UNIVERSITÀ DEGLI STUDI DI VERONA
CORSO DI LAUREA TRIENNALE IN INFORMATICA

Uno studio sull'utilizzo degli LLM per identificare vulnerabilità nel codice

CANDIDATO:
BRUNO VINCENZI
VR471499

RELATRICE:
PROF.SSA FEDERICA MARIA FRANCESCA PACI

CO-RELATRICE:
PROF.SSA MILA DALLA PREDA

INDICE

COSA SONO I LARGE LANGUAGE MODELS (LLM)
BREVE STORIA E UTILIZZI

Un Large Language Model è una tecnologia AI avanzata incentrata sulla comprensione e sull'analisi del testo. È in grado di cogliere le complessità del linguaggio naturale e riesce a generare risposte significative fornendo informazioni elaborando grandi quantità di testo.[01]

La storia dei Large Language Model (LLM) affonda le sue radici nella ricerca sul linguaggio naturale e sull'intelligenza artificiale. L'idea di macchine capaci di comprendere il linguaggio naturale risale agli anni '30 del XX secolo, con le teorie di Alan Turing e Kurt Gödel. Ovviamente la tecnologia di quell'epoca non consentiva la realizzazione di tali macchine, per questo motivo rimase solamente una brillante teoria.
Il primo programma conversazionale vicino all'idea di LLM fu realizzato nel 1967 all'interno del MIT (Massachusetts Institute of Technology) e gli fu attribuito il nome "ELIZA". Esso utilizzava una serie di script conversazionali per rispondere alle domande degli utenti.
Nonostante la somiglianza, ELIZA non fu un chatbot AI come quelli moderni, infatti si basava su semplici regole e pattern matching senza una vera e propria comprensione semantica.
Un primo grosso passo verso la nascita degli LLM invece fu la creazione di "Word2vec", un algoritmo di elaborazione del linguaggio naturale (NLP) in grado di convertire una parola in un vettore in uno spazio multidimensionale facilitando molte applicazioni di elaborazione del linguaggio naturale.

Il primo vero e proprio LLM fu "BERT", arrivato nel 2019, il suo scopo fu quello di migliorare l'algoritmo del motore di ricerca di Google per aiutarlo a "comprendere" meglio le ricerche degli utenti e fornire risultati migliori.
Nel 2022, OpenAI presenta al mondo ChatGPT un applicazione basata su GPT-3.5 ovvero un Large Language Model rilasciato nel 2022 e successivamente sostituito (anche se non completamente) da GPT-4 (2023). Il miglioramento non è avvenuto solo in termini di prestazioni, ma anche di sicurezza e affidabilità per un utilizzo responsabile.[02]
I compiti più frequenti eseguiti dagli LLM sono la risposta a domande aperte, la chat, la sintesi di contenuti, la traduzione e la generazione di contenuti e codici.
Allo scopo di eseguire questi compiti però gli LLM necessitano di un addestramento che viene eseguito a partire da enormi dataset utilizzando algoritmi avanzati di machine learning per apprendere i pattern e le strutture del linguaggio umano.[03]

01 | Hewlett Packard Enterprise, *Cos'è un Large Language Model?*, URL: hpe.com/it
02 | Fastweb, *Large Language Models*, cosa sono e come funzionano, URL: fastweb.it
03 | DataBricks, Modelli linguistici di grandi dimensioni (LLM), URL: databricks.com

CODICE WEAKNESS ENUMERATION (CWE)

CWE (Common Weakness Enumeration) è un elenco standardizzato e sviluppato dalla community che identifica e descrive le vulnerabilità comuni in software, firmware e hardware. Una vulnerabilità è una condizione intrinseca di un sistema che potrebbe compromettere la sicurezza, la funzionalità o la stabilità di un prodotto prodotto.[04]

Le condizioni di vulnerabilità sono nella maggior parte dei casi introdotte dallo sviluppatore durante la scrittura del codice.
La lista di CWE, insieme alle relative tassonomie e agli schemi di classificazione, costituisce un linguaggio comune per identificare e descrivere queste debolezze in termini di CWE.
Le CWE vengono identificate tramite un ID nella forma “CWE-<ID>”, dove ID è semplicemente un numero univoco scelto al momento dell’assegnamento. Ad esempio “CWE-798”. Inoltre il CWE è sempre seguito da un nome descrittivo della debolezza, ad esempio “CWE-798: Use of Hard-Coded Credentials”.

Affinché venga assegnato un CWE-ID ad una vulnerabilità e pubblicato sul sito di CWE, è necessario che includa determinate informazioni quali: nome, riassunto, descrizione, cause principali, possibili mitigazioni, conseguenze comuni, piattaforme applicabili, esempi dimostrativi, esempi di occorrenza, collegamenti con altre CWE, riferimenti.
Molte organizzazioni e sviluppatori utilizzano le CWE per diversi motivi. Per esempio, i software developers e i ricercatori sulla sicurezza usano CWE come un linguaggio comune per discutere su come eliminare e/o mitigare le vulnerabilità dei software nell’architettura, progettazione, codice e implementazione.

04 | CWE, *What is CWE?*, URL: cwe.mitre.org

ZERO SHOT PROMPTING

Questo studio ha l'obiettivo di valutare l'efficacia di un LLM nell'individuare vulnerabilità nel codice, con un focus specifico su LLaMA2:13B, il modello impiegato per condurre l'analisi.
Per farlo, utilizzeremo la tecnica del **Zero Shot Prompting**.

Il Zero Shot Prompting si riferisce alla capacità di un modello linguistico di svolgere un compito senza aver ricevuto esempi specifici durante l'addestramento. Il modello si basa sulla conoscenza generale acquisita da un ampio corpus di dati per affrontare nuovi compiti, comprendendo e seguendo le istruzioni fornite attraverso una semplice descrizione del compito.
È importante specificare che anche all'interno del prompt fornito al modello non sono presenti degli esempi.

I vantaggi dello Zero Shot Prompting sono:

- **Versatilità**: ovvero poter affrontare un'ampia gamma di compiti senza bisogno di addestramenti specifici.
- **Efficienza**: risparmio di tempi e risorse poiché non è necessario addestrare il modelo su ogni nuovo compito.
- **Applicabilità immediata**: permette di utilizzare il modello in più contesti senza ulteriori personalizzazioni

Di contro gli svantaggi sono:

- **Precisione variabile**: la mancanza di esempi specifici può portare a risposte meno accurate
- **Dipendenza dal prompt**: un prompt poco chiaro porta a risooste meno pertinenti, per questo è fondamentale formularlo con precisione.[05]

Il prompt utlizzato in questo studio è stato il seguente:
"You are the best tool to identify security vulnerabilities in source code.
You will be provided with a source code. If it contains any security vulnerabilities, reply with the name and the CWE identifier of each of the CWE vulnerabilities found.
If the code does not contain any vulnerabilities, write "Not Vulnerable".

05 | Strategie di Prompting per LLM: Zero Shot, Few Shot e Chain of Thought, URL: boraso.com

COME FUNZIONA LO ZERO SHOT PROMPTING

Per capire come funziona lo Zero-Shot Prompting, devono essere chiari due aspetti fondamentali: l'addestramento del Large Language Model e il concetto di "Prompt Design"

Addestramento del Large Language Model

Gli LLM vengono addestrati su vasti dataset per sviluppare la capacità di eseguire compiti tramite zero shot prompting.
Il processo di addestramento è il seguente:

- **Collezione di dati**: gli LLM sono addestrati su diversi testi provenienti da diverse fonti. Questi testi possono contare centinaia di miliardi di parole.
- **Tokenization**: il testo viene spezzato in unità più piccole chiamate "token", che possono corrispondere a parole, sottoparole o caratteri.
- **Architettura di rete neurale**: il modello, tipicamente basari su un'architettura transformer, elabora questi token attraverso più livelli di reti neurali.
- **Addestramento orientato alla predizione:** il modello è addestrato per prevedere il prossimo token in una sequenza a partire dal token precedente.
- **Pattern recognition**: attraverso questo processo, il modello impara a riconoscere pattern di un linguaggio, inclusa la grammatica, la sintassi e le relazioni semantiche.
- **Acquisizione delle conoscenze**: il modello costruisce una base di conoscenza ampia che copre vari argomenti e domini.
- **Comprensione del contesto**: il modello impara a riconoscere il contesto e generare risposte appropriate.

Prompt Design

Un prompt ben formulato è essenziale per garantire il successo dell'esecuzione dello Zero Shot Prompting. L'obbiettivo è comunicare in modo chiaro e preciso le intenzioni dell'utente, così da massimizzare l'efficacia delle capacità del modello.
Le strategie per una buona scrittura del prompt sono le seguenti:

- **Istruzioni chiare**: i prompt dovrebbero fornire istruzioni esplicite e inequivocabili che comunichino all'LLM il compito desiderato.
- **Task Framing**: presentare il compito in modo da ottimizzare l'utilizzo delle conoscenze e delle capacità già acquisite dal modello.
- **Context provision**: fornire un contesto rilevante o un background informativo per facilitare al modello la comprensione dei requisiti e dei vincoli del compito.
- **Specifiche di formattazione dell'output**: definire chiaramente il formato della risposta.
- **Evitare ambiguità**: utilizzare un linguaggio chiaro e preciso, evitando istruzioni ambigue che potrebbero essere interpretate in modo errato dal modello.
- **Utilizzo di un liguaggio naturale**: formulare i prompt in modo che risultino fluidi e conversazionali.
- **Iterative refinement**: se i risultati iniziali non soddisfano le aspettative, modifica il prompt aggiungendo dettagli più specifici o ottimizzando il linguaggio. [06]

06 | How Zero-Shot Prompting Works, URL: datacamp.com

LLAMA2 E META

“Llama 2 è una famiglia di modelli linguistici di grandi dimensioni (LLM) pre-addestrati e ottimizzati rilasciata da Meta AI nel 2023 gratuitamente per la ricerca e l'uso commerciale. I modelli AI Llama 2 sono in grado di svolgere una varietà di attività di elaborazione del linguaggio naturale (NLP), dalla generazione di testo alla programmazione del codice.”

Per il pre-addestramento di Llama2, Meta ha impiegato lo stesso approccio descritto in **Touvron et al. (2023)** utilizzato per l’addestramento di Llama1, ma usando un trasformatore autoregressivo ottimizzato con alcune modifiche importanti come: pulizia dei dati più robusta, aggiornamento dei mix di dati, maggiore quantità di dati, lunghezza del contesto raddoppiata, Grouped-Query Attention (GQA).
Per l’addestramento di Llama2, la mole di dati utilizzata è composta da fonti pubblicamente disponibili, escludendo dati dai prodotti o servizi di Meta. Sono stati rimossi dati provenienti da siti noti per conttenere grandi quantità di informazioni personali. Sono stati utilizzati 2 trilioni di token per il training (considerato un compromesso ottimale tra costi e prestazioni). Le fonti di dati più factuali sono state sovracampionate per ridurre le allucinazioni e aumentare la conoscenza del modello. Le differenze principali rispetto a Llama1 sono dunque: aumento della lunghezza del contesto e l’introduzione del Grouped-Query Attention (GQA). Il GQA è una tecnica di ottimizzazione applicata nell’architettura dei trasformatori che mira al miglioramento della scalabilità delle inferenze. I vantaggi principali sono: il raggruppamento delle query per ridurre la complessità computazionale e una miglior efficienza.

Llama2 si presenta in diverse versioni del modello con livelli di capacità crescenti: **7B**, **13B** (utilizzato in questo studio) e **70B**. Questi numeri indicano il numero di parametri nel modello. I parametri sono i componenti fondamentali di un modello di machine learning, che vengono addestrati per apprendere dai dati e rappresentano la complessità e la capacità del modello. [07]

07 | Llama2: Open Foundation and Fine-Tuned Chat Models , URL: ai.meta.com

DATASET UTILIZZATO PER IL CASO DI STUDIO[08]

Per il caso di studio è stato impiegato un dataset composto da file JSON contenenti numerose funzioni scritte in linguaggi C e Python. Ogni funzione presentava una vulnerabilità documentata, con il tipo di CWE specificato nel campo vul_type alla fine della riga corrispondente. Inoltre, il nome del file forniva un'indicazione del tipo di vulnerabilità presente nel codice delle funzioni incluse.
I campi del file json interessanti per questo caso di studio erano:

- **“func_name”**: che indica il nome della funzione analizzata.
- **“func_src_before”**: contente il codice da analizzare.
- **“vul_type”**: la vulnerabilità contenuta nel codice.


```
{
  "func_name": "get_item",
  "func_src_before": "@api.route('/items/<int:item_id>', methods=['GET'])\ndef get_item(item_id):\n    sql = '''SELECT id, name_enus FROM tblDBCItem WHERE id = {} AND auctionable = true;'''.format(item_id)\n    cursor = mysql.connection.cursor()\n    cursor.execute(sql)\n    data = cursor.fetchone()\n\n    if data:\n        item = {}\n        for tup in zip([column[0] for column in cursor.description], data):\n            item[tup[0]] = tup[1]\n    else:\n        return jsonify({\"error\": \"item not found\"}), 404\n\n    return jsonify(item)", "func_src_after": "@api.route('/items/<int:item_id>', methods=['GET'])\ndef get_item(item_id):\n    sql = '''SELECT id, name_enus FROM tblDBCItem WHERE id = %s AND auctionable = true;'''\n    cursor = mysql.connection.cursor()\n    cursor.execute(sql, [item_id])\n    data = cursor.fetchone()\n\n    if data:\n        item = {}\n        for tup in zip([column[0] for column in cursor.description], data):\n            item[tup[0]] = tup[1]\n    else:\n        return jsonify({\"error\": \"item not found\"}), 404\n\n    return jsonify(item)",
  "commit_link": "github.com/cosileone/TimeIsMoneyFriend-API/commit/3d3b5defd26ef7d205915bf643b6b1df90a15e44",
  "file_name": "timf/api/views.py",
  "vul_type": "cwe-089",
  "line_changes": {"deleted": [{"line_no": 3, "char_start": 75, "char_end": 182, "line": "    sql = '''SELECT id, name_enus FROM tblDBCItem WHERE id = {} AND auctionable = true;'''.format(item_id)\n"}, {"line_no": 5, "char_start": 221, "char_end": 245, "line": "    cursor.execute(sql)\n"}], "added": [{"line_no": 3, "char_start": 75, "char_end": 166, "line": "    sql = '''SELECT id, name_enus FROM tblDBCItem WHERE id = %s AND auctionable = true;'''\n"}, {"line_no": 5, "char_start": 205, "char_end": 240, "line": "    cursor.execute(sql, [item_id])\n"}]},
  "char_changes": {"deleted": [{"char_start": 136, "char_end": 138, "chars": "{}"}, {"char_start": 165, "char_end": 181, "chars": ".format(item_id)"}], "added": [{"char_start": 136, "char_end": 138, "chars": "%s"}, {"char_start": 227, "char_end": 238, "chars": ", [item_id]"}]}
}
```


```
Reading file: data/cwe-089.jsonl
FILE ANALYZED: data/cwe-089.jsonl
Function name: get_item
Function code:
@api.route('/items/<int:item_id>', methods=['GET'])
def get_item(item_id):
    sql = '''SELECT id, name_enus FROM tblDBCItem WHERE id = {} AND auctionable = true;'''.format(item_id)
    cursor = mysql.connection.cursor()
    cursor.execute(sql)
    data = cursor.fetchone()

    if data:
        item = {}
        for tup in zip([column[0] for column in cursor.description], data):
            item[tup[0]] = tup[1]
    else:
        return jsonify({"error": "item not found"}), 404

    return jsonify(item)



Final result:
{
    "cwe_id": "CWE-89",
    "cwe_name": "SQL Injection"
}
```

08 | sven, URL: github.com

CODICE UTILIZZATO PER IL CASO DI STUDIO[09]

Il programma utilizzato per il caso di studio è stato progettato per analizzare il codice sorgente, ricavato dal dataset precedentemente presentato, alla ricerca di vulnerabilità di sicurezza CWE utilizzando Llama2.
In particolare, esamina i file sorgente JSONL presi in input, passa le funzioni contenute al loro interno una alla volta al modello, attende la risposta contentente le vulnerabilità presenti nel codice e le inserisce come risposta in un file Excel indicando:

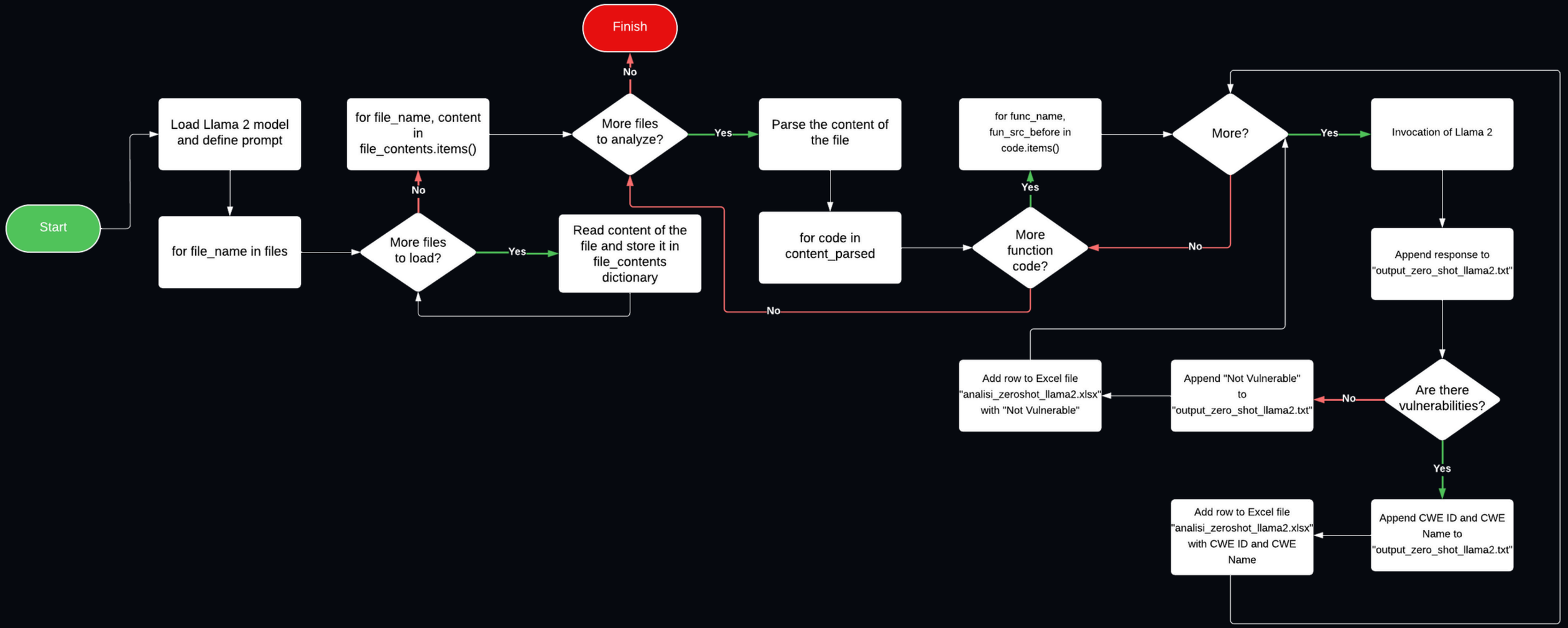
- Percorso del file analizzato ed in particolare il nome della funzione analizzata.
- Lista di CWE presenti nel codice.
- Lista dei nomi delle CWE presenti nel codice.

È importante specificare che nel caso in cui il modello non avesse trovato vulnerabilità nel codice, la risposta sarebbe stata: *“Not Vulnerable”*.

```
# Load the model
llm = Ollama(model="llama2:13b", temperature=0.0)

# Build the prompt template
template = """You are the best tool to identify security vulnerabilities in source code.
You will be provided with a source code. If it contains any security vulnerabilities,
reply with the name and the CWE identifier of each of the CWE vulnerabilities found.
If the code does not contain any vulnerabilities, write "Not Vulnerable".

Source code: ```{code}```

Return the name and the identifier of each CWE security vulnerability found.

{format_instructions}

"""

prompt = PromptTemplate(
    input_variables=["program"],
    template=template,
    partial_variables={"format_instructions": output_parser.get_format_instructions()},
)
```

09 | Uno studio sull’utilizzo degli LLM per identificare vulnerabilità nel codice, URL: github.com

DIAGRAMMA DI FLUSSO DEL CODICE



VALUTAZIONE DELL'ACCURATEZZA DEI RISULTATI

Durante l'esecuzione del programma, ho osservato che i risultati generati dall'analisi di ciascuna funzione non corrispondevano sempre alla vulnerabilità effettivamente presente nel codice fornito in input al modello. Questo mi ha portato, a fine analisi, a sentirmi scoraggiato.

Per comprendere le cause degli errori riscontrati, ho deciso di verificare se i risultati ottenuti dal modello fossero correlati alle gerarchie delle CWE (Common Weakness Enumeration) che stavo analizzando. Durante questa analisi, ho notato che alcune vulnerabilità rilevate dal modello corrispondevano a CWE "genitore" piuttosto che a quelle specifiche attese.
Successivamente, ho utilizzato un programma di valutazione per calcolare metriche di performance come precisione, richiamo e F1-Score. Questo ha confermato che, sebbene il modello non fosse preciso quanto sperato, alcuni risultati erano comunque plausibili, in quanto i CWE "genitore" rilevati rappresentano categorie generali che comprendono le vulnerabilità specifiche attese. Questi risultati suggeriscono che il modello possedeva una comprensione parziale delle vulnerabilità, ma mancava della granularità necessaria per identificare correttamente le CWE specifiche.

Ho cercato anche di capire quali fossero i motivi di queste imprecisioni, ricercando un po' sono arrivato a concludere che i motivi potessero essere i seguenti:

- **Limitazioni di Llama2**: il modello non è stato addestrato specificatamente per il rilevamento di vulnerabilità all'interno di codice.
- **Interpretazione limitata del codice**: i modelli linguistici non eseguono il codice e non comprendono in modo nativo il comportamento dinamico. Ad esempio: vulnerabilità come race conditions e buffer overflow avvengono a tempo di esecuzione e possono essere difficili da rilevare.
- **Complessità del codice analizzato**: funzioni scritte in C e Python, come quelle contenute nel dataset da analizzare possono avere una struttura complessa, con riferimenti incrociati, dipendenze esterne e contesti che non sono direttamente evidenti. Llama probabilmente non è riuscito a comprendere tali complessità basandosi solo sul testo fornito.
- **Limiti del Prompt Zero-Shot**: dato che il prompt non includeva esempi concreti di vulnerabilità né spiega come identificarle, Llama2 si è affidato alla sua conoscenza e sebbene sia una modello molto potente, la mancanza di contesto aggiuntivo riduce l'efficacia della sua risposta
- **Ambiguità nei risultati**: Llama ha restituito spesso vulnerabilità comuni ma non effettivamente presenti nel codice specifico analizzato.

METRICHE UTILIZZATE PER IL CALCOLO DELLE PERCENTUALI DI RIUSCITA

Le metriche utilizzate per utilizzate per valutare l'efficacia del programma sono le seguenti:

- **Precisione/Precision**: definita come il rapporto tra i **True Positive(TP)** e la somma dei **True Positive** e i **False Positive(FP)**, misura la precisione o l'accuratezza del modello. La metrica rivela dunque quante delle stime del modello siano effettivamente etichettate correttamente. **TP/(TP+FP)**
- **Richiamo/Recall**: definita come il rapporto tra i **True Positive(TP)** e la somma dei **True Positive(TP)** e i **False Negative(FN)**, misura la capacità del modello di rilevare le vulnerabilità corrette effettive. **TP/(TP+FN)**
- **F1-Score**: necessaria quando si cerca un equilibrio tra precisione e richiamo.
  Definita come: **2 x ((Precision x Recall)/(Precision+Recall))**[10]

10 | Metriche di valutazione, URL: learn.microsoft.com

PERCENTUALI DI RIUSCITA (RILEVAMENTI EFFETTIVI)

CWE-ID	Precision	Recall	F1-Score
CWE-022	50%	5.6%	10,1%
CWE-078	22,3%	32,4%	26,4%
CWE-079	0%	0%	0%
CWE-089	91,4%	17,2%	29%
CWE-125	60%	6,3%	11,3%

CWE-ID	Precision	Recall	F1-Score
CWE-190	0%	0%	0%
CWE-416	7,1%	9,5%	8,2%
CWE-476	17,4%	5,2%	8%
CWE-787	16%	12,7%	14,1%
Not Vulnerable	0%	0%	0%

PERCENTUALI DI RIUSCITA (RILEVAMENTI CORRELATI)

CWE-ID	Precision	Recall	F1-Score
CWE-022	4,11%	5.6%	4,8%
CWE-078	18,6%	32,4%	23,7%
CWE-079	0%	0%	0%
CWE-089	20,6%	17,2%	18,8%
CWE-125	12,7%	7%	8,97%

CWE-ID	Precision	Recall	F1-Score
CWE-190	1%	2,4%	1,4%
CWE-416	6,8%	9,5%	7,95%
CWE-476	17,4%	5,2%	8%
CWE-787	6%	12,7%	8,14%

CONCLUSIONI

Llama2:13B purtroppo ha mostrato una capacità **limitata** nell'identificare vulnerabilità. In alcuni casi è riuscito ad individuare la vulnerabilità effettivamente presente nel codice, mentre in altri ha rilevato delle vulnerabilità correlate. Tuttavia non sono mancati errori, con rilevamenti totalmente errati.
È probabile dunque che il modello possieda una **comprensione teorica**, ma non abbastanza dettagliata per un'applicazione pratica precisa.
Le metriche calcolate: **precisione**, **richiamo** e **F1-Score**, indicano una **discrepanza** tra i rilevamenti effettivi e le vulnerabilità specifiche presenti nel codice, è evidente la **necessità di miglioramenti**.
Questo può però essere giustificato dal fatto che Llama2 **non è stato addestrato particolarmente al rilevamento di vulnerabilità CWE**, limitandosi a una conoscenza generale.
Inoltre Llama2, essendo un LLM, esegue un'analisi **statica** del testo ricevuto in input (in questo caso il codice) e non può comprendere il comportamento **dinamico** del codice, che credo possa essere essenziale per individuare vulnerabilità come buffer overflow e race conditions.
In aggiunta la tecnica dello Zero Shot Prompting potrebbe non aver aiutato il modello all'esecuzione del suo task, poiché ne comporta un approccio basato su nessun esempio.

PROSPETTIVE FUTURE

L'analisi delle vulnerabilità nel codice potrebbe essere significativamente migliorata attraverso un'ottimizzazione mirata, sfruttando dataset simili a quelli utilizzati in questo studio per affinare le capacità del modello.
Un approccio efficace potrebbe includere l'uso di prompt avanzati mediante la tecnica del Few-Shot Prompting, che fornisce esempi concreti di vulnerabilità e CWE specifiche per aumentare la precisione delle risposte.
Inoltre, l'aggiunta di un contesto più dettagliato nel prompt o l'adozione di tecniche ibride, integrando strumenti di analisi statica del codice, potrebbe ulteriormente migliorare l'accuratezza, sfruttando i punti di forza di ciascuna metodologia.
Ottimizzando questi aspetti, sarebbe possibile sviluppare pipeline automatizzate in grado di integrare modelli come Llama2 per il rilevamento di vulnerabilità in tempo reale durante il ciclo di sviluppo del software, supportando in modo efficace il lavoro dei team di sviluppo.