Project Documentation:

AI-Powered Vulnerability Analysis & Red Team Simulation Engine

Author: Anmol Saini

1. Project Overview

1.1. Problem Statement:
Organizations face challenges in managing and interpreting vulnerability data scattered across various threat intelligence feeds (NVD, CISA KEV), exploit databases (Exploit-DB), tactical frameworks (MITRE ATT&CK), and internal scan reports. Consolidating this information and querying it efficiently in natural language is difficult.

1.2. Solution:
This project implements a Retrieval-Augmented Generation (RAG) system using a fine-tuned Large Language Model (LLM) to address this challenge. It involves:

- **Centralized Knowledge Base:** Ingesting data from multiple cybersecurity sources into specialized vector databases (RAG databases).
- **Contextual Understanding:** Utilizing a fine-tuned LLM (based on Mistral-7B) specialized in cybersecurity concepts and instruction following.
- **Intelligent Querying:** Allowing users to ask natural language questions about vulnerabilities, exploits, or attack tactics.
- **Grounded Responses:** Retrieving relevant information from the knowledge bases and the user's uploaded report to generate accurate, context-aware answers, avoiding hallucination.

1.3. Core Technologies:

- **Execution Environment:** Google Colab (GPU recommended)
- **Storage:** Google Drive
- **Core Language:** Python 3
- **LLM Framework:** Hugging Face Transformers, Unsloth (for optimized fine-tuning and inference)
- **RAG Components:** LangChain, FAISS (vector store), Sentence

Transformers (embedding model)
- **Data Handling:** Pandas, PyPDF2, Requests, JSON, mitreattack-python

2. System Architecture

The system operates in distinct phases: Setup, Data Ingestion (RAG DB Creation), Model Fine-Tuning, and Runtime Querying.

- **Setup:** Preparing the Google Drive environment.
- **Data Ingestion (Offline Phase):**
  - Downloading data from NVD, CISA, Exploit-DB, MITRE ATT&CK.
  - Processing and cleaning the data.
  - Chunking text into manageable pieces.
  - Generating vector embeddings using a Sentence Transformer model.
  - Storing these embeddings in separate FAISS vector databases persisted to Google Drive. (Handled by rag_1_b.ipynb, rag_2.ipynb, and potentially another script for NVD).

- **Model Fine-Tuning (Offline Phase):**
  - Loading a pre-trained base LLM (Mistral-7B variant via Unsloth).
  - Preparing a custom dataset (fine_tuning_dataset.json) with instruction/response pairs relevant to cybersecurity analysis.
  - Using Parameter-Efficient Fine-Tuning (PEFT) with LoRA via Unsloth to create specialized adapter weights.
  - Saving these adapter weights to Google Drive. (Handled by fine_tuning.ipynb).

- **Runtime Querying (Online Phase):**
  - User uploads their specific vulnerability report (PDF).
  - The report text is extracted.
  - The user submits a natural language query.
  - An initial classification step (using the *base* LLM) determines the query intent.
  - Relevant RAG databases are queried based on intent.
  - Context is retrieved from the RAG databases and the uploaded report.
  - A detailed prompt (including persona, instructions, report context, RAG

context, and the query) is constructed.
    - The prompt is sent to the *fine-tuned* LLM (base model + LoRA adapters) for response generation.
    - The generated response is displayed to the user. (Handled by final.ipynb).

3. Environment Setup (Crucial First Step)

A specific Google Drive structure is mandatory for the Colab notebooks to locate all necessary files.

Step 1: Create Project Root Folder

1. Navigate to your Google Drive (My Drive).
2. Create a new folder named exactly: **Vulnerability_Project**

Step 2: Create Subfolders
Inside Vulnerability_Project, create the following five subfolders:

1. **colab_notebooks**: For storing all .ipynb Colab files (rag_1_b.ipynb, rag_2.ipynb, fine_tuning.ipynb, final.ipynb, etc.).
2. **rag_database**: This will contain subfolders for *each* RAG database built by the ingestion scripts.
    - Create a subfolder inside rag_database named rag_1a_nvd (for NVD data - *adjust name if different*).
    - Create a subfolder inside rag_database named rag_1b_cisa_exploitdb (for CISA/Exploit-DB data).
    - Create a subfolder inside rag_database named rag_2_mitre_metasploit (for MITRE/Metasploit data).
3. **models**: For storing the fine-tuned LLM adapter weights.
    - Place the .safetensors files (e.g., adapter_model.safetensors) and associated config files (adapter_config.json, README.md, etc.) generated by fine_tuning.ipynb directly inside this folder.
4. **input_reports**: This is where the end-user will upload their *target* vulnerability scan report (e.g., scan_results.pdf) during the runtime phase (final.ipynb).
5. **fine_tuning_data**: For storing the dataset used for fine-tuning (fine_tuning_dataset.json).

Visual Structure:

```
My Drive/
└── Vulnerability_Project/
    ├── colab_notebooks/
    │   ├── rag_1_b.ipynb
    │   ├── rag_2.ipynb
    │   ├── fine_tuning.ipynb
    │   └── final.ipynb
    │   └── (rag_1a_nvd.ipynb - if applicable)
    ├── rag_database/
    │   ├── rag_1a_nvd/
    │   │   └── (FAISS index files: index.faiss, index.pkl)
    │   ├── rag_1b_cisa_exploitdb/
    │   │   └── (FAISS index files: index.faiss, index.pkl)
    │   └── rag_2_mitre_metasploit/
    │       └── (FAISS index files: index.faiss, index.pkl)
    ├── models/
    │   ├── adapter_model.safetensors
    │   ├── adapter_config.json
    │   └── README.md
    │   └── (...)
    ├── input_reports/
    │   └── (User uploads reports here later, e.g., scan_results.pdf)
    └── fine_tuning_data/
        └── fine_tuning_dataset.json
```

Step 3: Upload Provided Files

- Upload all .ipynb files into Vulnerability_Project/colab_notebooks/.
- Upload your pre-built RAG database files into the corresponding subfolders within Vulnerability_Project/rag_database/.
- Upload the fine-tuned adapter files (.safetensors, adapter_config.json, etc.) into Vulnerability_Project/models/.
- Upload the fine_tuning_dataset.json file into Vulnerability_Project/fine_tuning_data/.

4. 🛠️ Detailed Build Steps & Component Explanation

This section details how each major component was built. Run these steps (or provide the pre-built outputs) before running the final application.

4.1. RAG Database Creation

- **Purpose:** To convert diverse cybersecurity data sources into a structured, searchable vector format for efficient retrieval based on semantic similarity.
- **Embedding Model Used:** sentence-transformers/all-MiniLM-L6-v2 (Chosen for its balance of performance and resource efficiency).
- **Vector Store:** FAISS (Efficient similarity search library).
- **Framework:** LangChain

4.1.1. RAG Database #1a: NVD (National Vulnerability Database)
* Notebook: (Assuming rag_1a_nvd.ipynb or similar - Please provide details if different)
* Input Data: NVD CVE JSON Feeds (Downloaded via script or manually).
* Steps:
1. Load Data: Read NVD JSON files.
2. Process: Extract relevant fields (CVE ID, description, CVSS scores, CWEs). Combine into a structured text format for each CVE entry.
3. Chunking: Use RecursiveCharacterTextSplitter (e.g., chunk_size=1000, chunk_overlap=100) to break down long descriptions while maintaining context.
4. Embedding: Generate vector embeddings for each chunk using the HuggingFaceEmbeddings wrapper for the Sentence Transformer.
5. Vector Store Creation: Create a FAISS index from the embedded chunks.
6. Save: Persist the FAISS index files (index.faiss, index.pkl) to the designated Google Drive folder: Vulnerability_Project/rag_database/rag_1a_nvd/.
4.1.2. RAG Database #1b: CISA KEV & Exploit-DB
* Notebook: rag_1_b.ipynb
* Input Data: CISA Known Exploited Vulnerabilities JSON feed, Exploit-DB files_exploits.csv.
* Steps:
1. Install Dependencies: Installs requests, pandas, langchain, faiss-cpu, sentence-transformers.
2. Download Data: Fetches the latest CISA KEV JSON and Exploit-DB CSV via HTTP requests.

3. Process CISA KEV: Iterates through vulnerabilities, formats key information (CVE ID, name, description, action) into text, creates LangChain Document objects.
4. Process Exploit-DB: Reads the CSV using Pandas, formats relevant columns (description, ID, file path, type, platform, associated codes) into text, creates LangChain Document objects.
5. Combine Documents: Appends documents from both sources into a single list.
6. Chunking: Uses RecursiveCharacterTextSplitter (chunk_size=1000, chunk_overlap=100) on the combined documents.
7. Embedding: Initializes HuggingFaceEmbeddings with sentence-transformers/all-MiniLM-L6-v2.
8. Mount Drive: Connects to Google Drive.
9. Vector Store Creation: Creates a FAISS index from the chunked documents and embeddings.
10. Save: Persists the FAISS index files to Google Drive: Vulnerability_Project/rag_database/rag_1b_cisa_exploitdb/.
4.1.3. RAG Database #2: MITRE ATT&CK & Metasploit Modules
* Notebook: rag_2.ipynb
* Input Data: MITRE ATT&CK Enterprise data (via mitreattack-python), Exploit-DB files_exploits.csv.
* Steps:
1. Install Dependencies: Installs mitreattack-python, pandas, openpyxl, requests, langchain, etc.
2. Download/Process MITRE: Uses attackToExcel.export to download and save ATT&CK data as Excel files. Reads the enterprise-attack-techniques.xlsx file using Pandas. Iterates through techniques, formatting ID, name, description, and tactics into text, creates LangChain Document objects.
3. Download/Process Metasploit: Fetches the Exploit-DB CSV (if not already present). Filters the DataFrame for rows containing "Metasploit" in the description. Formats relevant columns (description, file path, type, platform, CVEs) into text, creates LangChain Document objects.
4. Combine Documents: Appends documents from both sources.
5. Chunking: Uses RecursiveCharacterTextSplitter (chunk_size=1000, chunk_overlap=100).
6. Embedding: Initializes HuggingFaceEmbeddings.
7. Mount Drive: Connects to Google Drive.

8. Vector Store Creation: Creates a FAISS index.
9. Save: Persists the FAISS index files to Google Drive: Vulnerability_Project/rag_database/rag_2_mitre_metasploit/.

4.2. Model Selection and Fine-Tuning

- **Purpose:** To adapt a general-purpose LLM to better understand cybersecurity terminology, follow specific instruction formats (like generating JSON or Markdown), and adopt the desired persona (e.g., a helpful cyber analyst).
- **Notebook:** fine_tuning.ipynb
- **Base Model:** unsloth/mistral-7b-instruct-v0.2-bnb-4bit (Selected for strong instruction-following capabilities and Unsloth's 4-bit quantization for reduced memory/faster training on Colab GPUs).
- **Fine-Tuning Technique:** Parameter-Efficient Fine-Tuning (PEFT) using Low-Rank Adaptation (LoRA) via the Unsloth library. This trains only a small fraction of the model's parameters, significantly reducing computational requirements.
- **Input Data:** fine_tuning_dataset.json (Located in Vulnerability_Project/fine_tuning_data/). This file contains structured examples of prompts and desired responses tailored for cybersecurity tasks. (*User must provide this file.*)
- **Steps:**
    1. **Install Unsloth:** Installs the Unsloth library, which patches Hugging Face Transformers for faster training and lower memory usage.
    2. **Upload Dataset:** Prompts the user to upload fine_tuning_dataset.json directly into the Colab environment for this session.
    3. **Load Base Model & Tokenizer:** Uses Unsloth's FastLanguageModel.from_pretrained to load the 4-bit quantized base model and its corresponding tokenizer. Unsloth applies performance patches automatically.
    4. **Prepare Model for PEFT:** Configures the base model for LoRA training using FastLanguageModel.get_peft_model. This involves specifying LoRA parameters like r (rank), lora_alpha, target_modules, lora_dropout, etc. (*These parameters are typically set within the Unsloth defaults or the notebook code*).
    5. **Load & Format Dataset:** Loads the uploaded fine_tuning_dataset.json

using Hugging Face datasets. Defines a formatting_func to structure each data sample into the required input format for the model (often following a specific instruction template like Alpaca or ChatML).

6. **Configure Trainer:** Sets up the Hugging Face SFTTrainer (Supervised Fine-tuning Trainer) provided by the trl library.
    - Passes the PEFT-prepared model and tokenizer.
    - Provides the formatted dataset.
    - Specifies TrainingArguments:
        - per_device_train_batch_size, gradient_accumulation_steps (controls effective batch size).
        - warmup_steps, num_train_epochs, learning_rate, lr_scheduler_type.
        - fp16/bf16 (mixed-precision training, automatically handled by Unsloth).
        - logging_steps, optim (e.g., adamw_8bit), weight_decay.
        - output_dir (local temporary directory for checkpoints).
        - report_to="none" (disables Weights & Biases logging if not needed).
7. **Start Training:** Calls trainer.train(). Unsloth's optimizations significantly speed up this process compared to standard Hugging Face training. Training progress (loss) is printed to the output.
8. **Save Adapters:** After training, uses model.save_pretrained(new_model_name) to save *only* the trained LoRA adapter weights (not the entire model) to a local folder (e.g., mistral-7b-cyber-analyst). This typically includes adapter_model.safetensors and adapter_config.json.
9. **Copy to Drive:** Mounts Google Drive and copies the contents of the saved adapter folder to the designated Google Drive location: Vulnerability_Project/models/. Uses rsync for robustness.

5. 🚀 Final Application Execution (final.ipynb)

This notebook integrates all components for the end-user interaction.

Step 1: Setup & Installation

1. **Open Notebook:** Open final.ipynb from Vulnerability_Project/colab_notebooks/.

2. **Run Cell 1 (Install):** Installs unsloth, langchain, faiss-cpu, sentence-transformers, PyPDF2, etc. **Crucially, restart the Colab runtime after this cell finishes.**

Step 2: Load Models and Databases

1. **Run Cell 2 (Load Base Model & Mount Drive):**
   - Mounts Google Drive: drive.mount('/content/drive').
   - Loads the *base* LLM (unsloth/mistral-7b-instruct-v0.2-bnb-4bit) and tokenizer using Unsloth's FastLanguageModel.from_pretrained. This ensures performance patches are applied. Sets pad_token.
2. **Run Cell 2.5 (Load Adapters):**
   - Specifies the path to the saved adapters: adapter_path = "/content/drive/My Drive/Vulnerability_Project/models/" (*Ensure this matches your setup*).
   - Uses PeftModel.from_pretrained(base_model, adapter_path) to load the trained LoRA weights and apply them *on top of* the loaded base model. The resulting finetuned_model is the specialized model ready for inference.
3. **Run Cell 3 (Load RAG DBs):**
   - Initializes the HuggingFaceEmbeddings model (sentence-transformers/all-MiniLM-L6-v2).
   - Specifies the paths to the *three* RAG database folders within Google Drive (e.g., db_path_1a = "/content/drive/My Drive/Vulnerability_Project/rag_database/rag_1a_nvd/", etc. - *Verify these paths*).
   - Uses FAISS.load_local for each database, passing the embedding_model and allow_dangerous_deserialization=True. Stores them in variables like vector_db_1a, vector_db_1b, vector_db_2.

Step 3: Define Core Logic

1. **Run Cell 4 (Query Router & Retriever):**
   - Defines classify_query(user_query): This function takes the user's input, creates a prompt asking the *base* LLM to classify the intent (vulnerability_details, exploit_commands, attack_tactics), runs inference, and returns the classification.
   - Defines retrieve_context(user_query, k=7): This function first calls

classify_query. Based on the intent, it selects the appropriate FAISS vector database(s) (vector_db_1a, vector_db_1b, vector_db_2), creates retrievers using .as_retriever(), invokes them with the user query using .invoke(), collects the retrieved Document objects, and formats their page_content into a single context string.

Step 4: Upload User Report

1. **Run Cell 5 (Upload PDF):**
    - Uses google.colab.files.upload() to present a file upload widget.
    - **Action Required:** The user must click "Choose Files" and select their target vulnerability report (a **text-based** PDF).
    - The script reads the uploaded PDF using PyPDF2, extracts text page by page, and stores it in the combined_report_content variable.
2. **Run Cell 6 (Chunk Report):**
    - Takes the combined_report_content.
    - Uses RecursiveCharacterTextSplitter (configured potentially with smaller chunks like chunk_size=1500, chunk_overlap=250 optimized for extraction tasks if applicable in your final.ipynb logic) to split the report text into report_chunks.

Step 5: Vulnerability Extraction (Optional but likely present)

1. **Run Cell 7.5 (Hybrid Extraction):**
    - This crucial cell processes the report_chunks from the uploaded PDF.
    - **Regex Pre-scan:** It first uses simple regex patterns (CVE_PATTERN, CVSS_PATTERN, keywords) to quickly find potential indicators within each chunk (extract_with_regex).
    - **LLM JSON Extraction:** It then constructs a prompt for the *base* LLM, instructing it to extract specific vulnerability details (title, CVEs, severity, description, remediation) from the chunk and output them as a JSON array. It includes regex highlights in the prompt to guide the LLM.
    - **JSON Repair:** Includes a repair_json function to handle potentially malformed JSON output from the LLM.
    - **Fallback:** If JSON parsing fails or confidence is low, it uses the initial regex matches to generate basic markdown table rows (generate_regex_rows).
    - **Aggregation:** Collects all extracted/generated vulnerability data (JSON

objects).
    - **Deduplication & Formatting:** Removes duplicate findings and formats the aggregated data into a Markdown table (summarized_vulnerability_report.md). This summary becomes the primary context for the chatbot in the next step.

Step 6: Interact with Chatbot

1. **Run Final Cell (Chatbot UI):**
    - Loads the summarized_vulnerability_report.md created in the previous step as the primary vulnerability_context. Includes fallback logic if the summary is empty.
    - Defines get_chatbot_response(user_query):
        - Checks if the query contains a CVE or exploit keywords to trigger RAG using retrieve_context.
        - Constructs the final, detailed prompt including the persona, workflows (Attack Chain, Single Exploit, Explanation), few-shot examples, the retrieved RAG context (rag_context), the summarized report context (vulnerability_context), and the user_query.
        - Runs inference using the **finetuned_model** (Base + Adapters) and the tokenizer. It uses .generate() with parameters like max_new_tokens, temperature, do_sample, etc. **Crucially, it decodes *only* the newly generated tokens, excluding the input prompt.**
        - Includes fallback logic for very short or potentially erroneous responses.
    - Sets up an ipywidgets interface (input_box, chat_history).
    - The handle_submit function takes the query from the input box, calls get_chatbot_response, and displays the query and the generated Markdown response in the chat history output widget.
2. **Ask Questions:** Type your query into the text box and press Enter. The fine-tuned model will generate a response based on the RAG context, the summarized report, and its specialized training.

6. File Component Descriptions

- **.ipynb (Colab Notebooks)**
    - rag_1_b.ipynb: Builds the RAG DB from CISA KEV and Exploit-DB data.

    - rag_2.ipynb: Builds the RAG DB from MITRE ATT&CK and Metasploit module data.
    - (rag_1a_nvd.ipynb): (If applicable) Builds the RAG DB from NVD data.
    - fine_tuning.ipynb: Performs PEFT/LoRA fine-tuning of the base Mistral-7B model using fine_tuning_dataset.json. Saves adapter weights.
    - final.ipynb: The main application notebook. Loads the base model, applies fine-tuned adapters, loads all RAG DBs, handles user report upload and processing, includes the query router and RAG retrieval, and provides the interactive chatbot interface.
- **rag_database/ (Subfolders with FAISS Index Files)**
    - Contains the pre-computed vector stores (index.faiss, index.pkl) for each data source. These are binary files used for fast similarity searches.
- **models/ (.safetensors, .json, .md)**
    - Contains the fine-tuned LoRA adapter weights (adapter_model.safetensors) and configuration (adapter_config.json) saved by fine_tuning.ipynb. These files are much smaller than the full base model.
- **fine_tuning_data/fine_tuning_dataset.json**
    - The user-provided dataset containing prompt/response pairs used to fine-tune the LLM. The quality of this dataset directly impacts the fine-tuned model's performance.
- **input_reports/**
    - Empty initially. Users place their target vulnerability scan PDF reports here when running final.ipynb.
- **summarized_vulnerability_report.md (Generated by final.ipynb)**
    - A Markdown file containing the table of vulnerabilities extracted from the user's uploaded report by the hybrid regex+LLM process in final.ipynb. Used as context for the chatbot.

7. Troubleshooting & Notes

- **Google Drive Paths:** Double-check all Google Drive paths defined in the notebooks match the exact structure created in Section 3. This is the most common error source. Ensure correct folder names (including spaces) and correct root path (/content/drive/My Drive/).
- **Colab Runtime:** Ensure you are using a GPU runtime (e.g., T4, L4, A100) for both fine-tuning and inference. Restart the runtime after installing libraries

in final.ipynb.

- **Dependencies:** If notebooks fail, check that all required libraries were installed correctly in the first cell of each notebook. Version conflicts might occasionally arise; consult error messages.
- **File Uploads:** Ensure fine_tuning_dataset.json is uploaded correctly in fine_tuning.ipynb and the target PDF report is uploaded in final.ipynb when prompted.
- **Unsloth:** Unsloth patches libraries for performance. If you encounter issues, ensure the correct version of unsloth is installed (pip install "unsloth[colab-new]") and that you restart the runtime after installation.
- **PDF Text Extraction:** The final.ipynb currently uses PyPDF2, which only works for text-based PDFs. Image-based PDFs will require OCR (Optical Character Recognition) libraries (tesseract, pytesseract, pdf2image) to be installed and integrated, which adds complexity.
- **Memory:** Fine-tuning and loading large models, even quantized ones, require significant GPU RAM. If you run into CUDA Out-of-Memory errors, try reducing batch sizes (per_device_train_batch_size, gradient_accumulation_steps) in fine_tuning.ipynb or consider using a Colab Pro subscription for access to higher-memory GPUs.