In the name of God

Sharif University of Technology
Department of Computer Engineering

CE 815: Secure Software Systems

Mehdi Kharrazi *Dey 17st, 1404*

Project: Agentic Vulnerability Detection

Overview

Traditional vulnerability detection systems use fixed pipelines where every code sample goes through the same sequence of steps, loading all code upfront before analysis. This approach is inefficient and doesn't mirror how human security auditors work—experts examine code iteratively, requesting specific information as needed to build understanding incrementally.

This project implements an **agentic vulnerability detection system** that performs iterative context retrieval, making dynamic decisions about what code to examine and when sufficient information has been gathered. The agent analyzes code, decides if it needs more information, requests specific functions or callers through tools, and repeats until reaching a conclusion.

What is an agent? An agent maintains state, calls tools, and makes decisions:

```python
def run(self, target):
    context = self.load_initial_context(target)

    for round in range(MAX_ROUNDS):
        response = self.llm.query(context)  # LLM analyzes and returns JSON
        decision = parse_json(response)      # Parse LLM's JSON response

        if decision["type"] == "tool":
            # LLM decided it needs more information
            tool_result = self.tools.call(decision["tool"], decision["params"])
            context.add(tool_result)
        elif decision["type"] == "final":
            # LLM made a decision
            return decision["verdict"]

    return "UNKNOWN"  # Max rounds reached
```

The core concepts are: (1) context management across rounds, (2) tool calling based on LLM decisions, and (3) determining when to conclude analysis.

How the Project Develops

The project develops incrementally across two parts. **Part 1** builds the complete agent core with basic tools using grep and regex. Once the agent works, **Part 2** adds CodeQL for production-grade analysis without modifying the agent—only adding new tools. Both tool sets remain available to the agent—the LLM chooses which tool to use based on the analysis task. The key principle is building the agent once and extending it by adding tools, demonstrating proper abstraction where the agent's decision-making logic remains unchanged while analysis capabilities grow.

∗Acknowledgement: This homework was developed by Mohammad Hadadian

Iterative Analysis Example

Consider analyzing a buffer overflow vulnerability. Instead of loading all code at once, the agent works iteratively:

```
Round 1: Agent examines buffer_copy() function
         LLM: "I see strcpy, but need to check who calls this"
         Agent requests: find_callers(buffer_copy)

Round 2: Agent now has buffer_copy() + caller process_input()
         LLM: "Caller passes user input, need to see the source"
         Agent requests: get_function(process_input)

Round 3: Agent has full chain: main -> process_input -> buffer_copy
         LLM: "User input from argv[1] flows to strcpy with no bounds check"
         Agent concludes: VULNERABLE, CWE-120 Buffer Overflow
```

This iterative context retrieval is the core learning objective.

Part 1: Agent Core with Basic Tools

The first part establishes the foundation by implementing a complete agentic system. The agent must perform iterative analysis where it loads initial code context, sends it to the LLM (VulnLLM-R-7B) for analysis, receives either a tool request or final decision, executes any requested tools, accumulates the results into growing context, and repeats this process for up to three rounds. This iterative loop is the core learning objective—the agent doesn't load everything upfront but builds understanding incrementally.

Implementation Components:
First, `agent.py` implements the `VulnerabilityAgent` class with methods for iterative analysis (`analyze_target`), loading initial context from target files, building system and user prompts that explain the agent's role and available tools, parsing LLM responses to extract JSON tool requests or decisions, executing requested tools and accumulating results, and formatting final verdicts with evidence.

Second, `simple_tools.py` provides basic code analysis capabilities: extracting functions by name using regex, finding dangerous patterns like strcpy and system calls, identifying function callers using grep, and locating functions at specific line numbers.

Third, `prompts.py` defines templates for the system prompt (explaining agent role, available tools, and JSON protocol) and user prompt (providing target code, context, and tool history).

Testing: Use vulnerable code samples from HW3 or create similar examples that require multi-round analysis where the agent must request callers or related functions to build complete attack paths.

Part 2: CodeQL Integration and Verdict System

Part 2 adds production-grade analysis using CodeQL, GitHub's code analysis engine. **Important:** CodeQL finds potential vulnerabilities and outputs them in SARIF format. The agent then uses the LLM (VulnLLM-R-7B) to reason about each finding—the LLM examines code, requests additional context through tools, and makes the final decision. Your VerdictSystem classifies the LLM's reasoning into structured verdicts with confidence scores. Understanding CodeQL's workflow is essential for this part.

What is CodeQL? CodeQL transforms source code into a queryable database. The process involves three steps:

1. **Database Creation:** Run `codeql database create` to build a database from your codebase
2. **Query Execution:** Write queries in `.ql` files using CodeQL's declarative language to find patterns, trace data flow, and identify vulnerabilities
3. **Results Processing:** Run `codeql database analyze` which executes queries and outputs results in SARIF format (a JSON-based standard for static analysis results)

Unlike simple pattern matching, CodeQL understands code semantics and performs deep inter-procedural analysis. The provided `codeql_tools.py` handles the complexity of running CodeQL commands and parsing results.

Implementation Components:
First, `codeql_integration.py` implements several key functions:

- `setup_codeql_database()` - Creates and manages the CodeQL database
- `load_targets_from_sarif()` - Parses SARIF output to identify and prioritize analysis targets
- `enhance_agent_with_codeql()` - Adds CodeQL tools to the agent

Important: This file also contains your agent's `VerdictSystem` class, which is part of your agent logic (not CodeQL). The verdict system is a critical agent component that classifies findings into four categories:

- **VULNERABLE:** Confirmed exploitable with clear attack path
- **LIKELY:** Strong indicators but needs verification
- **UNLIKELY:** Appears protected or not exploitable
- **UNKNOWN:** Insufficient information to decide

Each verdict includes a confidence score (0.0 to 1.0) calculated based on evidence quality, presence of complete attack paths, and identified mitigations.

Second, `codeql_custom_queries.py` contains CodeQL query templates for specific vulnerabilities like buffer overflows and command injections. The scaffolding provides these query templates which you can complete using knowledge from HW3, or alternatively, you can implement dynamic query generation where the agent uses the LLM to generate queries during execution based on the analysis context.

Testing: Analyze a large project such as nginx (10K+ LOC). Demonstrate that SARIF-based target selection works correctly, prioritizing analysis based on CodeQL's initial scan. Show that the verdict system produces reasonable confidence scores reflecting evidence strength.

Deliverables

Submit a single comprehensive report (maximum 10 pages) covering both parts. The report structure should include:

System Architecture: Complete architecture showing how both parts integrate, with flowcharts or diagrams illustrating the agent's iterative loop and how different tools connect.

Implementation Details: Document the implementation approach for each part—explain the agent core and basic tools, and detail CodeQL integration and the verdict system.

Execution Traces: Provide at least one complete execution trace showing the agent's iterative analysis across multiple rounds, demonstrating context accumulation and tool calling decisions.

Comparative Analysis: Include test results from both parts, comparing capabilities as tools are added. Show how Part 2 improves over Part 1 and reaches production quality.

Real-World Analysis: Analyze a large real-world project in Part 2 and document the findings, verdict classifications, and confidence scores. This demonstrates the complete system on production code.

All code should be functional and demonstrate the core concepts. The agent must demonstrate iterative context retrieval where the LLM makes tool calling decisions rather than following a predetermined path. Feel free to extend beyond the provided TODOs if you identify improvements or additional capabilities.

Evaluation

Your implementation will be evaluated on code functionality and execution. The evaluation process includes:

Test Cases: Your code will be tested on multiple levels:

- Provided test cases from each part
- Hidden test cases containing known vulnerabilities
- Medium-sized projects (for Part 2)
- Analysis of real-world code samples

Execution Requirements: Code must be executable using the VulnLLM-R-7B model via the provided API. The agent must successfully demonstrate iterative context retrieval where the LLM makes tool calling decisions dynamically, not following a predetermined path.

Core Capabilities: The evaluation will verify that your agent:

- Performs multi-round iterative analysis
- Correctly parses LLM responses and executes tool requests
- Accumulates context across rounds
- Detects vulnerabilities in test cases
- Demonstrates proper tool integration in Part 2

Submit all source code along with the report. Ensure your implementation is well-documented and executable.

Resources

- **VulnLLM-R Paper:** `https://arxiv.org/abs/2512.07533`
- **API:** `[REDACTED]` (VulnLLM-R-7B model)
- **VPN:** SharifVPN required for API access
- **Documentation:** See `README.md` files in scaffolding directories
- **CodeQL:** `https://codeql.github.com/docs`