

```

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
import os
import base64

# Key and IV Generation
def generate_key(password: str, salt: bytes):
    """
    Derives a 256-bit AES key from a password using PBKDF2.
    """
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32, # AES-256 key
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    return kdf.derive(password.encode())

def encrypt_response(response: str, key: bytes, iv: bytes):
    """
    Encrypts the test response using AES encryption.
    """
    # Pad the plaintext to make it a multiple of block size
    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    padded_data = padder.update(response.encode()) + padder.finalize()

    # Encrypt the data
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    encrypted_data = encryptor.update(padded_data) + encryptor.finalize()

    # Encode in Base64 for storage
    return base64.b64encode(encrypted_data).decode()

def decrypt_response(encrypted_response: str, key: bytes, iv: bytes):
    """
    Decrypts the AES-encrypted response.
    """
    # Decode from Base64
    encrypted_data = base64.b64decode(encrypted_response)

    # Decrypt the data
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())
    decryptor = cipher.decryptor()
    padded_data = decryptor.update(encrypted_data) + decryptor.finalize()

    # Remove padding
    unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
    data = unpadder.update(padded_data) + unpadder.finalize()
    return data.decode()

# Example Workflow
if __name__ == "__main__":
    # Test parameters
    password = "securepassword" # User-provided password for key derivation
    salt = os.urandom(16) # Unique salt for key generation
    iv = os.urandom(16) # Initialization vector for AES


    # Generate key
    key = generate_key(password, salt)

    # User response to encrypt
    response = "The capital of France is Paris."

    # Encrypt the response
    encrypted_response = encrypt_response(response, key, iv)
    print(f"Encrypted Response: {encrypted_response}")

    # Decrypt the response
    decrypted_response = decrypt_response(encrypted_response, key, iv)
    print(f"Decrypted Response: {decrypted_response}")

```

 Encrypted Response: +VV2jN6pODs8w46ew6QXjD6NrRthdU8lnIJwcjzNPRY=
 Decrypted Response: The capital of France is Paris.

