PES UNIVERSITY

100 Feet Ring Road, Banashankari Stage

III, Dwaraka Nagar, Banashankari,

Bengaluru, Karnataka 560085

Dept. of Computer Science & Engineering

UE23CS342BA9: Generative AI and its Applications

Lab Exercise 3: Prompt Engineering for Cyber Threat Intelligence - Orange Problem

Course Anchor: Dr. Uma D

Teaching Assistants: Suchir M Velpanur, Chirag A

Exercise : Exploring Prompting Paradigms for Enterprise Threat Analysis

Objective

The goal of this exercise is to transition from interacting with Large Language Models (LLMs) as simple conversational interfaces to engineering them as structured, reliable reasoning engines. You will move beyond naive zero-shot prompting to implement advanced structured reasoning frameworks (Chain, Tree, and Graph-of-Thought) and iterative extraction loops (ReAct) using a single LLM.

By the end of the exercise, you should be able to:

- Understand and demonstrate the limitations of base LLMs in semantic version math, interval boundary reasoning, and the interpretation of nuanced corporate mitigation statements.
- Implement foundational prompting techniques, such as zero-shot, one-shot and few-shot techniques to visualize how baseline models handle raw vulnerability assessments with varying degrees of provided context..
- Apply sequential and branching logic frameworks (Chain-of-Thought and Tree-of-Thought) to bypass linear reasoning traps and calculate robust vulnerability risk scores from unstructured JSON data.
- Engineer a Graph-of-Thought (GoT) to synthesize multiple distinct data nodes (Threats, Vendor PR, Internal Assets) into actionable mitigation tables and analyse severity of risks.
- Develop an iterative Reason and Act (ReAct) prompting loop in Python to demonstrate how an LLM can be constrained via generation stop-sequences to dynamically query specific data fields from a JSON object step-by-step (such as threat details or vendor mitigations) before making a final risk classification, without relying on heavy abstraction frameworks like LangChain.
- Experiment with exploratory prompt engineering techniques such as self-consistency prompting, prompt decomposition, role-based and reflection prompting to measure quantitative improvements in output reliability.

Domain Primer: Understanding Vulnerability Intelligence

Before leveraging Generative AI for cybersecurity, you must master the core mechanics of threat intelligence and vulnerability management. To engineer high quality and effective prompts for vulnerability assessment, you need a firm grasp of how security flaws are tracked globally and how their impact is quantified within an enterprise environment.

1. What is a vulnerability?

In cybersecurity, a vulnerability is a weakness or flaw in a software program, hardware device, or system configuration that an attacker can exploit to compromise the system's confidentiality, integrity, or availability. Common examples present in your lab dataset (but not limited to) include:

- **Denial of Service (DoS):** An attacker crashes the system (e.g., sending millions of requests to a server or excessive memory allocation), rendering it unavailable to legitimate users.

- **SQL Injection (SQLi):** An attacker manipulates input fields to execute malicious database queries, allowing them to bypass logins or steal data.

2. The Global standard in vulnerability: CVE and NVD

- **CVE (Common Vulnerabilities and Exposures):** A globally recognized dictionary of publicly disclosed vulnerabilities. Each vulnerability is assigned a unique ID (e.g., CVE-2018-6522).
- **NVD (National Vulnerability Database):** The U.S. government repository that synchronizes with the CVE list. The NVD enriches the raw CVE identifiers with detailed analysis, technical configurations, and risk scores. The dataset you are using (student_cve_dataset.json) is a processed data feed directly from the NVD.

3. Anatomy of an NVD JSON Payload

To engineer your prompts effectively, you must understand the three core components of the NVD JSON data that your LLM will be parsing:

- **descriptions:** Unstructured, human-readable text (often in multiple languages) explaining the root cause of the flaw and the potential impact of an exploit.
- **configurations:** This dictates *who* is affected. It uses a standard called CPE (Common Platform Enumeration). It defines version boundaries, such as "versionStartIncluding": "4.0" and "versionEndExcluding": "4.0.0.39".
- **vendorComments:** Unstructured statements provided by the software manufacturer identifying these vulnerabilities. These often contain PR statements, workarounds, or links to download patches.

4. Vulnerability Risk Scoring using CVSS

Cybersecurity professionals use the **CVSS (Common Vulnerability Scoring System)** to assign a numerical score from 0.0 to 10.0 to a threat. In this lab, you will prompt your LLM to classify threats into four categories: **SAFE** (0), **LOW** (1-3), **MEDIUM** (4-7), and **HIGHLY VULNERABLE** (8-10). CVSS is calculated based on the inherent severity of the vulnerability. For example, a vulnerability that allows a remote attacker to easily delete databases will always have a score of 10.0. It can also be influenced by the actual risk to a *specific* organization based on their internal setup.

Example: *Imagine a severe vulnerability has a base score of 9.0. The vendor releases a comment saying, "A patch is available, please click this link to download the update." A naive LLM will read the vendor comment, assume the problem is solved, and output a LOW risk score. However, if your internal computer is a highly sensitive, air-gapped workstation with no direct Internet access, you cannot download that patch. The asset remains unpatched and exposed. A sophisticated AI reasoning engine must recognize this environmental constraint and maintain a HIGH risk score.*

5. How Vulnerabilities are Detected and Triaged?

Detecting vulnerabilities in an enterprise relies on automated scanners that scan internal servers and compare their installed software versions against the configuration boundaries in the NVD database.

However, detection is only the first step. Triage is the process of deciding what to fix first. Traditionally, human Security Analysts must manually read the NVD descriptions and vendorComments, cross-reference them with the company's internal network architecture (e.g., "Is this server connected to the internet?"), and calculate the final risk score.

Your Goal: In this lab, you are automating this complex, human-level triage process. You will engineer an LLM through prompt engineering to read the raw threat data, apply internal environmental constraints, debate the effectiveness of vendor patches, and output a highly accurate triage classification.

6. Semantic Versioning

In the configurations field of a CVE, software bounds are written using Semantic Versioning (SemVer). This usually follows a strict sequence: Major.Minor.Patch (e.g., Version 4.0.0).

- **Major Version:** A massive change to the software, often breaking compatibility (e.g., moving from Windows 10 to Windows 11).
- **Minor Version:** Added features, but backwards compatible (e.g., v4.1 to v4.2).
- **Patch Version:** Small bug fixes and security patches (e.g., v4.0.0 updating to v4.0.1).
- **Release:** A release is simply the actual public distribution of any of the versions above. When a vendor fixes a vulnerability, they release a patch and publish "Release Notes" (which you will often see in the vendorComments of your JSON data).

Note: *To a human, it is obvious that version 4.0.0.35 is older (and therefore vulnerable) compared to the patched version 4.0.0.39.*

However, base LLMs are terrible at semantic version math. They often treat numbers with decimals as floating-point math. When an LLM sees 4.0.0.39, its mathematical reasoning engine gets confused by the multiple decimal points. The process of breaking text into tokens can sometimes strip a version string of its meaning. A value like 4.0.0.35 may be fractured into disjointed segments such as ["4", ".0.0", ".35"], effectively destroying its numerical hierarchy and context for the AI.

As a result, a naive LLM will frequently hallucinate and state that 4.0.0 is completely safe from a < 4.0.0.39 boundary just because it couldn't do the tokenization properly. In this lab, you will design robust and effective prompts to force the LLM to normalize these versions and perform step-by-step logic to overcome this fundamental AI weakness.

The Scenario

Modern Security Operations Centers (SOCs) rely on automated systems to ingest and interpret real-time vulnerability intelligence feeds from the National Vulnerability Database (NVD). In fact, this type of automated vulnerability analysis is the exact backbone powering leading developer cybersecurity tools like **GitHub's Dependabot** and **GitGuardian**.

You are a Lead Security AI Engineer at a top-tier Security Operations Center (SOC). Your team monitors thousands of enterprise servers and is responsible for ingesting real-time threat intelligence feeds from NVD.

The triage pipeline is currently overwhelmed. To automate the assessment of incoming threats, the engineering team attempted to plug a standard Large Language Model (LLM) into the pipeline. However, this naive approach is failing dangerously. The base LLMs frequently misunderstand semantic version intervals (e.g., falsely assessing an asset running version 4.0.0.35 as "Safe" when the vulnerable boundary is < 4.0.0.39). Worse, they blindly trust positive corporate vendor PR statements (e.g., *"A patch is readily available for download!"*) without considering your company's strict internal environmental constraints (e.g, such as the fact that your highly sensitive workstations are air-gapped and have no direct internet access)

To solve these critical security blind spots, you will investigate whether Prompt Engineering alone, with no fine-tuning or model retraining, can transform a naive LLM into a hyper-reliable, context-aware cyber threat reasoning assistant.

The Task

You have been tasked with engineering a robust prompting pipeline that can dynamically evaluate raw NVD JSON data, bypass LLM logic traps (specific architectural flaws where the model's "common sense" language processing contradicts the rigid mathematical rules of cybersecurity), and assign a mathematically sound vulnerability risk score (0-10) and accordingly classify them as SAFE, LOW, MEDIUM, HIGHLY VULNERABLE.

Instead of relying on basic API calls, you will engineer complex cognitive architectures. You will build this system in three modules:

- **Module 1: Baseline Logic** You will implement Zero-Shot, One-Shot, and Few-Shot prompts. Through these implementations, you will observe how base LLMs easily fall into logic traps.
- **Module 2: Sequential, Branching, and Connected Reasoning Frameworks:** To fix the logical failures observed in Module 1, you will deploy advanced prompting paradigms. You will first implement Chain-of-Thought (CoT) to enforce a strict, step-by-step heuristic evaluation. Next, you will engineer Tree-of-Thought (ToT) to computationally force the LLM to branch and debate its own logic. Finally, you will build a Graph-of-Thought (GoT) architecture to mathematically map and synthesize distinct data nodes (such as CVE threats, Vendor mitigation, Internal asset status) into an actionable defense matrix.
- **Module 3: Reasoning-based Workflows & Exploratory Prompting:** You will develop an iterative Reason and Act (ReAct) prompting loop from scratch in Python, utilizing text-generation stop-sequences to force the LLM to dynamically query specific data fields step-by-step. Finally, you will experiment with exploratory prompting techniques such as self-consistency prompting, prompt decomposition, role-based and reflection prompting to measure quantitative improvements in the LLM's output reliability.

Data Sources for Corpus and Dataset Schema

You will utilize a curated dataset of the National Vulnerability Database (NVD) JSON 2.0 feed namely nvdcve-2.0-2018.json. It contains a list of real-world vulnerabilities with critical fields that would assist in analysing the extent of vulnerability in the concerned software. You must use these to deduce the severity of the CVE vulnerability and categorize it into SAFE, LOW, MEDIUM, and HIGHLY VULNERABLE classes.

Dataset Schema

To engineer your prompts effectively, you must understand the schema of the JSON objects you will be feeding to the LLM. Each entry in the dataset contains a root cve object with the following accessible fields:

- **id**: The unique Common Vulnerabilities and Exposures alphanumeric identifier (e.g., "CVE-2018-4868").
- **sourceIdentifier**: The email address or organization name of the entity that initially reported or assigned the CVE (e.g., "cve@mitre.org").
- **published / lastModified**: Timestamp strings indicating exactly when the vulnerability was first disclosed and last updated in the database.
- **cveTags**: An array of specific tags associated with the CVE record (often empty in older records).
- **descriptions**: An array containing the human-readable text of the vulnerability, often provided in multiple languages (indicated by the lang key, such as "en" for English). The value key details the exploit mechanism, the affected component, and the potential impact (e.g., remote denial of service via excessive memory allocation).
- **configurations**: An array of nodes defining the exact boundaries of the affected software.
    - operator: Defines the logical relationship between configurations (e.g., "OR").
    - cpeMatch: An array containing standard criteria strings (Common Platform Enumeration, e.g., "cpe:2.3:a:exiv2:exiv2:0.26:*:*:*:*:*:*:*"). Depending on the CVE, this array may also contain strict interval boundaries such as versionStartIncluding and versionEndExcluding.
- **references**: An array of external links pointing to third-party advisories, exploit databases, or issue trackers related to the threat. It includes the url, the source of the link, and descriptive tags (e.g., "Exploit", "Third Party Advisory").
- **vendorComments**: *(Note: Present only in a subset of the dataset)* An array of statements issued directly by the software manufacturer. This field often contains proposed workarounds, PR statements, or download links to security patches.

Prerequisites

Technical Requirements

1. Python 3.10 or higher.
2. We strongly recommend running your labs on Google Colab or in Kaggle notebooks to ensure consistency and avoid local dependency issues.
3. A valid API Key for the **Groq API**.
    - *Note on Rate Limits:* Groq provides exceptional inference speed on their free tier, but strict rate limits apply. If you exhaust your API key rate limit (Requests Per Minute/Day or Tokens/Day) during your experiments, you may need to generate a new API key using an alternate account to continue the lab.
4. Unlike previous labs where a specific model was hardcoded, you are required to explore and experiment with different open-weights models hosted on Groq (e.g., the LLaMA 3 family, Mixtral, Gemma). You can find the complete list of supported models and their required string IDs on the Groq documentation page: https://console.groq.com/docs/rate-limits
5. Required Packages**:** groq, json, re, collections.

Instructions Steps to Complete the Exercise

You will execute this lab in 6 distinct phases, moving from foundational heuristic extraction to engineering complex, autonomous prompting & reasoning loops.

Testing Requirement: To properly observe the strengths and weaknesses of your prompts, you must test your pipelines in Phases 1 through 6 against **THREE specific CVE samples** from nvdcve-2.0-2018.json:

1. CVE-2018-3810
2. CVE-2018-3814
3. CVE-2018-6523

Phase 0: Environment Setup & Data Preparation

Before writing your prompts, you must initialize the lab environment, extract the target data from the raw NVD feed, and set up a dynamic testing pipeline.

1. Run the setup cells to install the groq library and initialize your LLM client.
2. The raw nvdcve-2.0-2018.json file contains tens of thousands of entries. You must implement an extraction script (extract_and_mask_target_cves) to isolate only the three target CVEs required for this lab.
    - Your script must actively extract crucial IDs such as descriptions, configurations and vendorComments and remove fields which may not contribute to our analysis.
    - Save this filtered, processed output as cve_dataset.json.
3. Once your dataset is generated, you need to write a utility to search cve_dataset.json for the matching 3 CVE IDs and load only that specific JSON object into a masked_cve_json variable. By doing this, you will be able to rapidly test your prompt architectures against all three vulnerabilities without rewriting your code.

Phase 1: Baseline Prompting (Zero, One, and Few-Shot)

Your first task is to establish a baseline of how standard LLMs perform vulnerability.

- **Zero-Shot:** Design a system prompt that explicitly defines an *Internal Asset Profile* (e.g., "Highly sensitive local workstation, no direct internet access"). Pass the raw CVE JSON directly and ask the model for a score (0-10) and classification. Note how it hallucinates scores or overlooks your constraints.
- **One-Shot:** Provide a structured example in the system prompt demonstrating how to calculate a severe risk score based on an example (refer to the boilerplate notebook for the example text).
- **Few-Shot:** Provide the **3 specific examples as given in the boilerplate notebook** markdown to guide the model's reasoning pattern before asking it to evaluate your target CVEs.

Phase 2: Chain-of-Thought (CoT) Prompting

You will force the LLM to abandon its guessing and execute a strict, step-by-step heuristic scoring engine.

- You must critically analyze the available JSON data fields (descriptions, configurations, vendorComments). How would a human security analyst sequentially process this information to triage a threat? (***Hint: Think about the order of operations. You need to understand what the threat is, identify exactly who it affects, and check if a fix exists before you can even begin to calculate a score***).
- Design a CoT system prompt that explicitly commands the model to follow your custom logical sequence (e.g., a 4 to 5-step process). The LLM must explicitly write out its reasoning for each of your defined steps before it is allowed to output the final score and classification.

Phase 3: Tree-of-Thought (ToT) Prompting

- You must instruct the LLM to computationally branch its reasoning into at least three distinct paths ***regarding the vendor's response (the vendorComments section)***. Think critically about how a human might interpret this conflict: ***What would an optimistic analyst conclude? What would a cynical analyst conclude? What is a neutral middle-ground?*** You must define these branching personas or logic paths in your prompt. Instead of a single logical path, you must engineer a prompt that explores multiple interpretations.
- A tree of thought is useless without pruning. Your prompt must force the model to rigorously evaluate all three generated branches against the strict internal asset profile. It must explicitly select the most logically sound, surviving branch before outputting the final risk score and classification.

Phase 4: Graph-of-Thought (GoT) Multi-Source Synthesis

You will move beyond sequential and branching logic by mapping independent pieces of unstructured data into a relational graph architecture.

- What are the core, independent entities involved in a vulnerability assessment? Design a prompt step that instructs the LLM to extract these pieces of context as distinct nodes (***e.g., you may need a node for the core threat, a node for the vendor's stance, a node for your internal asset etc.*****).**
- How do these nodes interact? Design the logical edges that connect them to act as pass/fail gates. For example: What logical check must happen to connect the Threat Node and the Asset Node? ***What asset profile check must connect the Vendor Node to the Asset Node?*** Instruct the LLM to establish these edges.
- Instruct the LLM to traverse the graph, using the edges to modify or nullify the risks presented by the nodes. The model must synthesize this graph into a final decision and output it in a strict markdown table (| Asset | CVE | Vulnerable | Vendor Status | Action |).

Phase 5: ReAct (Reason + Act) Iterative Threat Investigation

You will build an iterative Reason and Act (ReAct) prompting loop from scratch. This architecture prevents LLM data hallucination by grounding its reasoning in verifiable, step-by-step data retrieval.

- You must engineer the LLM to operate in a strict Thought -> Action -> Observation loop.
    - **Thought:** The LLM reasons about what information it is currently missing.
    - **Action:** The LLM requests to use a specific tool to fetch that information.
    - **Observation:** The Python environment returns the factual data.

- Design a prompt that forces the LLM to output exactly one Thought: and one Action: per turn. You must define the exact actions it is allowed to take (e.g., get_cve_description, get_cve_bounds, get_internal_asset).
- Build a while loop that performs these actions which will dynamically extract the corresponding data from your targeted CVE JSON object, and feed it back to the LLM.
- You MUST utilize the stop=["Observation:"] parameter in your API call. This forces the LLM to pause generation, allowing your Python code to inject the real data back into the conversation history until the LLM explicitly calls the final_decision action.
- Implement this for 6 Thought -> Action -> Observation loops.

Phase 6: Exploratory Prompt Engineering Techniques

You must implement **at least THREE** of the following techniques in your notebook to analyze quantitatively in your final report:

1. **Self-Consistency (Majority Voting):** Generate 5 independent reasoning chains using a higher temperature parameter. Use a Python counter to output the final majority vote.
2. **Prompt Decomposition:** Split the task into 4 sequential micro-prompts (Extract Interval → Normalize Version → Compare Versions → Classify Result), piping the output of one prompt directly into the input of the next.
3. **Role-Based Prompting:** Compare the behavioral differences and score variances when the LLM is explicitly instructed to act as a "Security Analyst", a "Software Engineer", and a "Risk Auditor".
4. **Reflection Prompting:** Ask the LLM for an initial assessment, and then supply a follow-up Reflection Prompt demanding it to: "Review your initial reasoning and explicitly correct logical errors regarding the fact that the asset has NO internet access."

IMPORTANT: Automated Evaluation Policy

READ CAREFULLY: This lab is graded using an automated evaluation script. To ensure you receive credit:

- DO NOT MODIFY any function signatures, names, or return types provided in the boilerplate code.
- DO NOT CHANGE the file structure or the names of the input/output files.
- DO NOT CHANGE any defined macros or model names that may be provided to you.
- STRICTLY FOLLOW the TODO blocks. You may add helper functions inside the block, but the main function definitions must remain exactly as given.

Failure to adhere to these constraints will cause the autograder to fail, resulting in a reduction of score assigned.

Submission Requirements

Your submission should include the following:

1. Code Implementation

- A Python file (<SRN>_GenAI_Lab3.py) containing all objectives implemented from scratch. (Download the Python file from Google Colab as : File > Download > Download .py)

- A Notebook file (<SRN>_GenAI_Lab3.ipynb) containing all objectives implemented with **clear, visible cell outputs.**

2. Report (<SRN>_GenAI_Lab3_Report.pdf)

A concise PDF report summarizing your observations and analysis. You must answer the following questions using trace logs, scores, and specific evidence extracted from your Jupyter Notebook outputs:

1. Construct a comparative table documenting the LLM-generated vulnerability analysis report for the three provided CVE IDs. For each experimental approach used in the lab, ensure the table explicitly captures the numerical risk score (0–10) and the specific vulnerability category label (e.g., SAFE, LOW, MEDIUM, or HIGHLY VULNERABLE) as output by the model.
2. Provide a trace snippet showing how your zero-shot prompt struggled with semantic version math (e.g., evaluating versions against < 4.0.0.39). How did the strict step-by-step heuristic extraction in Chain-of-Thought (CoT) temporarily help bypass the LLM's floating-point/tokenization trap?
3. Inspect your CoT output for a CVE ID: CVE-2018-6523. Did the model lower the risk score upon seeing the vendorComments? Explain why linear reasoning fails to recursively apply the Internal Asset Profile (such as "No Direct Internet Access" constraint) to this newly discovered mitigation.
4. For the Tree-of-Thought implementation, quote the exact reasoning generated by the LLM for the "Cynical" branch with the help of an example. How did computationally forcing this negative perspective prevent the logic trap observed in CoT?
5. Look at your ToT output for CVE ID: CVE-2018-3814. How did the ToT prompt handle the strict requirement to generate 3 vendor-related branches when the underlying data was null? Did the model hallucinate a patch, or did it adapt gracefully? Provide evidence from the logs.
6. Provide the exact markdown table row generated for GoT implementation for CVE ID: CVE-2018-3814 and explain how the logic of this edge successfully overrode the Vendor Status node.
7. Explain the precise implementation mechanism of stop=["Observation:"] in your ReAct Python loop. What would physically happen to the LLM's text generation trajectory if you omitted this parameter from the API call?
8. Provide a complete, sequential 3-step extraction trace (Thought -> Action -> Observation) from your Phase 5 logs for CVE ID: CVE-2018-3810. How does this iterative, tool-based data retrieval fundamentally prevent the context-window overload and hallucination seen in Phase 1?

Use all 3 CVE IDs for these set of questions:

9. List the 5 individual scores generated by your Self-Consistency chains. Did they diverge? Explain the relationship between the higher temperature parameter and the variance in how the LLM weighed the air-gapped constraint. Why is majority voting mathematically safer for threat triage?
10. In Phase 6.2, did Micro-Prompt 3 ever fail the mathematical comparison (e.g., incorrectly claiming that a normalized version like 4.0.0.0 is completely safe from the < 4.0.0.39

boundary)? Explain how a single failure in this micro-prompt cascades through the rest of the pipeline.

11. Create a brief comparison of final risk scores and key justification phrases used by the "Software Engineer" versus the "Risk Auditor" roles. How did the injected persona fundamentally alter the scores the model assigned to the vulnerability's *impact* versus its *mitigation*?
12. Provide the exact text snippet from Phase 6.4 where the model reviewed its initial assessment. How did the explicit reflection prompt regarding "NO internet access" force the model to course-correct its final classification?
13. Compare your final ReAct or GoT scores across the different CVEs tested. Explain how your advanced prompting architectures successfully transitioned the LLM from acting as a generic calculator of *base CVSS scores* (like a typical NVD lookup) to a highly specific engine calculating *robust risk scores* tailored to your simulated SOC.

Grading Criteria

Criterion	Weightage
Correct implementation	40%
Documentation and code readability	20%
Quality of analysis and discussion	30%
Presentation of results	10%