for AST parsing.
  - **Joern** (for advanced CFG and taint analysis).
- **Key Checks**:
  - Risky functions (`strcpy`, `system`, `malloc` misuse, etc).
  - Taint tracking (e.g., user input reaching `exec()`).
- **Setup (e.g.)**:

```
# Install Clang Static Analyzer
sudo apt install clang clang-tools
# Run analysis
clang --analyze -Xanalyzer -analyzer-output=text vulnerable_code.c
```

1.2 Binary Static Analyzer

Purpose: Identify vulnerabilities in compiled binaries without source code.
Implementation Details (e.g.):

- **Tools**:
  - **Ghidra** (open-source disassembler).
  - **Radare2** (for shellcode detection).
  - **Etc**

- **Key Checks**:
  - Stack-based buffer operations (e.g., `mov [ebp-4], eax`).
  - Function boundary detection (heuristics for stripped binaries).
- **Setup (e.g.)**:

```
# Install Ghidra
wget https://ghidra-sre.org/ghidra_10.1.2_PUBLIC_20220125.zip
unzip ghidra_10.1.2_PUBLIC_20220125.zip
./ghidraRun
```

1.3 Dynamic Instrumentation Module

Purpose: Detect runtime vulnerabilities like memory leaks and heap corruption.

Implementation Details (e.g.):

- **Tools**:
  - **Valgrind** (memory errors):

```
valgrind --leak-check=full ./target_binary
```

  - **AddressSanitizer (ASan)** (compile with instrumentation):

```
gcc -fsanitize=address -g vulnerable_code.c -o vulnerable
./vulnerable
```

  - **Etc**
- **Hooks**: Monitor `malloc`, `free`, and file I/O using `LD_PRELOAD` or `ptrace`.

1.4 Vulnerability Pattern Matcher

Purpose: Correlate static/dynamic findings with known vulnerability patterns.

Implementation Details (e.g.):

- **Tools**:
  - **YARA** (rule-based matching):

```
yara -r vuln_rules.yar ./target_binary
```

  - **Custom Regex** (e.g., detect `gets()` usage):

```
import re
with open("code.c") as f:
    if re.search(r'\bgets\s*\(', f.read()):
        print("Vulnerability: gets() detected!")
```

  - **Etc**
- **Advanced**: Train ML models (e.g., scikit-learn) on labeled vulnerability datasets.

1.5 Reporting Engine

Purpose: Generate structured vulnerability reports.
Implementation Details (e.g.):

- **Tools**:
    - **Python + Pandas** for data aggregation.
    - **Jinja2** or **LaTeX** for report templating.
    - **Etc**
- **Metrics**:
    - Severity (Critical/High/Medium/Low).
    - Confidence score (e.g., 90% for dynamic-confirmed issues).
    - Etc

---

2. System Integration and Workflow

2.1 Component Interaction

1. **Static Analysis Path**:
    - **Source Code**:
        - The **Source Code Static Analyzer** parses AST/CFG → outputs risky functions → **Vulnerability Pattern Matcher**.
    - **Binary**:
        - The **Binary Static Analyzer** disassembles code → identifies unsafe patterns → **Vulnerability Pattern Matcher**.
2. **Dynamic Analysis Path**:
    - The **Dynamic Instrumentation Module** runs Valgrind/ASan → logs runtime issues → **Vulnerability Pattern Matcher**.
3. **Correlation**:
    - The **Vulnerability Pattern Matcher** applies YARA rules or ML models to prioritize findings → sends results to the **Reporting Engine**.
4. **Reporting**:

- The **Reporting Engine** compiles a PDF/HTML report with vulnerability locations, severity, and remediation steps.

---

3. Implementation Steps (e.g.)

3.1 Environment Setup

- **OS**: Ubuntu 22.04 LTS (recommended for tool compatibility).
- **Dependencies**:
```
sudo apt install valgrind gcc clang yara python3-pip
pip install pandas jinja2 scikit-learn
```

3.2 Static Analysis Configuration

- **Source Code**:
  - Create a script to run Clang/Bandit on all source files:
```
find . -name "*.c" -exec clang --analyze {} \;
```
- **Binary**:
  - Use Ghidra's headless mode for automated disassembly:
```
./analyzeHeadless /path/to/project -import /path/to/binary -postScri
pt analysis_script.py
```

3.3 Dynamic Analysis Configuration

- **Compile with ASan**:
```
gcc -fsanitize=address -g -o target target.c
```
- **Run Valgrind**:
```
valgrind --track-origins=yes --log-file=valgrind.log ./target
```

3.4 Vulnerability Pattern Database

- **YARA Rules**:
```
rule shellcode {
  strings:
    $shellcode = { 31 c0 50 68 2f 2f 73 68 }
  condition:
    $shellcode
```

```
}
```

3.5 Report Generation

- **Python Script Example**:

```
import pandas as pd
from jinja2 import Template
data = pd.read_csv("vulns.csv")
template = Template(open("report_template.html").read())
with open("report.html", "w") as f:
    f.write(template.render(vulns=data))
```

---

4. Evaluation Criteria (e.g.)

1. **Detection Rate**: Percentage of known vulnerabilities identified (e.g., CVE test cases).
2. **False Positives**: Manual review to flag incorrect alerts.
3. **Performance**: Time taken for analysis (e.g., static vs dynamic).

---

5. Ethical Considerations

- Use only on authorized codebases.
- Avoid testing third-party software without permission.

---

6. Tools and Resources (e.g.)

- **Static Analysis**: Clang, Ghidra, Bandit, etc.
- **Dynamic Analysis**: Valgrind, AddressSanitizer, etc.
- **Pattern Matching**: YARA, scikit-learn, etc.
- **Reporting**: Pandas, Jinja2, etc.
