In the name of God

Sharif University of Technology
Department of Computer Engineering

CE 815: Secure Software Systems

Mehdi Kharrazi *Dey 1st, 1404*

Homework 3

LLM-Based Vulnerability Detection

Modern software systems contain millions of lines of code, making manual vulnerability analysis infeasible. Recently, Large Language Models (LLMs) have shown promising capabilities in reasoning about source code and identifying security vulnerabilities. However, directly applying LLMs to real-world codebases is challenging due to context window limitations, performance constraints, and high false-positive rates.

The goal of this assignment is to explore how LLMs can be effectively used as vulnerability detectors in realistic settings, where codebases exceed model context limits and security analysis must be both accurate and efficient. This homework focuses on the core problem of context preparation: deciding what code to show the LLM, how to structure it, and how to guide the analysis through carefully designed prompts.

Throughout this assignment, you will design, implement, and evaluate multiple strategies for preparing code context, constructing multi-stage analysis pipelines, and comparing different LLMs.

Assignment Workflow

This assignment follows a progressive workflow that builds your understanding of LLM-based vulnerability detection:

1. **Vulnerability Detection:** Test LLMs on provided vulnerable samples (simple to complex) using different context preparation strategies
2. **Multi-Stage Analysis:** Apply 3-stage iterative analysis pipeline (instead of single-shot prompting) to improve detection accuracy
3. **Patch Generation & Validation:** Generate valid patches that eliminate vulnerabilities without breaking functionality, then validate patches using LLM self-verification
4. **Real-World Application:** Apply your best approach to real open-source projects

Each stage builds on the previous, culminating in a robust vulnerability detection system ready for real-world use.

Background: The Context Window Problem

LLMs are limited by their context window (e.g., VulnLLM-R: $\sim$32K tokens $\approx$ 120KB code). Real projects exceed this limit. Your task is to design strategies that prepare code context intelligently before prompting the LLM.

Important: Even when all code fits within the context window, simply concatenating everything can confuse the LLM about where to focus its attention. Intelligent context preparation helps the model concentrate on security-relevant portions of the codebase.

To address the context window problem, we can consider different approaches for preparing code before sending it to the LLM. Understanding these approaches will help you design effective strategies in this assignment:

Three Common Approaches:

- **Naive Approach:** Simply concatenate all source files in the project and feed them to the LLM. This approach either exceeds the context window limit for large projects or, even when everything fits, causes the LLM to lose focus among irrelevant code sections.
- **Chain-of-Prompts Approach:** Break the analysis into multiple sequential stages, where each stage focuses on a specific sub-task. For example: first perform a high-level triage to identify suspicious locations, then conduct deep analysis on flagged areas, and finally verify findings with execution path reconstruction. Each stage's output guides the next stage's focus.
- **Heuristic Preprocessing Approach:** Apply static analysis techniques (pattern matching, call-graph analysis, data-flow analysis, etc.) to filter and extract only security-relevant code portions before prompting the LLM. This reduces context size while preserving critical information.

Throughout this assignment, you will implement and evaluate strategies based on these approaches to understand their trade-offs in real vulnerability detection scenarios.

1 Context Preparation Strategies (30 points)

In this part, you will implement different **context preparation strategies** to help the LLM focus on security-relevant code. A *strategy* is a systematic approach for selecting and organizing code from a target project before sending it to the LLM for vulnerability analysis.

The goal is to reduce irrelevant context while preserving enough information for the LLM to accurately detect vulnerabilities. Each strategy uses different heuristics to achieve this goal: some rely on syntactic patterns, others use program structure (call graphs), and some may use more sophisticated analysis techniques.

You will implement at least 4 strategies (3 required + 1 custom), evaluate their effectiveness on provided vulnerable samples, and analyze their trade-offs in terms of detection accuracy, context size, and execution time.

Required Strategies:

1. **Baseline:** Concatenate all code files (up to context limit)
2. **Pattern-based:** Extract only code matching Cpp-related vulnerability patterns, like:
   - Buffer overflow: `strcpy`, `gets`, `sprintf`
   - Command injection: `system`, `exec`, `popen`
   - Use-after-free: `free`, pointer dereferences after deallocation
3. **Call-graph based:** Build function call graph, extract paths from input sources to dangerous sinks

Additional Strategy (choose $\geq$ 1): Data flow analysis, complexity-based ranking, dependency-based grouping, or your own design.

Important: If implementing techniques from research papers or articles, you must cite the source.

Deliverable

Your `strategies/` directory should contain implementations of each strategy. Each strategy must:

- Accept a target directory path
- Prepare context according to the strategy's approach
- Return detected vulnerabilities in standardized format

Important: All prompts you use for LLM queries must be documented in your report for each strategy.

2 Multi-Stage Analysis Pipeline (25 points)

In this part, you will design and implement a **multi-stage analysis pipeline** that breaks vulnerability detection into sequential phases. Instead of asking the LLM to analyze all code at once (single-shot prompting), you will orchestrate multiple LLM calls where each stage has a focused purpose and builds upon the previous stage's output.

The key idea is to use a **chain-of-prompts approach**: start with a broad triage to identify suspicious areas, then perform deep analysis on those areas, and finally verify findings by reconstructing how vulnerabilities can be triggered. This staged approach helps manage context window limits while improving detection accuracy through focused attention at each step.

Your pipeline should have **at least 3 stages**:

1. **Stage 1 - Triage:** Identify potentially vulnerable functions/files
2. **Stage 2 - Deep Analysis:** Analyze flagged code in detail
3. **Stage 3 - Verification:** Provide execution path (sequence of functions/lines from entry point to vulnerability) to demonstrate how the vulnerability can be triggered. *Note: an exact exploit payload is not required—focus on identifying the vulnerable execution flow.*

Important: The output from each stage can be passed to the next stage either:

- **Directly:** Using the raw LLM output as-is
- **With local processing:** Your script may parse, filter, or reformat the LLM output before passing it to the next stage

All three stages should complete sequentially to produce the final vulnerability report.

Deliverable

Your `pipeline/` directory should contain:

- `stage1_triage.py`: Stage 1 implementation
- `stage2_analysis.py`: Stage 2 implementation
- `stage3_verify.py`: Stage 3 implementation
- `orchestrator.py`: Coordinates all stages

3 Patch Generation & Validation (15 points)

For vulnerabilities detected in the provided samples, use an LLM to automatically generate patches that eliminate the vulnerabilities.

Requirements

For each vulnerability detected, you must:

- Use an LLM to generate a patch that **eliminates the vulnerability** without breaking normal program functionality
- Ensure the patched code is **compilable** and runs correctly
- **Model Selection for Patch Generation:** Use any available model *except* VulnLLM-R-7B for generating patches (VulnLLM-R is specialized for detection, not generation)
- Do NOT write patches manually - all patches must be LLM-generated
- **Validate the patch** using two methods:
  1. Ask the *same LLM* that generated the patch to re-analyze the patched code and confirm it is no longer vulnerable
  2. Ask a *different LLM* to analyze the patched code to verify the fix (you can use VulnLLM-R-7B here for verification)
- Document the rationale for why the patch eliminates the vulnerability

Deliverable

- `patch_generator.py`: Your patch generation logic
- `patches/`: Directory containing patched versions of vulnerable samples
- Each patch must be a valid, compilable Cpp file

4 LLM Model Comparison (10 points)

The goal of this section is to compare the vulnerability detection capabilities of Large Language Models (LLMs). To achieve this, you will use six vulnerable code samples provided to you as positive (vulnerable) examples, and their corresponding patched versions as negative (non-vulnerable) examples. These samples will be used to evaluate the outputs of different LLMs.

To increase the statistical significance of the evaluation, you are also required to add five additional vulnerable code samples along with their corresponding patches, resulting in a larger evaluation dataset.

Evaluation Metrics The performance of each model will be assessed using the following standard classification metrics:

- **True Positives (TP):** The model correctly identifies vulnerable code as VULNERABLE
- **True Negatives (TN):** The model correctly identifies patched code as SAFE
- **False Positives (FP):** The model incorrectly classifies patched code as VULNERABLE
- **False Negatives (FN):** The model incorrectly classifies vulnerable code as SAFE

Models to Be Evaluated The following LLMs must be used in the evaluation:

- **VulnLLM-R-7B** — specialized for vulnerability detection
- **DeepSeek-V3.1**
- **GPT-OSS-120B**
- **Qwen3-32B**

Model Access Instructions for accessing the models are provided at the end of the assignment.

Deliverable

- Your `llm_wrapper.py` should support calling different models with a unified interface
- `test_samples/`: Directory containing all vulnerable and patched code samples you used for testing (both the 6 provided samples with patches, and the 5 additional pairs you found/created)

5 Real-World Analysis (20 points + 40 Bonus)

Apply your best strategy to real open-source projects.

Required (20 points):

- Analyze **at least 2** real-world Cpp projects from GitHub with the following criteria:
  - Primary language: Cpp
  - Project size: Between 5K and 10K lines of code
  - GitHub stars: $> 8{,}000$ (indicating significant real-world usage and reliability)
  - Public repository with clear purpose (utilities, libraries, parsers, network tools, etc.)
- **Additionally**, select **one recent CVE from 2025** (in Cpp) to ensure the LLM has not seen it during training. The CVE should be related to topics covered in this course. Read the CVE description to understand what the vulnerability is and its execution path (this information is provided in the CVE description). Then apply your detection strategies to see if your LLM-based approach can detect this known vulnerability.
- Document all findings (true vulnerabilities and false positives)
- Classify by CWE type and severity

Grading Criteria: Your grade will be based on the quality and thoroughness of your analysis, not solely on finding vulnerabilities:

- **Methodology (8 points):** Quality of context preparation strategies, choice of appropriate prompts, systematic approach to analysis
- **CVE Analysis (7 points):** Successfully detecting the known 2025 CVE vulnerability, comparing your LLM's findings with the official CVE description, explaining why detection succeeded or failed
- **Analysis Quality (5 points):** Clear documentation of all findings (both vulnerabilities and false positives), proper CWE classification, severity assessment, and critical analysis of why false positives occurred

Important: If the LLMs fail to detect any vulnerabilities in the GitHub projects (but successfully detect the known CVE), you can still receive full points by:

- Demonstrating thorough application of multiple strategies
- Providing evidence of systematic analysis (what was checked, which code sections were examined)
- Critically analyzing why detection failed: limitations of context preparation, LLM capabilities, or the projects being genuinely secure

- Showing successful detection of the known CVE demonstrates your methodology works

Note: We recognize that some projects may have no vulnerabilities. In such cases, provide:

- Evidence of thorough analysis (what strategies were applied, what was checked)
- Discussion of whether the lack of findings is due to: (1) the project being genuinely secure, (2) limitations of your context preparation, or (3) LLM limitations

Bonus (40 points):

- Find a **previously unreported vulnerability** (0-day) in a real-world project
- Submit your findings to the TA/instructor first. The instructor will verify the vulnerability and coordinate responsible disclosure if valid.
- Your submission must include:
  - Detailed vulnerability description (CWE type, affected code location)
  - **Proof of Concept (PoC):** Working exploit demonstrating the vulnerability
  - Impact analysis and severity assessment
  - Proposed patch (if possible)

Important: Bonus points are awarded only for *verified, genuine vulnerabilities* with a working PoC. False positives will not receive credit.

Suggested Projects: Small utilities, network daemons, parser libraries, embedded tools.

Deliverable

Your real-world analysis should produce:

- Source code of analyzed projects in your submission (or links to specific commits)
- Analysis results documenting all findings, including:
  - Detected vulnerabilities (CWE type, file location, line numbers, severity, explanation)
  - False positives identified during manual review
  - CVE 2025 analysis results comparing your findings with official CVE description
- Your results can be in any clear, structured format (JSON, CSV, or formatted text)

Submission Requirements

Code Structure

Your submission directory should follow this structure:

```
submission/
|
|-- main.py                          Main entry point for all assignment parts
      Usage: python main.py --mode <MODE> --target <DIR>
      <MODE> can be: strategies, pipeline, patch, compare, realworld
|
|-- strategies/                      Part 1: Context preparation strategies
      |-- baseline.py                Concatenate all code files
      |-- pattern_based.py           Pattern-based extraction
      |-- callgraph.py               Call-graph analysis
```

```
    |-- your_custom.py            Your additional strategy
|
|-- pipeline/                     Part 2: Multi-stage analysis pipeline
    |-- stage1_triage.py          Identify suspicious code
    |-- stage2_analysis.py        Deep vulnerability analysis
    |-- stage3_verify.py          Execution path verification
    |-- orchestrator.py           Coordinates all stages
|
|-- patch_generator.py            Part 3: Patch generation
|
|-- llm_wrapper.py                API wrapper for calling LLMs
|
|-- patches/                      Generated patched versions
    |-- sample1_patched.c
    |-- sample2_patched.c
|
|-- results/                      Analysis results (any format: txt/json/csv)
    |-- strategies_results.txt
    |-- pipeline_results.txt
    |-- comparison_results.txt
    |-- realworld_results.txt
|
|-- report.pdf                    Comprehensive written report
|-- README.md                     Setup and usage instructions
```

README.md Requirements

Your README.md must include:

- **Setup Instructions:** Dependencies, API keys, environment setup
- **Deployment Steps:** How to install and configure your code
- **Testing Instructions:** How to run each part of the assignment
- **Example Commands:** Concrete examples for running each mode

See the provided `README.md` template in the handout for detailed examples.

Report Requirements (report.pdf)

Your report must explain:

- **Part 1 - Strategies:**
  - Description of each strategy you implemented
  - Any new heuristics or techniques you designed
  - Comparison results: which strategy found what vulnerabilities
  - Analysis of false positives on patched code
- **Part 2 - Pipeline:**
  - Design rationale for your multi-stage approach
  - How you process and pass information between stages
  - Comparison with single-stage detection
- **Part 3 - Patch Generation:**

  - Methodology for generating patches
  - Validation results (same model + different model verification)
  - Examples of generated patches with explanations

- **Part 4 - Model Comparison:**
  - Comparison table with metrics (TP/TN/FP/FN, Precision, Recall, F1)
  - Analysis of which models perform best and why
  - Discussion of false positive rates

- **Part 5 - Real-World Findings:**
  - Projects analyzed (names, versions, links)
  - Vulnerabilities found (CWE, location, severity, explanation)
  - False positive analysis
  - CVE 2025 analysis results
  - Discussion of limitations (context preparation vs LLM capabilities)

Report Quality: A technically deep report is required for full credit. Shallow reports will receive significant penalties.

Provided Resources

The handout repository provides base templates with a **unified interface**:

- `main_template.py`: Main entry point template with unified CLI

```
# Usage: python main.py --mode <MODE> --target <DIR>
# MODE: strategies | pipeline | patch | compare | realworld
```

- `strategies/base_strategy.py`: Base class for context preparation strategies
  - Implement: `prepare_context(target_dir)` → returns prepared code
  - Implement: `detect_vulnerabilities(context)` → returns findings
- `pipeline/base_pipeline.py`: Base class for multi-stage pipeline
  - Implement: `stage1_triage(code)` → potentially vulnerable locations
  - Implement: `stage2_analysis(locations)` → detailed vulnerability info
  - Implement: `stage3_verification(vulnerabilities)` → execution paths
- `llm_client.py`: Wrapper for calling the course API server with different models
- `provided_samples/`: 6 vulnerable Cpp code samples (CWE-120, 78, 416, 119, 22, 190)

All templates follow the same pattern:

1. Accept a target directory path via command-line argument
2. Use mode flags (`--mode`) to determine which part of the assignment to execute
3. Output results in a clear, organized format of your choice (text, JSON, CSV, etc.)
   - The format should be structured and easy to read for grading
   - Include all required information (vulnerabilities, CWE types, locations, severity, etc.)
   - Consistency across all parts is recommended but not required

LLM Access

Course API Server (Required for Final Evaluation):

- **API URL:** [REDACTED]
- **Authentication:** Each student will receive a personal API token via email. If you do not receive your token, send your student ID and full name from your official Sharif email to: `m.hadadian76@sharif.edu`
- **VPN Requirement:** The API server is only accessible from Sharif University IP addresses. You must connect via **SharifVPN** to access the server.
- **Available Models:** VulnLLM-R-7B, DeepSeek-V3.1, GPT-OSS-120B, Qwen3-32B
- **Usage Limits:** Each token has limited usage quota. Use your tokens only for this homework assignment.
- **Logging Notice:** All API calls are logged for monitoring and fair usage enforcement.

Development and Testing: During development and testing, you may use free public APIs (e.g., OpenRouter with DeepSeek) to avoid exhausting your course API quota. Once your code is working, run your final experiments and comparisons on the course API server for grading.

Important: Save your API token quota for final runs. Do not waste tokens on debugging or repeated identical queries.

Important Notes

Automatic Strategies: All strategies must be fully automatic (no manual code selection).

Cross-File Vulnerabilities: Your strategies should handle vulnerabilities spanning multiple files.

Output Format: All scripts must show: input code, LLM response, detected vulnerabilities (CWE, location, severity), execution time. All prompts used must be documented in your report.

Citations: If using techniques from papers/articles, cite them properly.

Vulnerable-Patched Testing: All evaluations must test models on *both* vulnerable and patched versions. This measures both detection capability (TP rate) and false positive rate (FP on safe code). A good vulnerability detector should flag vulnerable code while accepting patched versions.

Code Robustness: Your implementation must be robust enough to handle unseen code:

- We may test your code on **6 additional vulnerable samples** not provided in the handout
- In Part 5, your code will be run on **real-world projects selected by other students**
- Your strategies should generalize beyond the provided samples

LLM Usage Policy

You may use LLMs (ChatGPT, Claude, DeepSeek, etc.) for:

- Writing utility functions
- Debugging

- Improving report clarity

Requirements:

- Disclose all LLM usage in your report (which services, for what tasks)
- Core strategies must be your original design
- You are fully responsible for correctness

Academic Integrity: Do not share code with other students. Cite any code from external sources.