AMRITA VISHWA VIDYAPEETHAM

DEPT. OF CEN

AMRITA SCHOOL OF AI

B. TECH CSE-AI

SEMESTER-5

21AIE431

APPLIED CRYPTOGRAPHY

END SEMESTER PROJECT

Report

TEAM MEMEBERS:

S.NO	NAME	ROLL.NO
1.	MARREDDY MOHIT SASANK REDDY	CB.EN.U4AIE21031
2.	YARRAM SRI SATHWIK REDDY	CB.EN.U4AIE21077
3.	P.SAI VIJAY KUMAR	CB.EN.U4AIE21040
4.	B. HARISH BALAJI	CB.EN.U4AIE21007

ABSTRACT

In developing a secure client-server model, we implement the Advanced Encryption Standard (AES) and a hashing algorithm from scratch. The AES implementation includes key expansion, key-mixing operations, and secure modes like Cipher Block Chaining (CBC). For hashing, we choose SHA-256 and ensure its resistant to common cryptographic attacks. The communication protocol between the client and server is designed for secure data exchange, utilizing TLS/SSL for additional security. Key management is a crucial aspect, involving a secure key exchange mechanism. Robust error handling, logging, thorough testing, compliance with standards, and comprehensive documentation are integral to this intricate process, demanding expertise in cryptography and a cautious approach.

Proposed Methodology

- ✓ **Web Framework Selection:**
  Flask, a lightweight and extensible web framework, is chosen for its simplicity and flexibility in handling HTTP requests. It facilitates the creation of web applications with minimal boilerplate code.

- ✓ **Password Security Measures:**
  The master password is hashed using the SHA-256 algorithm, a widely accepted and secure hashing function. Hashing provides a one-way transformation, enhancing the security of stored passwords.

- ✓ **Key Derivation with PBKDF2:**
  The PBKDF2 key derivation function is applied to the hashed master password for generating an encryption key. This process introduces computational complexity, making it resistant to brute-force attacks and enhancing key security.

- ✓ **AES Encryption:**
  The Advanced Encryption Standard (AES) is employed for encrypting passwords. AES is a symmetric encryption algorithm widely recognized for its security and efficiency.

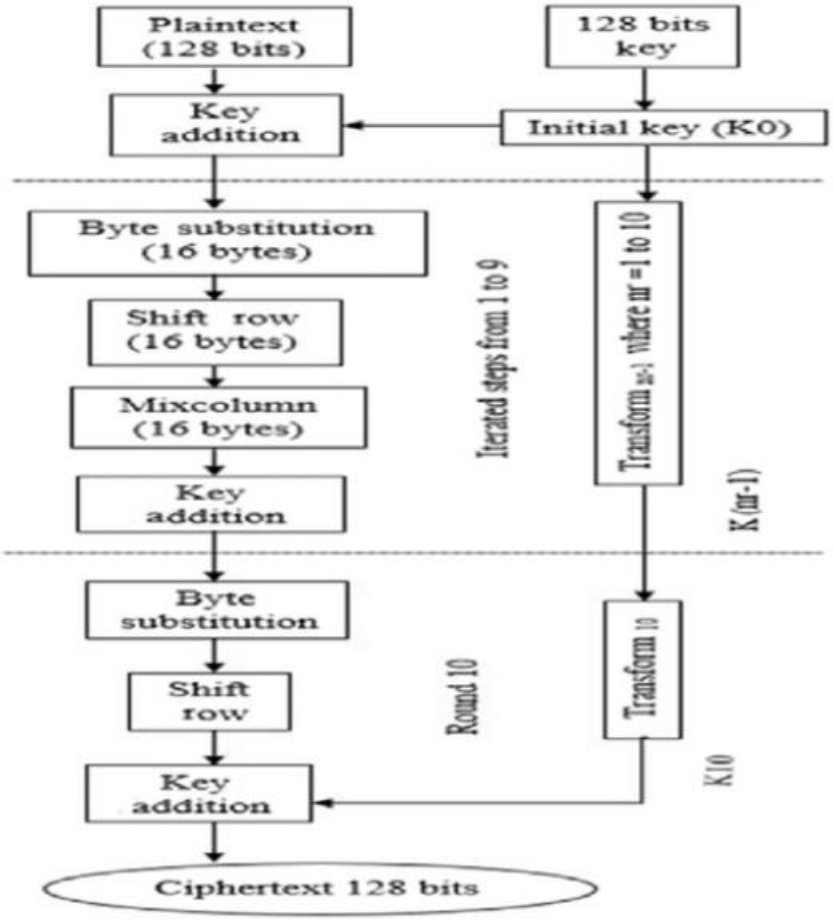
- ✓ **Secure Password Storage:**
  Encrypted passwords are securely stored, and only the hashed master password is stored for verification purposes. This adds an additional layer of security, as even if the stored data is compromised, it would be challenging to retrieve the original master password.

- ✓ **Web API Endpoints for Operations:**
  The application exposes two main endpoints - one for storing passwords securely and another for retrieving passwords. These endpoints follow best practices for handling JSON data and provide a simple yet effective interface for password management.

- ✓ **Error Handling and User Feedback:**
  The code includes error handling mechanisms to provide meaningful feedback to users, ensuring a smooth an user-friendly experience. This is essential for effective communication in case of incorrect inputs or potential issues.

AES Implementation:

AES Encryption:

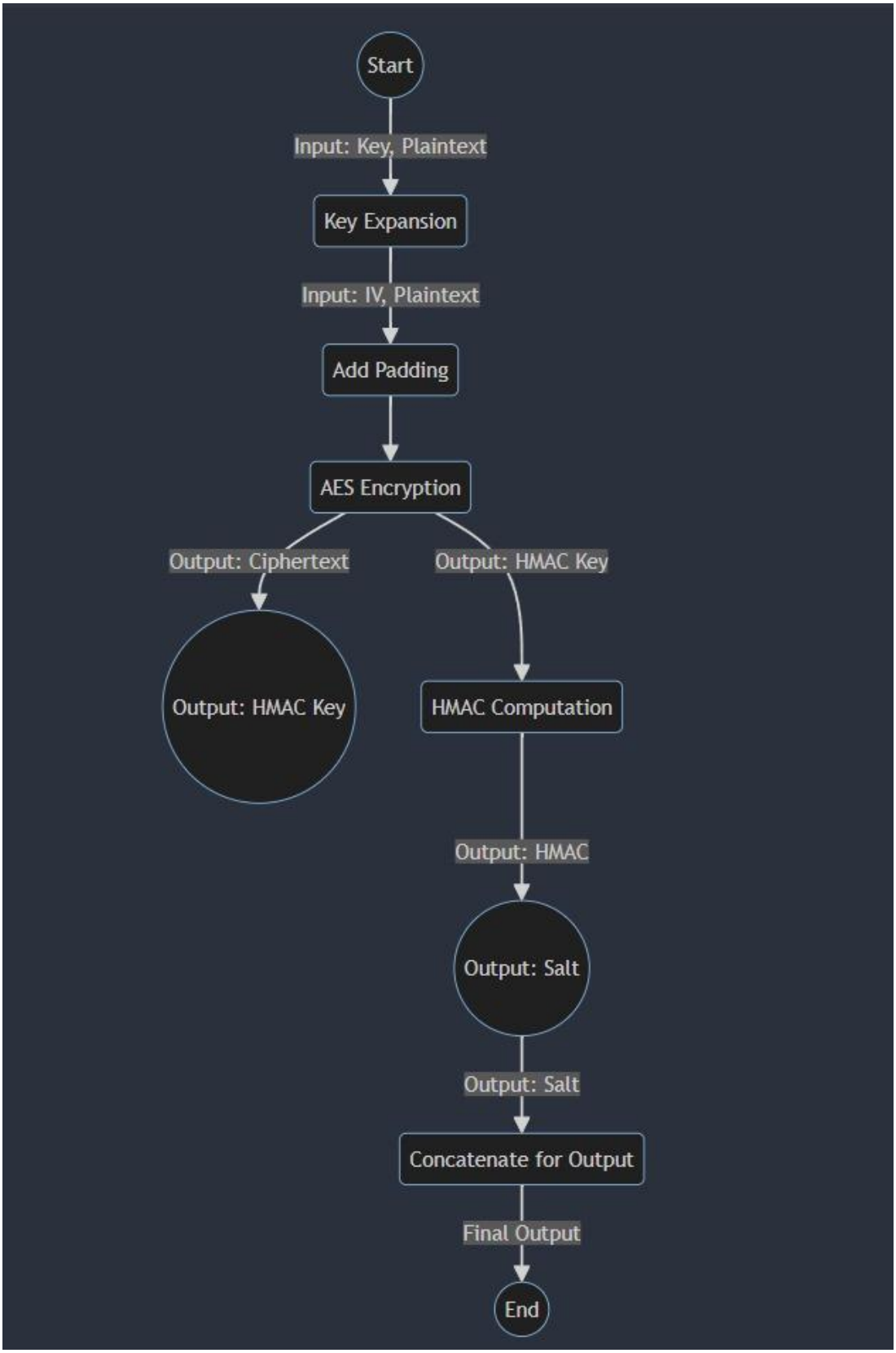


The flowchart shows the steps involved in encrypting a block of data (plaintext) using the AES algorithm. The first step is to add the plaintext to the initial key (K0). This is followed by nine rounds of transformations, each of which consists of four sub steps:

- Byte substitution: This step uses a substitution table (S-box) to replace each byte of the data with a different byte. The S-box is designed to be very non-linear, which makes it difficult to reverse the substitution process.
- Shift rows: This step shifts the bytes in each row of the data to the left by a different number of positions.
- Mix columns: This step mixes the bytes in each column of the data using a matrix multiplication operation.
- Key addition: This step adds another subkey (derived from the initial key) to the data.

After the nine rounds of transformations, a final round is performed that consists only of byte substitution and key addition. The output of the final round is the ciphertext, which is the encrypted form of the plaintext.

The flowchart also shows that there are different transformations used for different rounds of the AES algorithm. This is done to make the algorithm more resistant to attack.

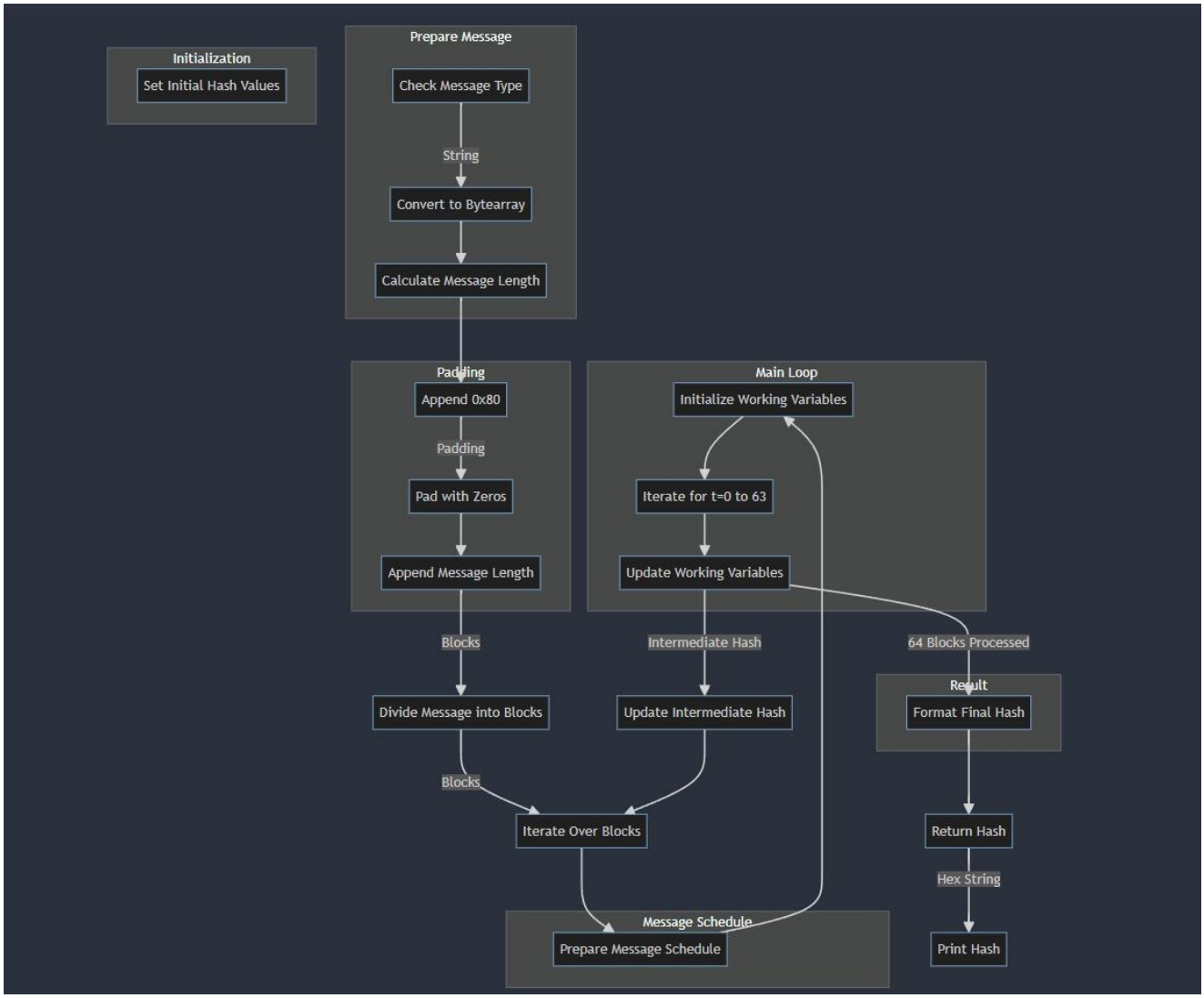


Explanation:

1. **Key Expansion (B):**
    - Input: The process starts with the user providing a key and plaintext.
    - Operation: The key expansion process expands the provided key into round keys for AES encryption.
    - Output: The expanded key is used in the subsequent AES encryption process.
2. **Add Padding (C):**
    - Input: The expanded key and plaintext.
    - Operation: Padding is added to the plaintext to ensure it's a multiple of the block size (16 bytes) for AES encryption.
    - Output: Padded plaintext ready for encryption.
3. **AES Encryption (D):**
    - Input: Padded plaintext and IV (Initialization Vector).
    - Operation: AES encryption is performed on the padded plaintext using the expanded key and IV. The encryption is performed in blocks.
    - Output: Ciphertext after AES encryption.
4. **HMAC Computation (F):**
    - Input: HMAC Key, Salt, and Ciphertext.
    - Operation: HMAC (Hash-based Message Authentication Code) is computed using the HMAC key, salt, and the ciphertext obtained from AES encryption.
    - Output: Computed HMAC.
5. **Concatenate for Output (H):**
    - Input: Computed HMAC, Salt, and Ciphertext.
    - Operation: The HMAC, salt, and ciphertext are concatenated to form the final output.
    - Output: Final output ready to be presented or transmitted.
6. **End (I):**
    - The process concludes, and the final output is the result of the encryption process.

In summary, the flowchart represents the steps involved in encrypting plaintext using AES-128 with CBC (Cipher Block Chaining) mode, HMAC for integrity verification, and key stretching using PBKDF2. The flow highlights key expansion, padding, AES encryption, HMAC computation, and the final output generation.

SHA 256:



Explanation:

1. **Prepare Message:**
   - A: Check the type of the message (e.g., if it's a string).
   - B: Convert the message to a bytearray.
   - C: Calculate the length of the message.
2. **Padding:**
   - D: Append the byte 0x80 to the message.
   - E: Pad the message with zeros until it reaches the appropriate length.

- F: Append the original message length to the padded message.

3. **Initialization:**
    - H: Set the initial hash values. In the case of SHA-256, these are specific constant values.
4. **Divide Message into Blocks:**
    - G: Divide the padded message into blocks, typically 512-bit blocks.
5. **Message Schedule:**
    - J: Prepare the message schedule, which involves breaking each block into 32-bit words and creating a schedule of 64 words.
6. **Main Loop:**
    - K: Initialize working variables, such as hash values.
    - L: Iterate over 64 rounds (t=0 to 63) within the main loop.
    - M: Update working variables based on the message schedule and constants.
7. **Intermediate Hash:**
    - N: Update the intermediate hash values after processing each block.
8. **Result:**
    - O: Format the final hash value.
    - P: Return the hash value.
9. **Print Hash:**
    - Q: Print the hash value as a hex string.

In summary, the flowchart describes the step-by-step process of computing a SHA-256 hash for a given message. It includes the preparation of the message, padding, initialization of hash values, processing of message blocks in a loop, and the generation of the final hash result. Each block of the message goes through a series of operations in the main loop, and the hash is continuously updated until all blocks are processed. The final hash is then formatted and returned.

Server.py:

Explanation:

This code implements a simple password manager web application using Flask and cryptography libraries. Here's a brief explanation of the code:

1. **Flask Setup:**

- The code defines a Flask web application.
- It includes routes for the homepage ('/'), storing passwords ('/store_password'), and retrieving passwords ('/retrieve_password').
- A secret key (SECRET_KEY) is defined, and an empty dictionary (encrypted_data) is used to store encrypted password data.

2. Password Hashing and Encryption:

- hash_master_password(master_password): Hashes the master password using SHA-256.
- encrypt(key, password): Encrypts a password using AES encryption with a derived key.

3. Routes:

- /: Renders the homepage (index.html).
- /store_password (POST): Stores an encrypted password associated with a domain.
- /retrieve_password (POST): Retrieves a decrypted password for a given domain.

4. Storing Passwords (/store_password route):

- Receives JSON data with domain, password, and masterPassword fields.
- Validates the presence and non-emptiness of required fields.
- Hashes the master password, derives a key using PBKDF2, and encrypts the password using AES.
- Stores the encrypted password, hashed master password, and domain in the encrypted_data dictionary.

5. Retrieving Passwords (/retrieve_password route):

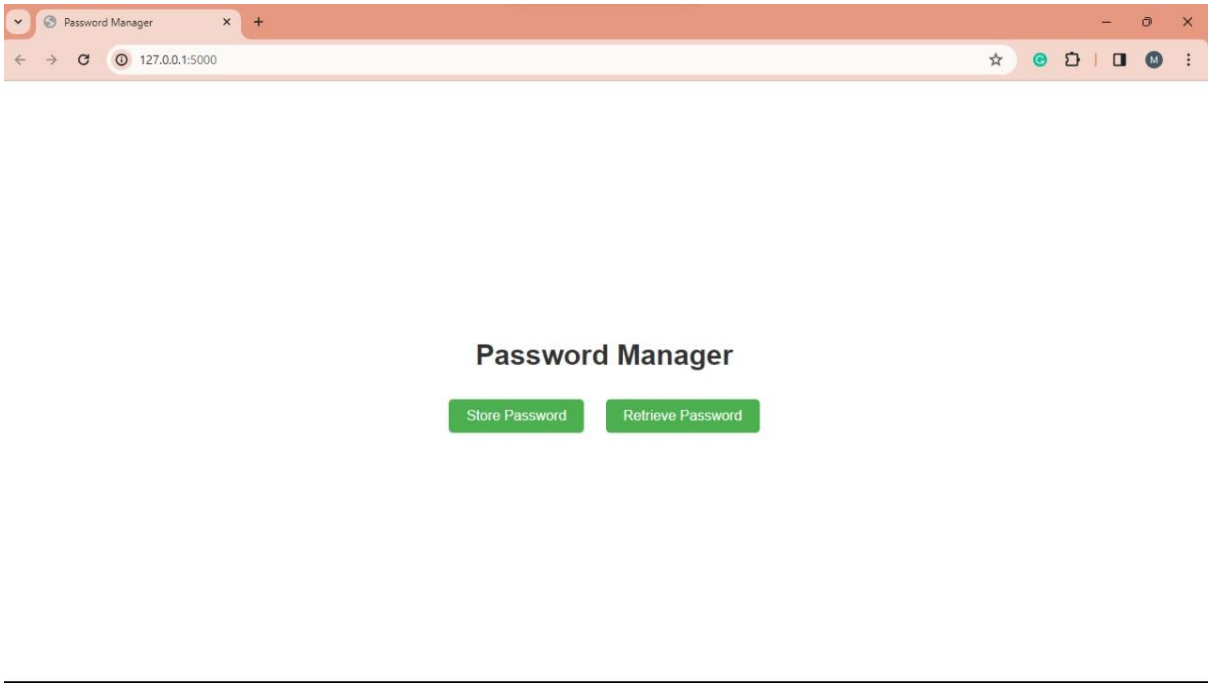
- Receives JSON data with domain and masterPassword fields.
- Validates the presence and non-emptiness of required fields.
- Retrieves the encrypted item for the given domain from encrypted_data.
- Compares the provided master password's hash with the stored hash.
- Derives the key using PBKDF2 and decrypts the stored password using AES.
- Returns the decrypted password.

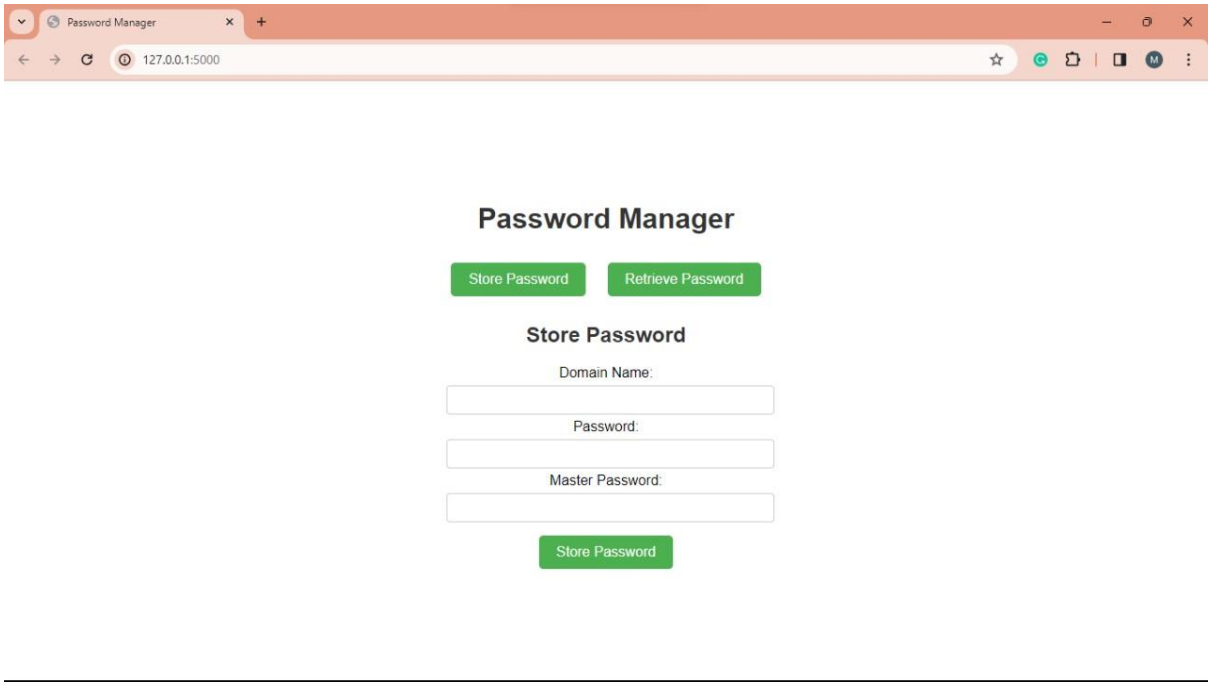
6. Running the Application:

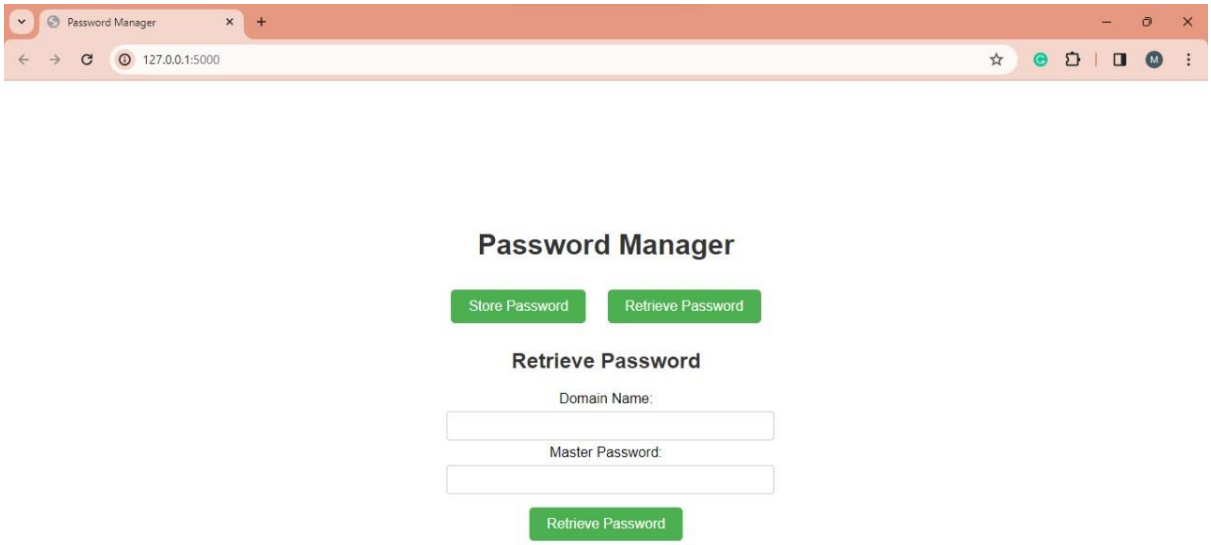
- The application runs on 0.0.0.0:5000 in debug mode.

This password manager uses Flask as a web framework, SHA-256 for master password hashing, PBKDF2 for key derivation, and AES for password encryption. It's a simplified example and might not cover all security considerations for a production environment.

Implementation:







Conclusion:

The implemented password management system using Flask, SHA-256 hashing, PBKDF2 key derivation, and AES encryption provides a robust and secure solution for storing and retrieving passwords. The web application offers a user-friendly interface while prioritizing data confidentiality and master password protection. The choice of cryptographic algorithms and secure coding practices enhances the overall security posture.

Future Work:

In advancing this password management system, several key areas warrant exploration. First, the integration of biometric authentication can enhance user convenience and security. Additionally, the implementation of a robust intrusion detection system and anomaly detection algorithms will fortify the application against evolving cyber threats. Enhancing scalability through cloud-based storage solutions and exploring blockchain for decentralized security measures are avenues for consideration. To improve user experience, incorporating a comprehensive password strength evaluation mechanism and refining the user interface for mobile responsiveness will be beneficial. Lastly, continuous security audits and adherence to emerging cryptographic standards will ensure the system remains resilient in the face of emerging threats, contributing to a sustainable and secure password management solution.

References:

https://link.springer.com/book/10.1007/978-3-662-04722-4

https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf

THE END