Writeup for The Obsidian Cipher

Challenge Name: The Obsidian Cipher

Category: Cryptography

Challenge Overview

Deep within the mysterious **Obsidian Temple**, an ancient encrypted message lies buried behind multiple cryptographic layers. Known as the **Obsidian Cipher**, this challenge demands persistence, cryptographic knowledge, and problem-solving skills to reveal the hidden treasure.

As the daring cryptographer, your mission is clear:

1. Break through the **XOR encryption layer** using the provided key.
2. Decrypt the **AES-encrypted layer** using the password.
3. Extract the flag hidden within the final plaintext.

Step 1: Key Details from the Challenge

1. **File Provided:** final_encrypted.bin (encrypted binary data).
2. **Hints:**
   - The XOR key is 42.
   - Password for decryption is super_secret_password.
3. **Output Format:** The flag must follow the format ctf{...}.

Step 2: Approach to Solve the Challenge

The challenge involves two primary decryption steps:

1. **XOR Decryption:**

The first encryption layer is an XOR operation with a static key (42).

2. **Password-Based Decryption:**

After XOR decryption, the resulting data is AES-encrypted. This step involves:

- Deriving a cryptographic key from the password (super_secret_password) using **PBKDF2**.

    - Decrypting the AES-encrypted data in **CBC mode**, where the initialization vector (IV) is stored in the first 16 bytes of the file.

Step 3: Solution Code

Here’s the complete Python code used to decrypt the file and extract the flag:

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.hashes import SHA256
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import os

# XOR Decryption Function
def xor_decrypt(data, key):
    return bytes([byte ^ key for byte in data])

# Password-Based Decryption Function
def decrypt_with_password(data, password):
    # Derive a 16-byte key using PBKDF2
    salt = b"this_is_a_salt"
    kdf = PBKDF2HMAC(
        algorithm=SHA256(),
        length=16,
        salt=salt,
        iterations=100000,
    )
    key = kdf.derive(password.encode())

    # Decrypt using AES-CBC
    iv = data[:16]  # The first 16 bytes are the IV
    encrypted_message = data[16:]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(encrypted_message) + decryptor.finalize()
    return plaintext

# Main Decryption Logic
def main():
    # Step 1: Read the encrypted file
    with open("final_encrypted.bin", "rb") as file:
        encrypted_data = file.read()

    # Step 2: XOR decryption
    xor_key = 42
    intermediate_data = xor_decrypt(encrypted_data, xor_key)

    # Step 3: Password-based decryption
    password = "super_secret_password"
    try:
        final_plaintext = decrypt_with_password(intermediate_data, password)
        flag = final_plaintext.decode().strip()  # Decode and clean the plaintext
        print("Decrypted flag:", flag)
    except Exception as e:
        print("Error during decryption:", str(e))

# Execute the decryption process
if __name__ == "__main__":
    main()
```

Step 4: Decryption Process

1. XOR Decryption:

- Open and read the binary file (final_encrypted.bin).
- Perform an XOR operation on every byte of the file using the key 42.
- This removes the first encryption layer, leaving data for the next decryption step.

2. Password-Based Decryption:

- Extract the IV (first 16 bytes of the XOR-decrypted data).
- Derive a cryptographic key using the password super_secret_password and a fixed salt.
- Decrypt the remaining data using AES in CBC mode with the derived key and IV.

3. Flag Extraction:

- Decode the final plaintext to reveal the flag in a clean format.

Step 5: Decrypted Flag

After executing the script, the decrypted output is:

Decrypted flag: ctf{obsidian_cipher_master}

Key Insights and Lessons Learned

1. **XOR Encryption:**
   - XOR is simple yet effective when combined with additional layers of encryption.
   - However, the challenge demonstrates how its weakness—predictable keys or plaintext hints—makes it easier to crack.
2. **AES in CBC Mode:**
   - AES encryption provides robust security, but it is crucial to manage IVs and key derivation carefully.

- The use of a static salt here simplifies the challenge but highlights how these parameters strengthen encryption.

3. **Practical Cryptographic Knowledge:**
    - This challenge required applying real-world cryptographic tools like PBKDF2, AES, and CBC mode.
    - Understanding these tools is essential for tackling modern cryptographic puzzles.