

**USER MANUAL**

1. **Prerequisites**

Software Requirements

Python: Version 3.9 or higher

Flask: Web framework for Python

SQLite: Lightweight database included in Python standard library

Additional Python Packages: Listed in `requirements.txt`

Installation Steps

1. Install Python:

Download and install Python from the official website [python.org](https://www.python.org/).

2. Install Flask:

* pip install flask

3. Create a Project Directory:

Create a directory for your project and navigate into it:

* mkdir breachbuster
* cd breachbuster

4. Create a Virtual Environment (optional but recommended):

Create and activate a virtual environment:

python m venv venv

source venv/bin/activate # On Windows use `venv\Scripts\activate`

5. Install Required Packages:

Following packages are listed in `requirements.txt` file in the project directory:

1. Flask
2. pandas
3. numpy
4. nvdlib
5. pyxploitdb
6. nltk
7. scikitlearn
8. requests
9. beautifulsoup4
10. FlaskSocketIO
11. paramiko

Install the packages listed in `requirements.txt`:

pip install r requirements.txt

Additional Setup for NLTK

After installing NLTK, you need to download additional resources. Run the following commands in your Python environment:

* import nltk
* nltk.download('punkt')
* nltk.download('wordnet')

By following these steps, you will have set up the necessary environment and installed all required packages to run your Flask web application. Make sure to activate the virtual environment each time you work on the project to ensure all dependencies are correctly managed.

1. **Project Structure**

Directory Layout

Create the following structure in your project directory:

Webapp/

│

├── \_\_pycache\_\_/

│

├── backup/

│

├── js/

│

├── static/

│ ├── chatbot.js

│ ├── logo.png

│ └── styles.css

│

├── templates/

│ ├── admin.html

│ ├── chatbot.html

│ ├── home.html

│ ├── login.html

│ ├── register.html

│ ├── session\_history.html

│ └── webshell.html

│

├── admin

├── chat\_history.db

├── cve\_exploit\_search

├── database

├── error

├── exploits\_and\_vulnerabilities\_scraper

├── intent\_model.pkl

├── intent\_model

├── intents.xlsx

├── logo.png

├── main.py

├── pentesting\_report.pdf

├── phase\_responses.json

├── pt\_report\_generate

├── test

├── url\_validator

├── users.json

├── webshell

├── requirements.txt

Files Description

**\_\_pycache\_\_/**: Directory for Python bytecode files (usually autogenerated, can be ignored).

**backup/**: Directory for backup files (if needed).

**js/**: Directory for JavaScript files (if any).

**static/**: Directory for static files like CSS and JavaScript.

chatbot.js: JavaScript for handling clientside interactions.

logo.png: Logo image file.

styles.css: CSS styles for the web pages.

**templates/**: Directory for HTML template files.

admin.html: HTML template for the admin page.

chatbot.html: HTML template for the chatbot page.

home.html: Main HTML template for the homepage.

login.html: HTML template for the login page.

register.html: HTML template for the registration page.

session\_history.html: HTML template to display a specific session's chat history.

webshell.html: HTML template for the web shell page.

**admin**: Adminrelated code (not specified, could be a Python file or directory).

**chat\_history.db**: SQLite database file.

**cve\_exploit\_search**: CVE exploit searchrelated code (likely a Python file or module).

**database**: Databaserelated code (likely a Python file or module).

**error**: Error handling code (likely a Python file or module).

**exploits\_and\_vulnerabilities\_scraper**: Code for scraping exploits and vulnerabilities (likely a Python file or module).

**intent\_model.pkl**: Pickle file containing the intent model.

**intent\_model**: Intent modelrelated code (likely a Python file or module).

**intents.xlsx**: Excel file containing intents.

**logo.png**: Logo image file.

**main.py**: Main Python application file (likely contains the Flask app).

**pentesting\_report.pdf**: Pentesting report in PDF format.

**phase\_responses.json**: JSON file containing phase responses.

**pt\_report\_generate**: Code for generating pentesting reports (likely a Python file or module).

**test**: Testrelated code (likely a Python file or module).

**url\_validator**: URL validationrelated code (likely a Python file or module).

**users.json**: JSON file containing user information.

**webshell**: Web shellrelated code (likely a Python file or module).

**requirements.txt**: List of required Python packages.

1. **Setting Up the Database**

Database Initialization

The database for this project is set up using SQLite, a lightweight and easytouse database system that comes bundled with Python. The create\_database function in app.py is responsible for initializing the database and ensuring that the necessary tables are created if they don't already exist. This ensures that the application has the required structure to store chat sessions and their corresponding histories.

Here is a stepbystep explanation of the database setup process:

1. Database Connection:
   * The function establishes a connection to the SQLite database file named chat\_history.db. If the file does not exist, SQLite will create it automatically.

* conn = sqlite3.connect(DATABASE)

1. Creating a Cursor Object:
   * A cursor object is created using the connection. This cursor will be used to execute SQL commands.

* c = conn.cursor()

1. Creating the chat\_sessions Table:
   * The chat\_sessions table stores information about each chat session. It includes:
     + session\_id: An autoincremented integer serving as the primary key.
     + start\_time: A text field recording the start time of the chat session.

* c.execute('''
* CREATE TABLE IF NOT EXISTS chat\_sessions (
* session\_id INTEGER PRIMARY KEY AUTOINCREMENT,
* start\_time TEXT
* )
* ''')

1. Creating the chat\_history Table:
   * The chat\_history table stores individual messages exchanged during each chat session. It includes:
     + id: An autoincremented integer serving as the primary key.
     + session\_id: An integer linking the message to a specific chat session.
     + sender: A text field indicating who sent the message (e.g., user or bot).
     + message: A text field containing the message content.
     + timestamp: A text field recording the time when the message was sent.
   * The session\_id in this table is a foreign key referencing the session\_id in the chat\_sessions table, ensuring referential integrity.

* c.execute('''
* CREATE TABLE IF NOT EXISTS chat\_history (
* id INTEGER PRIMARY KEY AUTOINCREMENT,
* session\_id INTEGER,
* sender TEXT,
* message TEXT,
* timestamp TEXT,
* FOREIGN KEY (session\_id) REFERENCES chat\_sessions (session\_id)
* )
* ''')

1. **Committing Changes and Closing the Connection**:
   * After executing the SQL commands, the changes are committed to the database, and the connection is closed.

* conn.commit()
* conn.close()

By following these steps, the create\_database function ensures that the necessary tables are available in the database before the application starts handling chat sessions and storing chat histories. This setup is crucial for maintaining organized and retrievable chat data.

1. **Running the Flask Application**

Steps to Run the Application

1. Navigate to the Project Directory:

sh

cd breachbuster

2. Activate the Virtual Environment (if you created one):

sh

source venv/bin/activate On Windows use `venv\Scripts\activate`

3. Run the Flask Application:

sh

python app.py

4. Access the Application:

Open a web browser and go to `http://127.0.0.1:5000`.

Detailed Breakdown:

1. Navigate to the Project Directory

* Ensure you are in the directory where your project files are located.
* Use the `cd` (change directory) command to navigate to your project directory.

2. Activate the Virtual Environment

* Activating a virtual environment ensures that the dependencies installed for this project do not affect other projects.
* On Windows, the command to activate the virtual environment is slightly different:
* sh
* venv\Scripts\activate

3. Run the Flask Application

* Execute the `python app.py` command to start the Flask development server.

You should see output indicating that the server is running, similar to:

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

This output confirms that the server is listening for incoming requests.

4. Access the Application

* Open a web browser of your choice (e.g., Chrome, Firefox, Edge).
* Enter the URL `http://127.0.0.1:5000` in the address bar.
* You should see the homepage of your Flask application.

Features on the Homepage:

* Logo Ribbon: Displays the logo, "Launch Kali" button, and "Logout" button.

1. The "Launch Kali" button triggers a JavaScript function `launchKali()`.
2. The "Logout" button triggers a JavaScript function `logout()`.

* Chat Container: Main area for displaying chat messages and user input.

1. The chat box will display messages from both the user and the bot.
2. A text input field allows users to type their messages.
3. A "Send" button submits the user's message to the server for processing.

* Side Bars: Provide buttons for different phases of the pentest, saving chat, and starting a new chat.

1. Phase Buttons: Shows colors for different phases of the pentest process.
2. "SAVE CHAT" Button: Saves the current chat session to the database.
3. "NEW CHAT" Button: Starts a new chat session by resetting global variables and reloading the page.

Debugging Tips:

Check Console Output: Monitor the terminal or command prompt where you ran `python app.py` for any error messages or logs.

Browser Developer Tools: Use the developer tools (F12) in your web browser to check for JavaScript errors, inspect elements, and monitor network requests.

Log Statements: Add `print` statements in your Python code to debug and track the flow of execution.

By following these steps in detail, you should be able to set up, run, and interact with your Flask web application. If you encounter any issues, ensure all dependencies are installed correctly and that the project structure is followed precisely.