[Date]

Sharnbasava University

department of aiml foet(w),suk

**PROTECTING USER PASSWORD KEYS AT REST (ON THE DISK)**

**ABSTRACT**

In today's digital age, the security of sensitive information, particularly user passwords, is of paramount importance. Traditional methods of storing password keys in plaintext on disk pose significant security risks, making them vulnerable to unauthorized access and cyber-attacks. This project addresses these concerns by implementing a robust encryption mechanism using the Advanced Encryption Standard (AES) with a 256-bit key (AES-256). The solution ensures that user password keys are encrypted and securely stored at rest, thereby significantly enhancing data protection and privacy. The project involves generating a random File Encryption Key (FEK) for encrypting files and a Key Encryption Key

TABLE OF CONTENTS:

|  |  |
| --- | --- |
| Sl no. | Contents |
| 1. | Introduction |
| 2. | Objectives |
| 3. | What is Encryption? |
| 4. | What is Decryption? |
| 5. | Flow chart |
| 6. | Implementation |
| 7. | Source code |
| 8. | Output |
| 9. | Requirements |
| 10. | References |
| 11. | Conclusion |

**1.INTRODUCTION:**

In the digital era, the protection of sensitive information such as user passwords has become a critical concern. With the increasing frequency and sophistication of cyber-attacks, ensuring the security of user data stored on disk is essential to maintain privacy and trust. Traditional methods of storing password keys in plaintext are highly vulnerable to unauthorized access, leading to potential data breaches and significant security risks.

To address these challenges, this project focuses on implementing a robust encryption mechanism to protect user password keys at rest using the Advanced Encryption Standard (AES) with a 256-bit key (AES-256). AES-256 is a widely recognized and trusted encryption standard that provides strong security due to its large key size and resistance to cryptographic attacks.

The primary objective of this project is to develop a secure method for encrypting user password keys before storing them on disk. This involves generating a random File Encryption Key (FEK) for encrypting files and a Key Encryption Key (KEK) derived from a user passphrase to encrypt the FEK. This two-layer encryption approach ensures that even if the encrypted file is compromised, the FEK remains secure and inaccessible without the correct passphrase.

By leveraging AES-256 encryption, this project aims to significantly enhance the security of stored user password keys, providing a practical solution to mitigate the risks associated with unauthorized access and data breaches. This document outlines the methodology, implementation, and results of the project, demonstrating the effectiveness of the proposed encryption approach in safeguard (KEK) derived from a user passphrase to further encrypt the FEK.

**2.** **Objectives**

The primary objectives of this project are to enhance the security of user password keys stored on disk by implementing a robust encryption mechanism. Specifically, the project aims to:

**1. Implement AES-256 Encryption**:

- Utilize the Advanced Encryption Standard (AES) with a 256-bit key to ensure a high level of security for encrypting user password keys.

**2. Generate Secure File Encryption Keys (FEKs):**

- Develop a method for generating random File Encryption Keys (FEKs) that will be used to encrypt files, ensuring the encryption keys themselves are not stored in plaintext.

**3. Derive Key Encryption Keys (KEKs) from User Passphrases:**

- Implement a process to derive Key Encryption Keys (KEKs) from user-provided passphrases using a secure key derivation function (PBKDF2) with a unique salt, ensuring that the KEK is unique and secure.

**4. Encrypt and Secure FEKs:**

- Encrypt the FEKs using the derived KEKs, ensuring that even if the encrypted files are accessed, the encryption keys remain protected and can only be decrypted with the correct user passphrase.

**5. Implement and Test the Solution:**

- Develop and test the encryption and decryption functions to ensure they work correctly and securely, providing reliable protection for user password keys.

By achieving these objectives, the project aims to deliver a secure, reliable, and user-friendly solution for protecting user password keys at rest, thereby enhancing overall data security and privacy

**3.WHAT IS ENCRYPTION?**

**Encryption** is the process of converting plain, readable data (plaintext) into an encoded format (ciphertext) that can only be read or decrypted by someone who possesses the appropriate decryption key. Encryption is used to ensure the confidentiality of data, protecting it from unauthorized access, even if the data is intercepted or accessed by unauthorized individuals.

**The Encryption Process**

The encryption process involves several key steps to ensure data is securely converted from plaintext to ciphertext. Here's a detailed explanation of the encryption process:

1. **Plaintext**:
   * The original, readable data that needs to be protected.
2. **Encryption Algorithm**:
   * A set of mathematical rules or procedures used to transform the plaintext into ciphertext. Common encryption algorithms include AES (Advanced
   * Encryption Standard), DES (Data Encryption Standard), and RSA (Rivest-Shamir-Adleman).
3. **Encryption Key**:
   * A piece of information used by the encryption algorithm to perform the transformation. The key must be kept secret because it is required to decrypt the ciphertext back into plaintext. The security of the encryption largely depends on the secrecy and complexity of the key.
4. **Ciphertext**:
   * The encrypted data that is the result of the encryption process. Ciphertext is not readable without the appropriate decryption key.

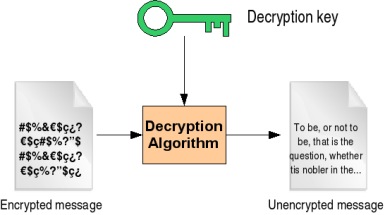


Figure Decryption process

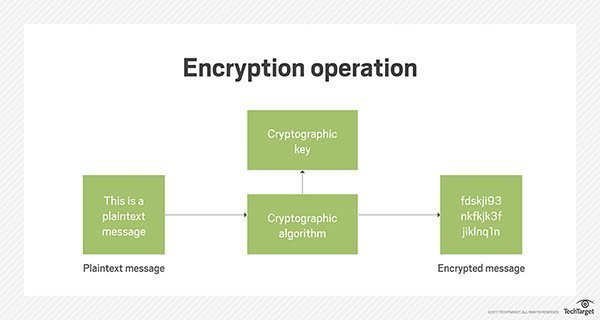


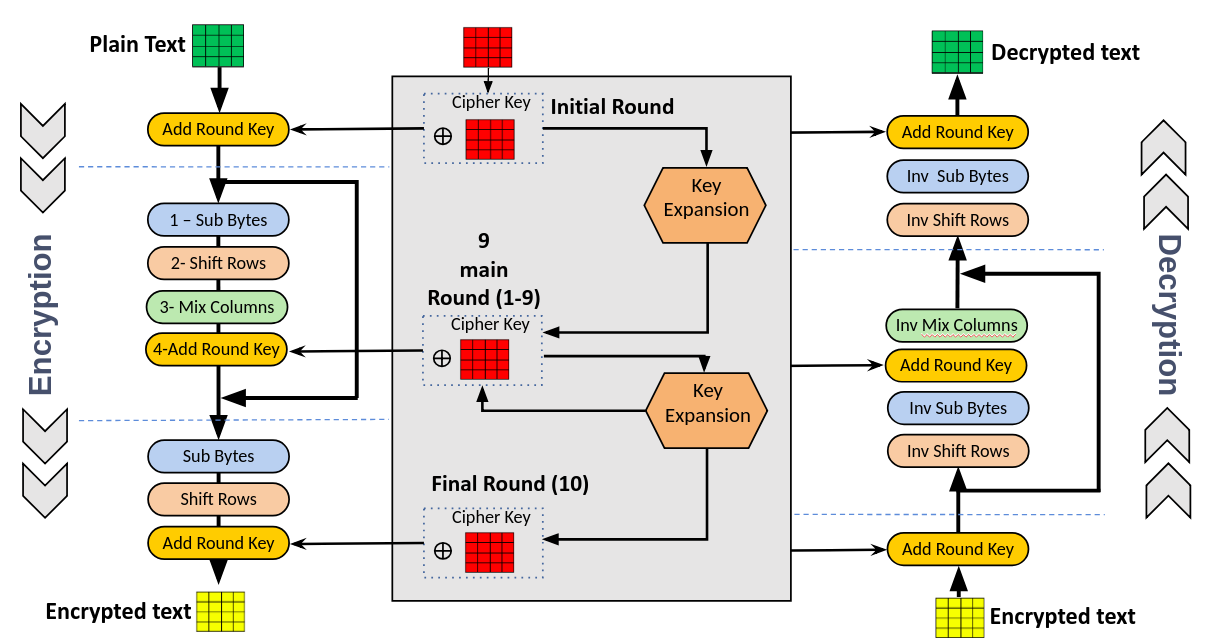
Figure Encryption process

**4.WHAT IS DECRYPTION?**

The decryption process is the reverse of encryption. It involves converting the ciphertext back to plaintext using the appropriate decryption key.

1. **Ciphertext Input**:
   * The encrypted data that needs to be decrypted.
2. **Retrieve the Nonce and Tag**:
   * Extract the nonce and tag used during encryption.
3. **Decrypt the Ciphertext**:
   * Use the same encryption algorithm and key to decrypt the ciphertext.

**5.FLOW CHART:(AES ALGORITHM)**



**6.IMPLEMENTATION**:

1. **Import Libraries**: We import necessary libraries from PyCryptodome.
2. **Constants**: We define constants for sizes of salt, key, nonce, and tag, as well as the file name to save encrypted data.
3. **Generate Key**: The generate\_key function creates a 256-bit key using the PBKDF2 function with a given password and salt.
4. **Encrypt Data**: The encrypt\_data function:
   * Generates a random salt and nonce.
   * Derives a key from the password and salt.
   * Encrypts the plaintext using AES-256 in EAX mode.
   * Saves the salt, nonce, tag, and ciphertext to a file.
5. **Decrypt Data**: The decrypt\_data function:
   * Reads the salt, nonce, tag, and ciphertext from the file.
   * Derives the same key from the password and salt.
   * Decrypts the ciphertext and verifies the integrity using the tag.
   * Make sure to use a strong and secure password in real applications to ensure the security of the encrypted data

**7.SOURCE CODE:**

import os

fromcryptography.hazmat.primitives.kdf.pbkdf2import PBKDF2HMAC

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.backends import default\_backend

from cryptography.fernet import Fernet

import base64

SALT\_SIZE = 16

ITERATIONS = 100000

KEY\_SIZE = 32

def generate\_password\_hash(password: str, salt: bytes) -> bytes:

kdf = PBKDF2HMAC(

algorithm=hashes.SHA256(),

length=KEY\_SIZE,

salt=salt,

iterations=ITERATIONS,

backend=default\_backend()

)

return kdf.derive(password.encode())

# Encrypt data using Fernet symmetric encryption

def encrypt\_data(data: bytes, encryption\_key: bytes) -> bytes:

fernet = Fernet(encryption\_key)

return fernet.encrypt(data)

# Decrypt data using Fernet symmetric encryption

def decrypt\_data(encrypted\_data: bytes, encryption\_key: bytes) -> bytes:

fernet = Fernet(encryption\_key)

return fernet.decrypt(encrypted\_data)

# Encrypt a file

def encrypt\_file(file\_path: str, encryption\_key: bytes):

try:

with open(file\_path, 'rb') as file:

file\_data = file.read()

except FileNotFoundError as e:

print(f"Error: File not found - {file\_path}")

print(e)

return

encrypted\_data = encrypt\_data(file\_data, encryption\_key)

with open(file\_path + ".enc", 'wb') as file:

file.write(encrypted\_data)

print (f"File encrypted and saved as {file\_path}.enc")

# Decrypt a file

def decrypt\_file(file\_path: str, encryption\_key: bytes):

try:

with open(file\_path, 'rb') as file:

encrypted\_data = file.read()

except FileNotFoundError as e:

print(f"Error: File not found - {file\_path}")

print(e)

return

decrypted\_data=decrypt\_data(encrypted\_data, encryption\_key)

with open(file\_path[:-4], 'wb') as file:

file.write(decrypted\_data)

print(f"File decrypted and saved as {file\_path[:-4]}")

# Main encryption workflow

def main():

file\_path = input("Enter the file path to encrypt: ")

password = input("Enter a password: ")

# Generate random encryption key for file encryption

file\_encryption\_key = Fernet.generate\_key()

# Encrypt the file

encrypt\_file(file\_path, file\_encryption\_key)

# Generate salt and derive key from password

salt = os.urandom(SALT\_SIZE)

derived\_key = generate\_password\_hash(password, salt)

# Encrypt the file encryption key using the derived key

encrypted\_file\_key=encrypt\_data(file\_encryption\_key, base64.urlsafe\_b64encode(derived\_key))

# Store the encrypted file key and salt

with open(file\_path + ".key", 'wb') as key\_file:

key\_file.write(salt + encrypted\_file\_key)

print (f"File encrypted and key stored in {file\_path}.key")

password = input("Enter the password to decrypt: ")

with open(file\_path + ".key", 'rb') as key\_file:

salt = key\_file.read(SALT\_SIZE)

encrypted\_file\_key = key\_file.read()[SALT\_SIZE:] # Remove salt from encrypted file key

derived\_key = generate\_password\_hash(password, salt)

decrypted\_file\_key=decrypt\_data(encrypted\_file\_key, base64.urlsafe\_b64encode(derived\_key))

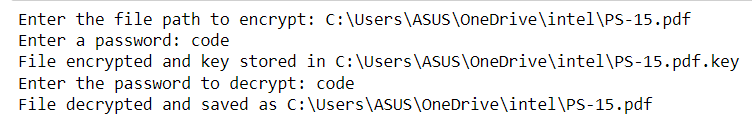
decrypt\_file(file\_path + ".enc", decrypted\_file\_key)

print(f"File decrypted and saved as {file\_path}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**8.OUTPUT:**



**9.REQUIREMENTS:**

**9.1. Hardware Requirements:**

1. Processor: Intel i5 or Intel i7

2. At least 500 GB

- SSD: Recommended Memory (RAM): 4 GB (minimum), 8 GB

3. Storage:

- Hard Drive:

for better performance

4. Graphics: Integrated Graphics (minimum), Dedicated GPU (optional, for enhanced performance)

5. Network: Internet connection for downloading dependencies and resources

6. Peripherals: Standard keyboard, mouse, monitor, and optionally, external storage for backups

**9.2. Software Requirements:**

1. Operating System:

- Windows 10 or 11

2. Python:

- Version 3.12.4

3. Libraries:

- PyCryptodome (for cryptographic functions)

- Installation command: `pip install pycryptodome`

4. IDE/Code Editor:

- Visual Studio Code, PyCharm or any preferred text editor

5. Additional Tools:

- Github

- Virtual Environment

6. Documentation Tools:

- Microsoft Word

- Microsoft powerpoint

**9.3. Installation and Setup Instructions:**

1. Python Installation

- Download and install Python from the [official Python website] (<https://www.python.org/downloads/windows/>) . Make sure to check the box "Add Python to PATH" during installation.

2. Setting up a Virtual Environment

It is recommended to use a virtual environment to manage dependencies:

# Install virtualenv if not already installed

pip install virtualenv

# Create a virtual environment

virtualenv venv

# Activate the virtual environment

venv\Scripts\activate

3. Installing Required Libraries

With the virtual environment activated, install the required libraries:

pip install pycryptodome

4. Setting up an IDE

- Visual Studio Code: Install from the [official website] (<https://code.visualstudio.com/>) . Add the Python extension for enhanced support.

- PyCharm: Download and install from the [official website**] (**[**https://www.jetbrains.com/pycharm/**](https://www.jetbrains.com/pycharm/)**).**

**10. REFERENCES:**

* [**https://www.w3schools.com/**](https://www.w3schools.com/)
* [**https://github.com/**](https://github.com/)
* [**https://www.geeksforgeeks.org/**](https://www.geeksforgeeks.org/)
* [**https://pypi.org/project/cryptography**](https://pypi.org/project/cryptography)

**11.CONCLUSION:**

The project on protecting user password keys at rest (on the disk) using AES-256 encryption effectively demonstrates the importance of securing sensitive data. Through the implementation of advanced encryption standards and robust cryptographic techniques, this project ensures that user password keys are protected from unauthorized access and potential breaches.

This project underscores the significance of adopting strong encryption methods to protect sensitive information in an increasingly digital world.

In conclusion, the successful implementation of AES-256 encryption for protecting user passwordkeys at rest demonstrates a robust approach to data security. This project serves as a valuable reference for developers and security professionals seeking to implement similar encryption solutions to safeguard sensitive information.