uc3m | Universidad Carlos III de Madrid

Entregable 2

Grupo 85

ID de grupo de prácticas 29

Nombre de todos los alumnos:

Claudia Fernández Menéndez | Ines Fuai Guillén Peña | Nicola Stefania Cindea

Correo de todos los alumnos:

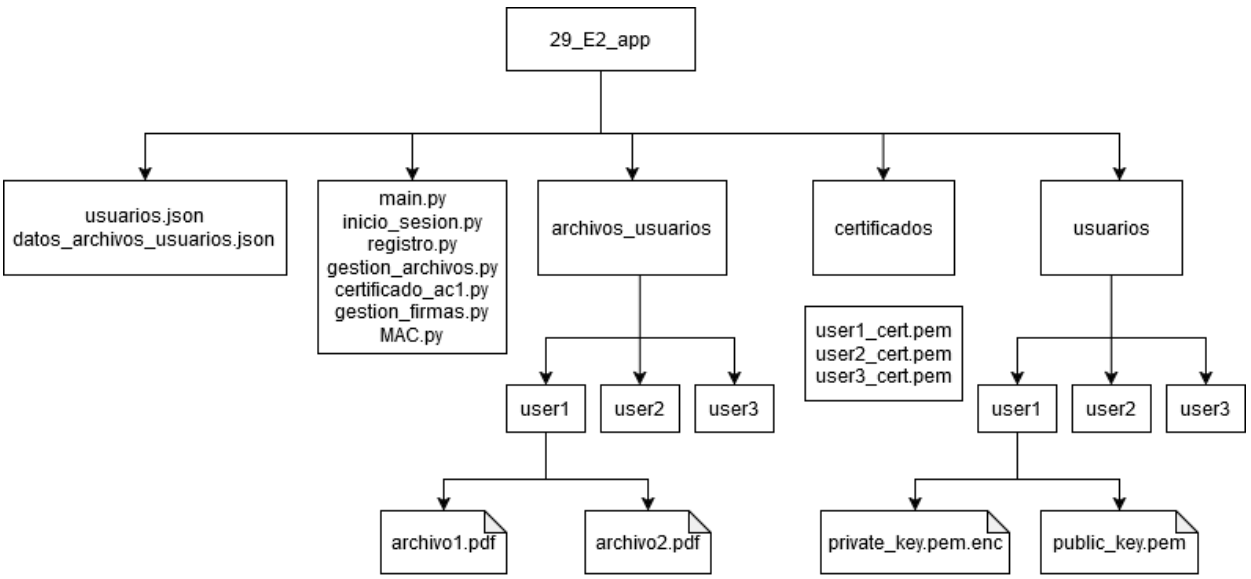
100495919@alumnos.uc3m.es | 100495752@alumnos.uc3m.es
100495713@alumnos.uc3m.es

1. ¿Cuál es el propósito de su aplicación? ¿Cuál es su estructura interna?

El objetivo de nuestra aplicación ha sido desarrollar un sistema sencillo de manejo de archivos PDF (*Portable Document Format*). La aplicación se maneja enteramente por la consola de salida de PyCharm. Nada más comenzar, el usuario es presentado con 3 opciones: iniciar sesión, registrarse y salir. Tras iniciar sesión (con previo registro si fuese necesario) se le presentarán las opciones en lo referente a sus propios archivos: ver todos los archivos (formato de lista numerada), abrir un archivo específico, añadir un archivo, eliminar un archivo, firmar un archivo, verificar firma de un archivo, renovar certificado y salir (que devuelve al usuario a las tres opciones iniciales). Estaba inicialmente orientado a ser usado como gestor de archivos médicos, aunque por el momento es compatible como gestor para cualquier archivo con extensión .pdf.

La estructura interna consiste en archivos JSON, archivos.py y carpetas para los datos de los usuarios:

- Archivos JSON. Tendremos dos. El archivo *usuarios.json* actúa como base de datos. En él, guardamos los datos de los usuarios como un array de diccionarios. El archivo *datos_archivos_usuarios.json* almacena por cada usuario un directorio madre (a la carpeta en la que se guardarán sus archivos) y una lista con el nombre de todos sus archivos.
- Archivos.py. Contienen el código del proyecto. Tenemos un *main.py,* que contiene toda la parte de la "interfaz" y es el archivo que debemos ejecutar para que la aplicación funcione. El resto de *.py* se usan para ser llamados y realizar distintas funciones, dependiendo del contexto.
- Carpetas. Tenemos una carpeta principal *archivos_usuarios,* dentro de la cual se generan subcarpetas con el mismo nombre del usuario que tienen asignado, en las que se almacenan los archivos de cada usuario. Esta estructura está pensada para que un usuario estándar no pueda acceder a los datos de otro, pues no tiene acceso a la ruta madre de la carpeta del otro usuario. Además, generamos una carpeta *certificados* donde guardamos los certificados de todos los usuarios y otra carpeta *usuarios* donde tendremos, organizado en subcarpetas para cada usuario, las claves públicas y privadas (encriptadas).



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

La firma digital se utiliza para garantizar autenticidad e integridad de los archivos subidos por cada usuario al sistema. El sistema permite al usuario firmar un archivo para garantizar que ese archivo fue firmado por el propietario de la clave privada. En cuanto a la verificación de la firma, la función se encarga de comprobar que el archivo no ha sido modificado desde que se firmó. La validación de la firma puede ser llevada a cabo por terceras partes con la clave pública del usuario.

A continuación explicaremos el funcionamiento de la firma y los algoritmos utilizados en cada parte.
Dentro de la clase GestionFirmas se genera el par de claves (pública y privada) para cada usuario. Cuando el usuario inicia sesión, se genera o se cargan estas claves.

generar_claves:

- **clave privada:** se genera utilizando el algoritmo RSA con un tamaño de 2048 bits para garantizar alta seguridad. La clave se serializa (convertir la clave a un formato que se pueda almacenar) y se almacena en formato PEM (Privacy Enhanced Mail) utilizando el estándar PKCS8 (Public Key Cryptography Standard #8) que define como se estructura la clave privada, incluyendo información adicional como el algoritmo de cifrado. Para el cifrado, se utiliza la mejor opción disponible y se cifra con una clave derivada de la contraseña del usuario. Esta clave derivada también fue generada en el momento de inicio de sesión. (es la clave que se utiliza para cifrar y descifrar archivos) Se explicará cómo se genera esa clave en la parte de de complejidad y diseño del sistema (punto 4).

```
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
```

Se genera y se almacena.

```
with open(self.private_key_file, "wb") as private_file: #abrir el archivo de la clave privada
    private_bytes = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.BestAvailableEncryption(self.clave_derivada)
```

- **clave pública:** se genera a partir de la clave privada y se serializa para poder almacenarla en formato PEM utilizando el estándar SubjectPublicKeyInfo que define la estructura de claves públicas.

cargar_clave_privada:

- si no es la primera vez que el usuario inicia sesión , las claves ya estarán generadas y tendremos que cargar la clave privada desde el archivo. No es necesario cargar la clave pública ya que se puede derivar de la privada. Para cargar la clave, tendremos que descifrar la clave privada con la clave derivada de la contraseña. Almacenaremos esta clave descifrada temporalmente en una variable mientras la sesión está iniciada para facilitar la generación de la firma.

```
with open(self.private_key_file, "rb") as private_file:  #
    self.private_key = serialization.load_pem_private_key(
        private_file.read(),
        password=clave_derivada,
```

Descifrar para cargar.

firmar_archivo:

- garantiza que el archivo no pueda ser alterado sin invalidar la firma. Se utiliza el algoritmo SHA-256 para generar un resumen del archivo de 256 bits. La función firma el hash del archivo en vez del archivo completo para optimizar y agilizar el proceso. Para la firma se utiliza la clave privada del usuario y el esquema PSS (Probabilistic Signature Scheme) especialmente diseñado para RSA. PSS permite incluir un salt aleatorio dentro del padding para garantizar que si el archivo es el mismo, la firma sea diferente, aumentando la seguridad. El relleno se genera con MGF1 (Mask Generation Function 1) utilizando SHA-256 como algoritmo subyacente. La firma resultante se guarda en un archivo .sig.

```
with open(ruta_archivo, "rb") as archivo:
    datos = archivo.read()
hash_archivo = hashes.Hash(hashes.SHA256())
hash_archivo.update(datos)
resumen = hash_archivo.finalize()
firma = self.private_key.sign(
    resumen,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
```

Generar firma con el hash.

verificar_firma:

- Cargamos la clave pública desde el certificado del usuario y se lee el archivo y su firma correspondiente. Volvemos a generar un hash del archivo requerido y para ello volvemos a utilizar SHA-256. Utilizamos un método de RSA que usa la clave pública para verificar que la firma corresponde al hash generado. Para ello recibe la firma, el resumen y genera nuevamente el padding con el salt anterior.

```
try:
    public_key.verify(
        firma,
        resumen,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
```

Primero crea el hash y luego verifica.

En conclusión, el sistema utiliza criptografía asimétrica para proporcionar un enfoque seguro. Los algoritmos RSA, SHA-256 y PSS garantizan seguridad, integridad y autenticidad mientras que el almacenamiento de la clave privada cifrada protege la identidad del usuario (confidencialidad). La clave privada garantiza que solo el propietario pueda generar una firma válida; mientras que la pública se puede compartir y permite a usuarios externos comprobar que la firma no ha sido alterada.

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?

Generar AC1 (Autoridad Certificadora) se encuentra en un archivo aparte, *certificado_ac1.py*, dedicado exclusivamente a crearlo, para que sea común a todos los usuarios. Se genera al principio de main.py, devolviendo el certificado de la autoridad y su clave privada. Mientras exista esta aplicación, tanto el certificado como la autoridad existirán como variables dentro del *main*:

1. *generar_certificado_ac1:* Primero, se crea la clave privada de la autoridad, y con esa clave genera una Solicitud de Firma de Certificado (CSR). Luego, esta solicitud se firma con la clave privada del usuario para demostrar que la ha creado él.

```
clave_privada_ac1 = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)  # con el estándar RSA para el exponente y un tamaño de 2048 bits
```

   Después, la Autoridad de Certificación (AC1), que somos nosotros en este caso, firma esta solicitud con su propia clave privada para emitir un certificado. Este certificado sigue el estándar X.509 y tiene toda la información necesaria para identificar al usuario y garantizar que el certificado es válido y confiable.

```
subject = issuer = x509.Name([
    x509.NameAttribute(NameOID.COMMON_NAME, value: "AC1"),
])
```

El resto de funciones que vamos a describir a continuación se generan dentro de *gestion_firmas.py:*

1. *generar_certificado_usuario:* Cuando un usuario solicita un certificado, lo primero que hacemos es generar una Solicitud de Firma de Certificado (CSR). Este CSR contiene información importante del usuario, como su nombre, correo electrónico y su centro médico. Se crea una firma de esta solicitud con la clave privada del usuario (recordemos que está almacenada en una variable temporal ya descifrada) para que la autoridad sepa que se puede fiar de que la solicitud fue mandada por la persona correspondiente.

```
csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
    x509.NameAttribute(NameOID.ORGANIZATION_NAME, value: "Centro Médico"),
    x509.NameAttribute(NameOID.COMMON_NAME, usuario_info["nombre"] + " " + usuario_info["apellido1"] + " " + usuario_info["apellido2"]),
    x509.NameAttribute(NameOID.EMAIL_ADDRESS, usuario_info["mail"])
])).sign(self.private_key, hashes.SHA256())  # clave privada usuario ya en el __init__
```

   Luego, la AC1 firma esta solicitud con su clave privada para emitir un certificado que contiene tanto la identidad como la clave pública del usuario, garantizando que el usuario es confiable y que su certificado ha sido validado por la AC1. El proceso de crear el certificado de AC1 y el usuario es el mismo y solo cambia la clave privada que lo firma (ambos se firman con la clave privada de AC1)

```
certificado_usuario = x509.CertificateBuilder().subject_name(
    csr.subject
).issuer_name(
    self.certificado_ac1.subject  # AC1 debe "validar" nuestro certificado
).public_key(
    csr.public_key()
).serial_number(
    x509.random_serial_number()
).not_valid_before(
    datetime.datetime.utcnow()
).not_valid_after(
    datetime.datetime.utcnow() + datetime.timedelta(days=365)
    # El certificado será válido por los días especificados
).sign(
    self.clave_privada_ac1, hashes.SHA256()  # Firmamos el certificado con la clave privada de la AC1
)
```

2. *cargar_certificado:* Abrimos el certificado PEM y validamos que sea correcto usando la función *verificar_certificado*

3. *verificar_certificado:* Cargamos los certificados tanto del AC1 como del usuario, y verificamos la firma del usuario con la clave pública del AC1. La función public.key().verify() coge la clave pública de la AC1 y recibe como parámetros la firma del certificado del usuario, el certificado, el padding y el hash usado para la firma. Se comprueba también la validez del certificado, utilizando "*verficiar_validez*", para parar la verificación del certificado si estuviera caducado.

4. *verificar_validez:* Recibe un certificado y saca su valor *not_valid_after*, es decir, hasta cuando es válido. Lo compara con el momento en el que llamamos a esta función (*datetime.datetime.utcnow()*) para ver si el certificado está caducado. Se usará para renovar un certificado y también para verificar el certificado antes de hacer cualquier operación con él.

5. *renovar_certificado_usuario:* Verifica la validez de un certificado. Si no está caducado, lanza un mensaje de que no es necesaria la renovación. En caso contrario genera un nuevo certificado con validez de un año para el usuario. Primero se verifica su validez con la función anterior. Si no es válido se genera uno nuevo cargando los datos del usuario y generando el certificado con la función generar_certificado_usuario(), que recibe la información cargada del usuario. Finalmente se guarda el certificado en la misma ruta que el anterior.

```
        #generamos nuevo certificado
        nuevo_certificado = self.generar_certificado_usuario(info_usuario)
        with open(ruta_certificado, "wb") as f:
            # sobreescribimos el certificado caducado
            f.write(nuevo_certificado.public_bytes(serialization.Encoding.PEM))

        print("Renovación exitosa!")
        return
```

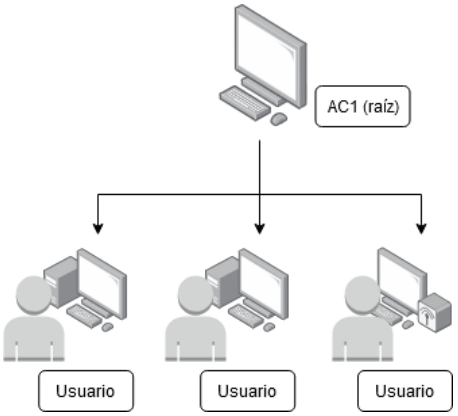
La renovación lanza warnings de la librería cryptography por un problema de compatibilidad con zonas horarias. Es completamente funcional y no afecta a su funcionamiento, ni al funcionamiento del resto de la aplicación tras aparecer.

6. *guardar_certificado:* Guarda un certificado en la carpeta "*certificados*" como "*nombre_usuario_cert.pem*", el nombre de usuario lo recibe como parámetro

7. *autenticar_con_certificado:* Se iba a utilizar para que un usuario pudiera iniciar sesión directamente con su certificado sin requerir de contraseña, pero lamentablemente no hemos conseguido que sea funcional.

Hemos montado una **jerarquía** con **una sola Autoridad de Certificación raíz (AC1)**. Esta AC1 hace todo: genera certificados, valida solicitudes y firma los certificados de los usuarios. No hemos puesto autoridades intermedias porque, debido a la simplicidad de la aplicación, no hemos considerado necesario implementar más de un nivel, pues nuestros usuarios sólo se comunican con la app para trabajar con archivos. Esta configuración funciona porque es un entorno controlado y cerrado. Estamos trabajando con un grupo manejable y no necesitamos la estructura complicada y compleja de un sistema más grande.

Elegimos esta configuración porque es más fácil de gestionar. Tener una sola AC raíz nos permite tener control completo sin tener que preocuparnos por la administración de varias autoridades. Como sabemos quiénes son los usuarios, no hace falta montar una infraestructura más pesada.



Lo implementamos usando la biblioteca **cryptography** de Python.

Primero generamos una clave privada para la AC1 usando RSA. Luego, con esa clave, creamos un certificado autofirmado que identifica a la AC1 como la raíz de confianza.

Para los usuarios, cada vez que necesitan un certificado, se genera un CSR con sus datos. Esa solicitud la firma la AC1, y se emite un certificado válido. Guardamos estos certificados en archivos .pem para que sea más fácil usarlos y verificarlos después.

Usaremos los certificados en dos situaciones diferentes:

- Inicio de sesión. Si un usuario elige la opción de iniciar sesión usando su certificado, el sistema lo compara con el certificado raíz de la AC1 para asegurarse de que es válido. Si todo coincide, la autenticación se considera exitosa. Hemos decidido esta implementación como una barrera más de seguridad, pues aumentamos la seguridad frente a ataques como el phishing o los ataques de fuerza bruta. Además dado el periodo de validez, este certificado se regenerará de forma periódica (siempre y cuando el usuario solicite uno nuevo). *El inicio con certificado finalmente no es funcional.*

- Firma. También usamos los certificados para verificar la firma de archivos. Si un archivo está firmado con la clave privada del usuario, podemos verificarlo con su certificado público ya que contiene la clave pública del usuario. Esto nos garantiza que el archivo no ha sido alterado y que la firma realmente pertenece a ese usuario. De esta forma, mantenemos la confianza y seguridad en la gestión de los datos. También permite a una persona externa verificar una firma de un usuario, ya que el certificado contiene la clave pública del usuario, necesaria para esta verificación.

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

El diseño y la complejidad de la aplicación se han estructurado para lograr un balance entre seguridad, funcionalidad y facilidad de uso, integrando elementos avanzados de criptografía y gestión de archivos.
Durante las anteriores preguntas ya hemos explicado cómo funciona nuestro código con detalle, por lo que intentaremos hacer un resumen visual de todo el sistema.

Como hemos mencionado anteriormente, la firma está basada en criptografía asimétrica lo que añade robustez al sistema. Las claves se generan al iniciar sesión y se almacenan cada una en su archivo PEM correspondiente. La clave pública no se cifra, mientras que la privada se cifra con la clave derivada de la contraseña del usuario para otorgar una capa extra de seguridad. El hecho de firmar el hash del archivo en vez de firmar los datos completos hace que nuestro sistema sea más eficiente. En cuanto al diseño de la firma, se manejan excepciones de que claves inexistentes generando nuevas claves si fuera necesario. Hay una clara separación de responsabilidades evitando redundancia en el código.

El módulo de gestión de archivos permite subir, ver y abrir archivos mediante un sistema de MAC que verifica la integración de los archivos. Los archivos son cifrados con la clave derivada de la contraseña del usuario justo antes de almacenarlos. Se genera un MAC del archivo que se sube al sistema para asegurar que el archivo no fue alterado. Este módulo tiene una estructura lógica con operaciones (subir, ver, eliminar y abrir) modularizadas, facilitando su mantenimiento y escalabilidad. El uso de buffers para leer y escribir mejora el rendimiento y evita problemas de memoria. Las operaciones relacionadas con la generación y verificación del MAC se encuentran en un archivo de python aparte para conseguir que la estructura del sistema sea aún más clara. Estas funciones se utilizan únicamente en la gestión de archivos a la hora de subir o abrir archivos pdf.

Dentro del inicio de sesión es donde se deriva la clave de la contraseña que introduce el usuario usando PBKDF2-HMAC con SHA-256. La clave se deriva una vez que el sistema ya ha comprobado que la contraseña es correcta, esta clave se devuelve en el método de *inicio_sesion()* para que se pueda almacenar temporalmente mientras el usuario esté usando la aplicación. Su uso es exclusivo para cifrar y descifrar archivos; generar y verificar MAC; y cifrar la clave privada de la firma (evita redundancia). Gracias al uso del salt evitamos que dos contraseñas idénticas generen la misma clave. La validación y derivación de claves están alineadas con el proceso de inicio de sesión, evitando operaciones innecesarias.

Se ha implementado un sistema de certificados digitales para reforzar la autenticidad y confianza entre las partes. Se generan en un formato X.509, lo que permite su compatibilidad con otros sistemas de seguridad estándar. Los certificados son firmados por nuestra autoridad interna AC1 y validados para evitar falsificaciones.

El certificado AC1 se firma a sí mismo con su propia clave privada y firma también los certificados de los usuarios con su clave privada. A la hora de verificar si el certificado del usuario es válido, utiliza la clave pública de AC.

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

- Validación de los datos introducidos:
  - Contraseña: hay un máximo de 3 intentos para introducir la contraseña incorrecta, después de esto no se permitirá acceder a la aplicación. (Optativa: iniciar sesión con un certificado digital).
  - Usuario ya existente: si se detecta que el usuario ya ha sido registrado con los mismos datos, no se permite volver a registrarlo y se muestra la opción de iniciar sesión.
  - Nombre de usuario: si se detecta que dos usuarios tienen el mismo nombre, se le pide al usuario que escoja otro. (Se refiere al nombre que se registra en la aplicación, no al nombre real de la persona).
  - Correo electrónico: comprueba que el correo tenga un formato válido.

- Clave derivada de la contraseña del usuario: mejora con respecto al entregable 1. La función de *derivar_clave()* se ha incluido dentro del archivo de inicio de sesión para generar una clave derivada en el momento en el que se valida la contraseña del usuario para iniciar sesión. El hecho de que se almacene temporalmente mientras la sesión está iniciada, evita que se pida la contraseña cada vez que ocurre una operación que requiere la clave.

- Captura de excepciones: El código referente a lo que sería la propia interfaz por la que el usuario se va a mover tiene algunas restricciones, como el formato en el que se debe introducir el correo electrónico, una doble verificación de contraseña o excepciones si se introdujera datos no válidos, como por ejemplo, tratar de generar un usuario cuyo nombre de usuario ya esté en nuestra base de datos.

- Otras mejoras: se ha modificado la forma de almacenar los usuarios en usuarios.json. Antes se almacenaban en una lista y ahora se utiliza un diccionario. Esto ayuda a que el sistema no tenga que recorrer toda la lista para encontrar a un usuario sino que lo encuentra con su nombre de usuario (key: nombre del usuario) y así ahorramos tiempo en caso de que la lista de usuarios sea muy larga. Por este motivo no podemos permitir que dos usuarios tengan el mismo nombre de usuario.