2024 ISM Challenge

Instructions for the Students

Each step must be implemented in the sequence provided, with each step building upon the previous. Each completed task should be tested independently.

Development: Implement the entire solution in a single .java file using static methods

Upload a single .zip file containing

- your java solution
- all the files your solution has generated

Evaluation Criteria

- **Correctness**: Implementation of each algorithm and encryption/decryption process works as intended.
- **Sequential Dependence**: Each task builds upon the previous one correctly.
- **All or nothing**: There are no partial evaluations. Each step is evaluated based on the output results. If the output results are ok you get the points for each step.
- **Fair game**: The code is not copied from others and not generated with tools.

Story: Secure Communications for SafeCorp

Scenario: You have been hired as a cybersecurity engineer by **SafeCorp**, a fictional tech company, to build a secure communication protocol for their internal messaging app. The protocol must ensure that messages sent between employees are secure, verified, and protected from tampering. The requirements for this system are provided in a series of tasks, with each step depending on the successful completion of the previous ones.

Task Breakdown

1. **Message Integrity**: **(10 pts)**
    - SafeCorp needs you to create a mechanism that will ensure message integrity. Implement a solution using a **Message Digest** algorithm (e.g., SHA-256) to generate a hash of the message before it's sent. When the message is received, the hash should be verified to ensure it hasn't been altered in transit.
    - **Objective**: Verify file integrity before any operations and find your assigned file.

- **Input**: The files from the safecorp_random_messages.zip archive. Extract the files and put them in a folder, *safecorp_random_messages*, in the root of your java project
- **Task**:
    1. Calculate a **Message Digest** (e.g., SHA-256) for all the files in *safecorp_random_messages* folder and check it gains the value given to you in the Excel file.
    2. If you find your file, print its name on the screen and return its name for future use. For the rest of the challenge, we will call this file **message.txt** (is one of the message_X_Y.txt files)
- **Output**:
    1. The name of your assigned text file.
-

2. **Verification of Message Authenticity (10 pts)**
    - SafeCorp wants to confirm that messages come from verified users. Implement an **HMAC (Hash-based Message Authentication Code)** using SHA-256 to add an authentication tag to each message. This HMAC should be derived from a secret key shared between the sender and receiver (given in the MS Excel file).
    - **Objective**: Ensure authenticity of the received message.
    - **Input**: your message txt file, **message.txt**.
    - **Task**:
        1. Generate an **HMAC (e.g., HMAC-SHA256)** for your **message.txt** using a shared secret key (given in the MS Excel file).
        2. Save the HMAC as a hex string in the ***hmac.txt*** file.
    - **Output**:
        1. **hmac.txt**: Contains the HMAC for your allocated message text file.

3. **Password-Based Key Derivation**: **(10 pts)**
    - To ensure all encryption keys are strong and unpredictable, SafeCorp requires that all keys are derived from passwords. Use **PBKDF2** with salt to derive a symmetric encryption key from a given password. Store the salt securely in the message so it can be used to verify or decrypt the message on the receiving end.
    - **Objective**: Securely derive encryption keys from a password.

- **Input**: User given password. No of iterations for the PBKDF2
- **Task**:
    1. Use **PBKDF2** with a salt (you can choose) to derive a symmetric encryption key from the password.
    2. The number of iterations are given in the MS Excel file
    3. Save the salt as **salt.txt**.
- **Output**:
    1. **salt.txt**: Contains the salt used in the PBKDF2 key derivation.
    2. A byte array that can have any required size

4. **Confidentiality with Encryption: (20pts)**
    - Messages are being sent in plaintext, which could be intercepted. Implement **AES encryption in CBC mode** to secure the messages during transit. Use a generated initialization vector (IV) for each encryption (check the MS Excel file for the IV requirements). The generated IV should be sent along with the encrypted message in a separate file. For padding use PKCS5.
    - Ensure that the encrypted data is **Base64 encoded** for safe transfer over text-based protocols.
    - **Objective**: Secure the file content for transmission.
    - **Input**: **message.txt**, a key derived from the previous master shared password with the PBKDF2 implementation
    - **Task**:
        1. Encrypt **message**.txt using **AES in CBC mode** with a generated initialization vector (IV). Check MS Excel for instructions on IV.
        2. Base64 encode both the encrypted content and the IV.
        3. Save the encrypted file as ***encrypted.txt*** and the IV in ***iv.txt***.
    - **Output**:
        1. **encrypted.txt**: Base64-encoded, AES-CBC-encrypted version of message.txt.
        2. **iv.txt**: Base64-encoded IV used for encryption.

5. **File Encryption Requirement**: **(20pts)**
    - Messages can sometimes contain attached files. Develop functionality to encrypt a file using **3DES in ECB mode**. Ensure that the file's contents are properly encoded for transmission (using Base64) and can be decrypted successfully on the receiving end.
    - **Objective**: Secure the file for archival using an alternate encryption method.
    - **Input**: message.txt from Step 1, another key derive from PBKDF2.
    - **Task**:
        1. Encrypt **message.txt** using **3DES in ECB mode** and store the result as binary.
        2. Use the previous PBKDF2 function to generate the required key from the master password
        3. Save this version as ***archived.sec***.
    - **Output**:
        1. **archived.sec**: binary, 3DES-encrypted version of message.txt.
6. **Bitwise Operations for Security Enhancements**: **(20pts)**
    - Implement a simple bitwise operation that, for each byte in the encrypted data, toggles specific bits (e.g., shifts bits left by two positions). This will serve as an extra layer of obfuscation. Include this step in both encryption and decryption.
    - **Objective: Add an extra layer of reversible obfuscation to the encrypted file before** transmission.
    - **Input**: encrypted.txt from Step 4 (the AES-CBC-encrypted, Base64-encoded version of the original file).
    - **Task**:
        1. Perform a cyclic (circular) left shift by 2 bits on each byte of the file contents. This process rotates the bits in each byte so that the leftmost bits wrap around to the right side, preserving all information and making the operation reversible.
        2. Save the obfuscated result as **obfuscated.txt**.
    - **Output**:
        1. **obfuscated.txt**: A file where each byte has been cyclically left-shifted by 2 bits, creating an obfuscated, reversible transformation of encrypted.txt.
    - **Reversibility**:

1. To reverse this step, apply a cyclic right shift by 2 bits on each byte of **obfuscated.txt**. This will restore the file to its original **encrypted.txt** state, allowing decryption to be performed in later steps.
