PROYECTO CRIPTOGRAFÍA

Alberto Menchen Montero 100495692
Javier Moyano San Bruno 100495884

ÍNDICE

1. **¿Cuál es el propósito de su aplicación?**

El propósito de la aplicación es proporcionar un sistema que simule las funciones de un gestor de contraseñas. Cuenta con un sistema de registro y autenticación seguro para los usuarios mediante el cual pueden almacenar cualquier tipo de contraseña y asegurarse de que va a estar encriptada de buena manera y bajo un tratamiento de sus datos personales seguro y eficaz.
Incluye funcionalidades como el cifrado de datos personales, la gestión de contraseñas y restauración de las mismas mediante envío de PINs al correo vinculado a la cuenta. Como estas un amplio abanico de funciones capaces de hacer intuitiva, sencilla y segura la utilización de la aplicación.También, cuenta con técnicas de cifrado y hashing para asegurar la integridad y confidencialidad.

2. **¿Para qué utiliza la firma digital? ¿Qué algoritmos ha utilizado y por qué? ¿Cómo se gestionan y almacenan las claves y las firmas?**

2.1. ¿Para qué utiliza la firma digital?

La firma digital se usa para garantizar la integridad y autenticidad de los datos. Al firmar digitalmente un documento o mensaje, se asegura que el contenido no ha sido alterado y que proviene de una fuente confiable.

En nuestro proyecto la hemos usado para firmar cada una de las contraseñas que se ingresan en el uso de la aplicación. Se genera una firma digital para cada par de asunto y contraseña al almacenarlas y se verifica la firma al consultar las contraseñas, verificando que la información no ha sido alterada.

2.2. ¿Qué algoritmos ha utilizado y por qué?

El sistema utiliza RSA con PSS (Probabilistic Signature Scheme) para la generación y verificación de firmas digitales, como se puede observar en la función generar_firma_digital del archivo digital_signature.py

Razón para elegir RSA con PSS

Seguridad probada:

RSA es un estándar ampliamente utilizado para criptografía asimétrica y ofrece un alto nivel de seguridad. PSS es un esquema de padding para RSA que proporciona una mejora significativa en términos de seguridad al evitar ataques basados en patrones predecibles.

Esquema probabilístico:

PSS introduce aleatoriedad en el proceso de firma, lo que hace que cada firma sea única incluso si se firman los mismos datos múltiples veces. Esto erradica ataques que podrían aprovechar patrones repetidos en las firmas.

Cumplimiento de estándares modernos:

PSS está recomendado en estándares criptográficos modernos, como FIPS 186-4 y RFC 8017, ya que proporciona una mayor resistencia a ataques teóricos en comparación con esquemas de padding más antiguos como PKCS#1 v1.5.

Compatibilidad:

RSA es ampliamente soportado y puede integrarse fácilmente con otros sistemas o aplicaciones, lo que lo hace ideal para un proyecto que utiliza certificados digitales y claves públicas/privadas.

Integración con SHA-256:

La combinación de RSA con SHA-256 utilizada en este sistema ofrece una resistencia robusta contra colisiones, garantizando que los datos no puedan ser alterados sin invalidar la firma.

2.3. Ejemplo de uso y propósito de digital_signature

Ejemplo de Uso

Generación:

Un usuario guarda un asunto con su contraseña asociada. La aplicación cifra la contraseña y genera una firma digital de los datos asociados, asegurando la autenticidad e integridad de esa información.

Verificación:

Cuando el usuario recupera la contraseña, el sistema descifra los datos y verifica que la firma generada coincide con los datos descifrados. Esto garantiza que la contraseña no fue alterada por terceros.

Resumen de Propósito

generar_firma_digital: Garantiza que los datos estén protegidos contra manipulaciones, proporcionando autenticidad y no repudio.

verificar_firma_digital: Confirma que los datos recibidos no han sido modificados y que provienen del usuario legítimo que posee la clave privada correspondiente.

2.4. ¿Cómo se gestionan y almacenan las claves y las firmas?

Gestión de Claves

Durante el registro o en momentos específicos, cada usuario genera un par de claves RSA: una clave privada y su correspondiente clave pública. La clave privada, de tamaño 2048 bits, se genera mediante la biblioteca cryptography, utilizando un exponente público estándar de 65537. Este nivel de seguridad es suficiente para proteger datos sensibles contra ataques conocidos.

La clave privada se cifra utilizando el algoritmo AES-GCM para garantizar su confidencialidad. La clave de cifrado para AES-GCM se deriva de la contraseña del usuario empleando PBKDF2-HMAC-SHA256, como ya hacíamos para la primera entrega ya que era nuestro método principal de cifrado, el simétrico. Los parámetros resultantes del cifrado, como el nonce, el tag y el salt, se almacenan junto con la clave privada cifrada en el archivo usuarios.json. Este enfoque asegura que la clave privada no pueda ser utilizada sin la contraseña del usuario.

Por otro lado, la clave pública se serializa en formato PEM y se almacena directamente en el archivo usuarios.json. Se usa en la verificación de firmas digitales y en la emisión de certificados por parte de la autoridad certificadora (CA) del sistema.

Gestión de Firmas Digitales

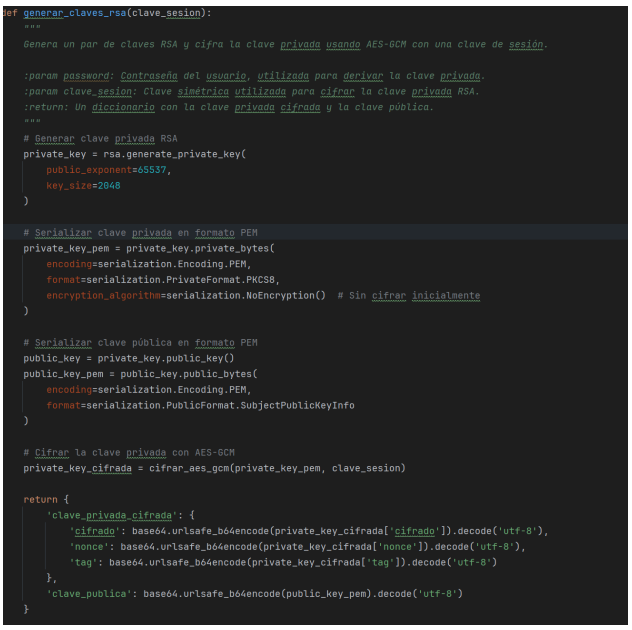
La autenticidad e integridad de los datos se garantiza mediante el uso de firmas digitales, implementadas utilizando RSA con el esquema de padding probabilístico PSS y la función de hash SHA-256. La función de generación de firmas digitaliza los datos en un formato JSON ordenado antes de firmarlos con la clave privada del usuario. Esto asegura que los datos sean consistentes y fácilmente verificables.

La firma resultante se codifica en Base64 para su almacenamiento y transmisión en el archivo de usuarios. Durante la validación de datos, la firma digital se verifica utilizando la clave pública correspondiente. Este procedimiento garantiza que los datos no hayan sido alterados y que provengan de una fuente legítima. Cualquier manipulación en los datos o en la firma resultará en una verificación fallida.

Almacenamiento Seguro

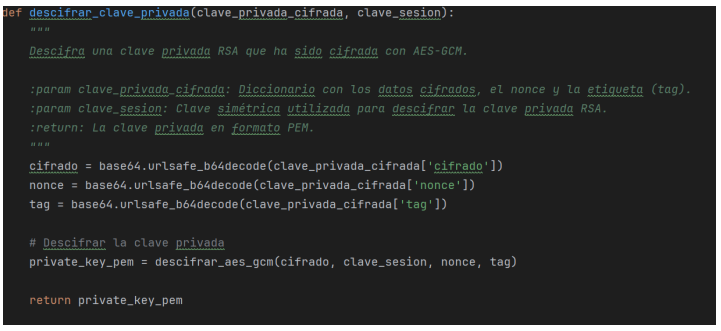
Se emplean técnicas avanzadas de cifrado y autenticación para proteger el archivo JSON contra accesos no autorizados. Además, el uso de un nonce único por operación de cifrado mitiga posibles ataques de repetición, mientras que las claves derivadas con PBKDF2 añaden resistencia contra intentos de descifrado directo.

encryption.py
generación clave rsa:



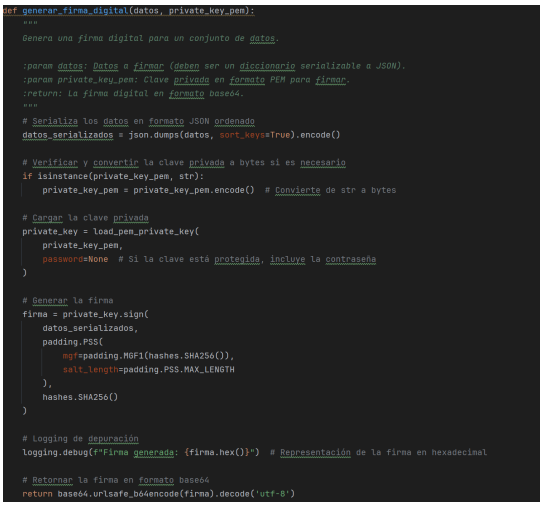
```python
def generar_claves_rsa(clave_sesion):
    """
    Genera un par de claves RSA y cifra la clave privada usando AES-GCM con una clave de sesión.

    :param password: Contraseña del usuario, utilizada para derivar la clave privada.
    :param clave_sesion: Clave simétrica utilizada para cifrar la clave privada RSA.
    :return: Un diccionario con la clave privada cifrada y la clave pública.
    """
    # Generar clave privada RSA
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )

    # Serializar clave privada en formato PEM
    private_key_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()  # Sin cifrar inicialmente
    )

    # Serializar clave pública en formato PEM
    public_key = private_key.public_key()
    public_key_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    # Cifrar la clave privada con AES-GCM
    private_key_cifrada = cifrar_aes_gcm(private_key_pem, clave_sesion)

    return {
        'clave_privada_cifrada': {
            'cifrado': base64.urlsafe_b64encode(private_key_cifrada['cifrado']).decode('utf-8'),
            'nonce': base64.urlsafe_b64encode(private_key_cifrada['nonce']).decode('utf-8'),
            'tag': base64.urlsafe_b64encode(private_key_cifrada['tag']).decode('utf-8')
        },
        'clave_publica': base64.urlsafe_b64encode(public_key_pem).decode('utf-8')
    }
```

La clave privada se cifra y se descifra usando cifrado simétrico para mayor robustez.

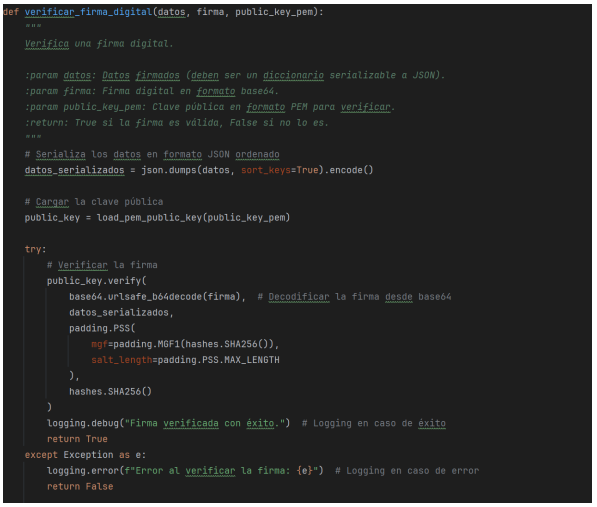


```python
def descifrar_clave_privada(clave_privada_cifrada, clave_sesion):
    """
    Descifra una clave privada RSA que ha sido cifrada con AES-GCM.

    :param clave_privada_cifrada: Diccionario con los datos cifrados, el nonce y la etiqueta (tag).
    :param clave_sesion: Clave simétrica utilizada para descifrar la clave privada RSA.
    :return: La clave privada en formato PEM.
    """
    cifrado = base64.urlsafe_b64decode(clave_privada_cifrada['cifrado'])
    nonce = base64.urlsafe_b64decode(clave_privada_cifrada['nonce'])
    tag = base64.urlsafe_b64decode(clave_privada_cifrada['tag'])

    # Descifrar la clave privada
    private_key_pem = descifrar_aes_gcm(cifrado, clave_sesion, nonce, tag)

    return private_key_pem
```

digital_signature.py



```python
def generar_firma_digital(datos, private_key_pem):
    """
    Genera una firma digital para un conjunto de datos.

    :param datos: Datos a firmar (deben ser un diccionario serializable a JSON).
    :param private_key_pem: Clave privada en formato PEM para firmar.
    :return: La firma digital en formato base64.
    """
    # Serializa los datos en formato JSON ordenado
    datos_serializados = json.dumps(datos, sort_keys=True).encode()

    # Verificar y convertir la clave privada a bytes si es necesario
    if isinstance(private_key_pem, str):
        private_key_pem = private_key_pem.encode()  # Convierte de str a bytes

    # Cargar la clave privada
    private_key = load_pem_private_key(
        private_key_pem,
        password=None  # Si la clave está protegida, incluye la contraseña
    )

    # Generar la firma
    firma = private_key.sign(
        datos_serializados,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    # Logging de depuración
    logging.debug(f"Firma generada: {firma.hex()}")  # Representación de la firma en hexadecimal

    # Retornar la firma en formato base64
    return base64.urlsafe_b64encode(firma).decode('utf-8')
```



```python
def verificar_firma_digital(datos, firma, public_key_pem):
    """
    Verifica una firma digital.

    :param datos: Datos firmados (deben ser un diccionario serializable a JSON).
    :param firma: Firma digital en formato base64.
    :param public_key_pem: Clave pública en formato PEM para verificar.
    :return: True si la firma es válida, False si no lo es.
    """
    # Serializa los datos en formato JSON ordenado
    datos_serializados = json.dumps(datos, sort_keys=True).encode()

    # Cargar la clave pública
    public_key = load_pem_public_key(public_key_pem)

    try:
        # Verificar la firma
        public_key.verify(
            base64.urlsafe_b64decode(firma),  # Decodificar la firma desde base64
            datos_serializados,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        logging.debug("Firma verificada con éxito.")  # Logging en caso de éxito
        return True
    except Exception as e:
        logging.error(f"Error al verificar la firma: {e}")  # Logging en caso de error
        return False
```

3. ¿Cómo se generan los certificados de clave pública? ¿Qué jerarquía de autoridades de certificación se ha desplegado? ¿Por qué ha escogido esta configuración y no otra? ¿Cómo se ha implementado? ¿En qué momento se utilizan los certificados y para qué?

3.1. ¿Cómo se generan los certificados de clave pública?

Generación de Certificados de Clave Pública

La generación de certificados de clave pública en el sistema sigue un proceso que involucra la autoridad certificadora (CA) y utiliza estándares establecidos para la emisión de certificados digitales. El procedimiento consta de:

1. Generación de la Clave Privada y Pública del Usuario

Antes de emitir un certificado, el usuario genera un par de claves RSA (privada y pública) durante su registro. La clave pública generada será la base para el certificado. Este proceso se realiza mediante las funciones relacionadas en el archivo encryption.py y garantiza que el usuario tenga un par de claves exclusivo.

2. Solicitud del Certificado

Cuando el usuario solicita un certificado, el sistema valida la existencia de una clave pública para el usuario. Si la clave pública no está registrada, se lanza un error indicando que el usuario debe generar su par de claves.

3. Descifrado de la Clave Privada de la CA

La CA utiliza su propia clave privada para firmar los certificados. Esta clave privada está cifrada con AES-GCM y almacenada en el archivo ca_key.json. Para descifrarla, se requiere la contraseña maestra de la CA, que se solicita al administrador. La contraseña se utiliza para derivar una clave con

PBKDF2-HMAC-SHA256, que a su vez descifra la clave privada de la CA.

4. Construcción del Certificado

Definición del usuario y el Emisor:

El sujeto del certificado se configura con información relevante, como el nombre del usuario y su organización.
El emisor del certificado es la propia CA, que actúa como autoridad certificadora.

Firma del Certificado:

El certificado se firma con la clave privada de la CA utilizando el algoritmo RSA y el hash SHA-256.

5. Serialización y Entrega

El certificado generado se serializa en formato PEM. Este certificado se codifica en Base64 y se asocia al usuario en el archivo usuarios.json. El certificado firmado garantiza que la clave pública del usuario es legítima y ha sido verificada por la CA.

3.2. ¿Qué jerarquía de autoridades de certificación se ha desplegado? ¿Por qué ha escogido esta configuración y no otra? ¿Cómo se ha implementado?

Jerarquía de Autoridades de Certificación

El sistema utiliza una autoridad de certificación única (CA raíz) para emitir y firmar certificados digitales. Esta CA raíz es la única responsable de firmar los certificados de clave pública de los usuarios registrados.

Razón para Escoger esta Configuración

Simplicidad:

Una configuración de CA raíz es más fácil de implementar y administrar en sistemas pequeños o medianos, donde no se requiere una estructura jerárquica más compleja como una CA subordinada.
Este diseño es adecuado para el alcance de un gestor de contraseñas, que tiene un número limitado de usuarios y no necesita delegar funciones de certificación.

Control Centralizado:

La CA raíz centraliza el control y garantiza que todos los certificados sean emitidos directamente por una única entidad confiable.
Esto facilita la validación de los certificados.

Menor Complejidad Operativa:

Las jerarquías más complejas, como una CA raíz con CAs subordinadas, requieren una gestión más avanzada. Para nuestro gestor de contraseñas estas características no son necesarias ya que todos los usuarios confían en una única CA.

Implementación de la CA

1. Generación de la Clave Privada de la CA
La clave privada de la CA se genera utilizando el algoritmo RSA. Esta clave se almacena de manera segura cifrada con AES-GCM, utilizando una contraseña maestra que protege el acceso. El cifrado se implementa mediante la función cifrar_aes_gcm del archivo encryption.py

2. Creación del Certificado de la CA
El certificado de la CA se genera mediante la función generar_ca en el archivo certificate_management.py
El proceso incluye:

- Definir el emisor y el sujeto como la propia CA.
- Configurar el certificado con una validez de 10 años.
- Firmar el certificado con la clave privada de la CA utilizando RSA y SHA-256.
- Serializar el certificado en formato PEM para su almacenamiento.

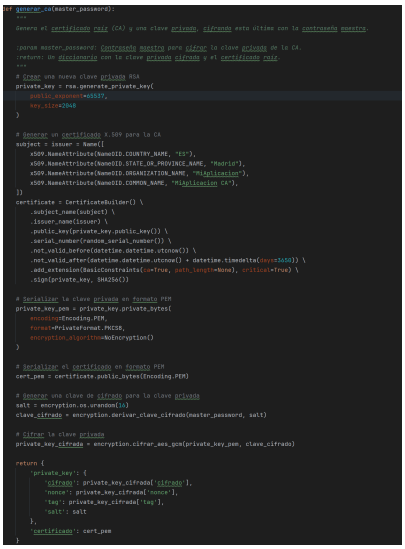
4. Validación de Certificados
El sistema incluye una función llamada validar_certificado que utiliza la clave pública de la CA para verificar que un certificado de usuario fue firmado por la entidad raiz.

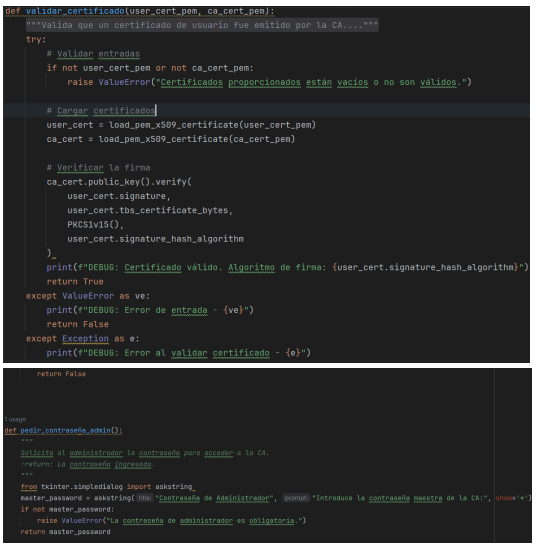
3.3. ¿En qué momento se utilizan los certificados y para qué?

En la implementación del gestor de contraseñas el certificado se utiliza para verificar que el usuario que está intentando acceder a consultar las contraseñas almacenadas en el gestor es una persona autorizada por la entidad raíz, verificando que es un usuario autorizado y se le permite consultarlas y poder modificarlas.

La utilidad del certificado para nuestra aplicación solo se desempeña en la administración de las contraseñas, ya que es la idea principal de la aplicación y es la información que estar certificado para consultarla.

certificate_managment.py





Además, una función de emitir_certificado que por cuestión de espacio y longitud del código no la reflejamos en captura de pantalla, pero puede consultarla en el mismo archivo .py que el resto de ellas.
La emisión del certificado genera un .PEM y un JSON para almacenar el certificado y una clave, respectivamente.
En usuarios.json se puede apreciar como se guarda el certificado asociado para cada usuario junto con sus credenciales.

4. Discuta la complejidad y diseño del código de su aplicación.

4.1. Arquitectura del Sistema

El código está organizado en módulos independientes. Esta separación permite una clara división de tareas y facilita la comprensión del código. Los principales módulos incluyen:

Gestión de usuarios (register_login.py):

Maneja el registro y autenticación de usuarios.
Incluye validaciones de datos sensibles como contraseñas y correos electrónicos.

Cifrado (encryption.py):

Proporciona funciones de cifrado y descifrado simétrico (AES-GCM) y asimétrico (RSA).
Implementa la derivación de claves mediante PBKDF2.

Firmas digitales (digital_signature.py):

Gestiona la creación y verificación de firmas digitales utilizando RSA-PSS y SHA-256.

Certificados (certificate_management.py):

1. Genera certificados firmados por la CA.
2. Valida los certificados emitidos para garantizar su autenticidad.

4.2. Complejidad Técnica

Cifrado simétrico y asimétrico:

La integración de AES-GCM para proteger datos sensibles como claves privadas y contraseñas añade un nivel alto de seguridad, pero también requiere manejo cuidadoso de nonce y etiquetas de autenticación (tag).
RSA se utiliza tanto para la generación de firmas como para la emisión de certificados, lo que implica múltiples procesos de serialización y validación.

Firmas digitales y certificados:

La generación de firmas digitales con RSA-PSS introduce aleatoriedad, lo que aumenta la seguridad pero complica la verificación.
La gestión de certificados requiere interacciones entre la CA y las claves de los usuarios.

Validaciones y Manejo de Errores

Las validaciones de contraseñas, correos electrónicos y nombres de usuario siguen reglas estrictas definidas en data_validation.py. Esto asegura datos consistentes pero también aumenta la complejidad del código.

4.3. Diseño del Código

Puntos Fuertes

Modularidad:

La división en módulos específicos hace que el código sea más fácil de leer, mantener y extender.

Enfoque en la Seguridad:

Se utilizan prácticas criptográficas modernas y seguras, como AES-GCM, RSA con PSS, y PBKDF2, siguiendo estándares actuales.

Facilidad de Uso:

La interfaz gráfica (app.py) proporciona un punto de entrada intuitivo para los usuarios.

Control de Errores:

Los errores son capturados y gestionados correctamente, lo que mejora la robustez del sistema.

Limitaciones

Escalabilidad:

El uso de un archivo JSON para almacenar datos puede volverse ineficiente a medida que el número de usuarios y contraseñas aumenta.
Para ello, podríamos organizar la información en una base de datos como la extensión que nos permite pycharm con sql.
La podríamos implementar si escalásemos la aplicación y fuésemos a recibir mayor volumen de datos.
Podría ser una mejor a implementar.

5. Si ha realizado mejoras, explique cuáles y las implicaciones de seguridad de cada una de ellas en su programa/aplicación.

1. Validación Exhaustiva de Datos de Entrada

Se ha desarrollado un módulo (data_validation.py) para validar los datos de entrada proporcionados por los usuarios. Este módulo verifica:

Nombres de usuario
Contraseñas
Correos electrónicos
Teléfonos

Implicaciones de Seguridad:
Previene ataques de inyección de código y uso de datos malformados que podrían comprometer el sistema.

2. Manejo Sistemático de Excepciones

Se ha implementado una gestión estructurada de errores en todo el sistema mediante clases personalizadas de excepciones (exceptions.py). Esto asegura:

Captura y manejo de errores específicos como ValidationError, mejorando la capacidad del sistema para recuperarse de fallos.
Registro detallado de errores, lo que facilita la depuración y el análisis.

Implicaciones de Seguridad:

- Evita que fallos inesperados expongan información sensible.

- Proporciona mensajes de error específicos sin revelar detalles internos.

3. Interfaz Gráfica de Usuario

Se ha desarrollado una interfaz gráfica basada en Tkinter que permite a los usuarios interactuar fácilmente con el sistema. Esta interfaz presenta de forma clara y accesible las funcionalidades principales.

Ventajas de la Interfaz Gráfica:

Accesibilidad:

Permite que usuarios con conocimientos limitados de tecnología accedan a la funcionalidad completa del sistema.

Feedback en tiempo real:

Las operaciones importantes generan mensajes informativos o de error, mejorando la experiencia del usuario.

Organización visual:

Se han incluido pestañas o secciones bien diferenciadas para cada funcionalidad (registro, autenticación, gestión de contraseñas, etc.).

4. Gestión de Cambio de Contraseña vía Correo Electrónico

Para fortalecer la seguridad y proporcionar un control exhaustivo de las sesiones, se ha implementado un sistema de cambio de contraseña que incluye:

Envío de correos electrónicos de notificación: Cuando un usuario solicita cambiar su contraseña, el sistema envía un correo electrónico a la dirección registrada.
El correo incluye una notificación clara y advierte al usuario en caso de que no haya realizado esta acción.

Descifrado seguro del correo electrónico: Los correos electrónicos se almacenan cifrados en el sistema (AES-GCM) y se descifran temporalmente para enviar la notificación.

Sobre algunas de estas mejoras añadidas en el trabajo las tratamos en la memoria de la entrega 1 que es cuando planteamos la mayoría de ellas. En esta segunda entrega hemos perfeccionado el tratamiento de datos y la depuración de mensajes, aparte de algunas de las funcionalidades extra explicadas previamente.

El objetivo principal de está práctica es hacer una implementación de una aplicación práctica basada en satisfacer un problema que tienen la mayoría de los usuarios de internet. Nuestra finalidad es proporcionar una app capaz de satisfacer esas necesidades y, además, siguiendo unas buenas prácticas de cifrado en la información y tratamiento de los datos.

No hemos podido meter todos los fragmentos de código que nos gustaría por no exceder el número de páginas permitidas, pero al disponer del código base, hemos preferido dejarlo bien referenciado en la memoria.