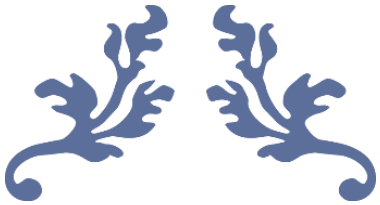


Criptografía y Seguridad Informática

PRÁCTICA DE CRIPTOGRAFÍA ENTREGA 2

Curso 2024/2025

Nombre	NIA	Correo Electrónico	Grupo
Javier Rosales Lozano	100495802	100495802@alumnos.uc3m.es	81
Manuel Roldán Matea	100500450	100500450@alumnos.uc3m.es	81

Fecha de presentación: 11/12/2024

Índice de contenidos

1. Introducción

El objetivo de la práctica de Criptografía y Seguridad Informática se centra en el conocimiento y aprendizaje del uso de librerías criptográficas, proponiendo como enunciado el desarrollo de una aplicación cuya funcionalidad es elegida por los alumnos, pero que haga uso de operaciones criptográficas en algún momento de su ejecución.

El siguiente documento presenta toda la información relacionada con el desarrollo de la segunda entrega de la práctica, atendiendo a los requerimientos que se piden y las funcionalidades solicitadas en el enunciado de ésta.

Primeramente, trataremos las decisiones de diseño tomadas en el trabajo, incluyendo las mejoras de código no especificadas en la primera entrega, y hablaremos también de la estética de la interfaz. Después, explicaremos el método de firma pedido en el enunciado (generación de conjunto de claves y firma digital con clave privada), seguido la autenticación de firmas mediante certificado de clave pública. Finalmente, concluiremos con el resultado obtenido durante el desarrollo del proyecto globalmente, y daremos nuestra opinión acerca del trabajo realizado.

2. Metodología de trabajo

En este apartado trataremos la metodología de trabajo realizado, así como todas las decisiones de diseño tomadas para el desarrollo de la interfaz y la calidad del código presentado.

2.1. Calidad del código y usabilidad

El código, como explicamos en la memoria de la primera entrega, se segmenta en diferentes módulos y directorios, de manera que se segmenten las operaciones y no se centralicen las funcionalidades de la aplicación. Con respecto a la primera entrega, se han dividido los módulos por directorios, de manera que podamos entender dónde aparece cada parte y, en caso de errores, separar el código correcto del que se tiene que revisar:

Directorio “data”	Incluye todos los archivos que guardan información acerca de los usuarios registrados y par de claves del sistema. Esto, a diferencia de la primera entrega (la cual se guardaba en la raíz del proyecto), aumenta la usabilidad y segmentación del código. Heredamos de la primera parte de la práctica un archivo .json que guarda los usuarios registrados junto con su password token y salt generado al cifrar la contraseña. También incluimos el directorio “data/notes”, donde guardamos archivos del tipo {usuario}.json, que almacenan las notas cifradas. Con lo que respecta a la segunda entrega, se han añadido los directorios “data/pks” y “data/sign”, que guardan el par de claves del sistema (clave privada y certificado de clave pública) y ficheros del tipo {usuario}.json con las firmas de cada nota de dicho usuario, respectivamente.

Directorio "docs"	Incluye la documentación del proyecto (memorias, enunciados y referencias del laboratorio de OpenSSL).
Directorio "openssl"	Incluye las entidades raíz y sistema (A y AC1, respectivamente) conteniendo la configuración y desarrollo de certificación de ambas, respecto a la práctica de OpenSSL realizada en el laboratorio de la asignatura.
Directorio "src"	Como su propio nombre indica ("src", de "source"), contiene el código fuente de la aplicación mencionado por módulos en la primera memoria del proyecto. Toda la lógica de la interfaz gráfica y funciones de cifrado se manejan dentro de este directorio.
Módulo "csr.py"	Este nuevo módulo se ejecuta antes de inicializar la aplicación, ya que genera el CSR (Certification Request) del sistema para realizar su certificado. Este certificado se almacena en un directorio "data/reqx509".
Módulo "main.py"	Ejecuta la aplicación.

2.2. Estética de la interfaz: decisiones de diseño

Se ha utilizado la librería Tkinter para el desarrollo de la interfaz gráfica. Por motivos de complejidad y falta de tiempo, decidimos realizar la aplicación usando esta librería, en vez de lanzarnos al desarrollo HTML de la aplicación. Creímos necesario centrarnos en el aspecto clave de la práctica (funciones de cifrado, firma y certificación autenticada) antes de invertir tiempo en el diseño visual de la interfaz. Sin embargo, creemos que el resultado obtenido es más que gratificante.

Por lo referente a las funciones de cifrado, las librerías utilizadas en este proyecto son Cryptography y Base64, las cuales nos han servido para la implementación de las distintas metodologías de cifrado de datos implementadas en la aplicación. Debemos destacar el uso de un "plugin" o extensión para la visualización de certificados digitales del tipo X.509 (X.509 Cert Buddy), el cual nos ha ayudado a entender cómo funcionaban los certificados, además de poder comprobar el correcto funcionamiento de estos.

3. Firma digital

A continuación, explicamos toda la documentación necesaria para entender la funcionalidad de firma implementada en el proyecto.

3.1. Generación de par de claves

El diseño implementado para la generación de las claves asimétricas se corresponde con el módulo genrate_pait_keys.py. En este archivo, la clase GeneratePairKey() se encarga de crear las instancias de claves asimétricas para el sistema, se instancia una contraseña como variable global para cifrar la clave privada de cara a su almacenamiento, y la correspondiente clave privada.

En cuanto al par de claves, solamente se crea un par de claves del sistema, de modo que todos los usos de la clave privada serán de la propia aplicación, tales como firmar, generar una petición de certificado, etc.

La clave privada se genera con el algoritmo RSA con una longitud de clave de 2048 bits a partir de la biblioteca Cryptography. De este modo se tendrá una clave privada pkv, de la cual podremos derivar la clave pública.

```python
def generate_keys(self):
        """Genera el par de claves para el sistema"""
        # Creamos la clave privada, a partir de la cual podremos obtener también la pública
        private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
        )
        return private_key
```

En cuanto al almacenamiento de la clave, ha evolucionado su funcionalidad durante el desarrollo de la segunda parte de la práctica. En primera instancia, se almacenaba la clave privada serializada y cifrada con la contraseña del sistema anteriormente mencionada. Este algoritmo de cifrado también lo proporciona la biblioteca Cryptography (con la funcionalidad de serialization), además de almacenar la clave pública para verificar la firma. Ambas claves se guardan en formato .pem siendo pkv.pem la clave privada y pku.pem la clave pública derivada.

El siguiente fragmento de código no corresponde con la entrega del código de la aplicación, pero sirve para entender los pasos intermedios referentes a la serialización del par de claves del sistema. Posteriormente la clave pública se sustituirá por un certificado firmado por una entidad raíz y con la clave privada del sistema correspondiente.

```python
def save_keys(self):
        """Crea un par de claves (pública y privada) para el usuario"""
        private_key_path = f"data/pks/pkv.pem"
        public_key_path = f"data/pks/pku.pem"
        # Verificar si los archivos de claves ya existen
        if os.path.exists(private_key_path) and os.path.exists(public_key_path):
            return
        # Serializamos la clave privada
        pem_private_key = self.private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.BestAvailableEncryption(self.password)
        )
        # Serializamos la clave pública
        pem_public_key = self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        # Guardamos la clave privada en un archivo .pem
        if pem_private_key.splitlines()[0] == b'-----BEGIN ENCRYPTED PRIVATE KEY-----':
            with open(private_key_path, "wb") as archivo:
                archivo.write(pem_private_key)
        # Guardarmos la clave pública en un archivo .pem
        if pem_public_key.splitlines()[0] == b'-----BEGIN PUBLIC KEY-----':
            with open(public_key_path, "wb") as archivo:
                archivo.write(pem_public_key)
        # Prints de depuración
        print(f"Claves guardadas:\n- Privada:\n{pem_private_key}\n- Publica:\n{pem_public_key}")
```

3.2. Firma con clave privada

El siguiente paso tras la creación del par de claves es realizar una firma de datos con la clave privada creada para el sistema. La intención de la aplicación es que cada nota generada por el usuario sea firmada por este, para saber que esas notas son correspondientes a dicho usuario. Esta funcionalidad se implementa en la creación de una nueva nota. Cuando se presiona el botón “Crear Nota”, se realiza este proceso de firmado.

La intención inicial de la firma implementada para esta aplicación correspondía con la idea de compartición de notas entre usuarios, de tal manera que, si existiera otro usuario con el que compartir las notas, este usuario pudiese verificar si esas notas son generadas por el sistema. Sin embargo, debido a la falta de tiempo, esta funcionalidad no se ha podido implementar.

El módulo encargado de manejar todas las funcionalidades de la firma es sign_validation.py, en el cual se crea una clase Sign() que instanciará el usuario que inicie sesión para poder firmar cada una de sus notas y la clave privada para poder firmar los datos.

El método load_private_key() extrae del fichero .pem la clave serializada, y la deserializa usando la contraseña del sistema establecida como variable global. Para firmar, utilizamos el hash SHA-256 y esta clave, convertimos la firma en base 64 y la serializamos como objeto JSON para incluirla en un fichero que almacenará las firmas de cada nota por separado.

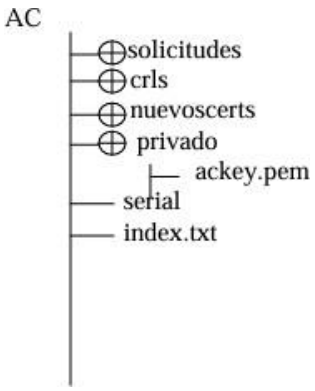
```python
def sign(self, message):
        # Creamos la firma
        signature = self.private_key.sign(
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        # Convertir la firma a Base64 para almacenamiento en JSON
        signature_base64 = base64.b64encode(signature).decode('utf-8')
        note_signature = {"signature": signature_base64}
        # Si no existe el archivo, lo creamos y firmamos
        if not os.path.exists(self.filename):
            # Inicializamos el formato JSON
            data = [note_signature]
        # Si ya existe el archivo, añadimos la nueva firma
        else:
            with open(self.filename, 'r') as file:
                data = json.load(file)
                data.append(note_signature)
        # Guardamos en un archivo JSON
        with open(self.filename, 'w') as file:
            json.dump(data, file, indent=4, sort_keys=True)
        return signature_base64
```

4. Certificados de clave pública

Para el manejo de los certificados se ha utilizado la herramienta OpenSSL para la creación de una entidad raíz de certificación que "autofirme" su certificado, posteriormente realizar una petición de creación y firma de un certificado para el sistema, y, por último, realizar la verificación de la firma a partir de este último certificado.

4.1. Certificado de la entidad raíz en OpenSSL

En primer lugar, la creación de la autoridad de certificación raíz se ha basado en el guión de la práctica de OpenSSL realizado en los laboratorios correspondientes de la asignatura. De este modo, se crea en un entorno aparte (en Linux) una carpeta AC1 que será la autoridad raíz, la cual tendrá la configuración proporcionada en el aula global según el fichero openssl_AC1.cnf, siguiendo este esquema.



Esta entidad tendrá su propia clave privada creada directamente con los comandos de OpenSSL, y también su certificado:

```
openssl req -x509 -newkey rsa:2048 -days 360 -out ac1cert.pem outform PEM -config openssl_AC1.cnf
```

```
openssl x509 -in ac1cert.pem -text -noout
```

Una vez realizada esta parte, tendremos la base para poder realizar una petición a esta entidad y generar un certificado firmado por esta.

4.2. Petición de certificado (CSR)

El módulo csr.py se encarga de la creación de la petición del certificado del sistema. Se ejecuta independientemente de la aplicación, ya que no tiene nada que ver con la funcionalidad del código base. Este módulo copia el método de deserialización de clave privada (mencionado anteriormente en la clase Sign()) a partir del archivo pkv.pem guardado en data/pks. Cuando se ejecuta el módulo, antes de inicializar la aplicación (ejecutar el módulo main.py), comprueba el directorio correspondiente donde almacenar esta petición, crea la petición a partir de unos datos introducidos "a mano", y lo firma con la clave privada del sistema extraída. Estas funcionalidades también son otorgadas por la librería Cryptography, mencionada en muchas otras partes del proyecto.

```python
if __name__ == "__main__":
    # Creamos un directorio para almacenar el CSR
    dir = "data/reqx509"
    if not os.path.exists(dir):
        os.makedirs(dir)
    # Creamos la solicitud de certificado (CSR)
    csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, "ES"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "MADRID"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, "UC3M"),
        x509.NameAttribute(NameOID.COMMON_NAME, "MyDiary.com"),
        x509.NameAttribute(NameOID.EMAIL_ADDRESS, "100500450@uc3m.es"),
    ])
    ).sign(load_private_key(), hashes.SHA256())
    # Guardamos la CSR en un archivo
    with open("data/reqx509/Areq.pem", "wb") as csr_file:
        csr_file.write(csr.public_bytes(serialization.Encoding.PEM))
    # Print de depuración
    print("CSR generado y guardado en 'Areq.pem'")
```

4.3. Certificado de clave pública del sistema

Cuando se ejecuta el script de petición de certificado del sistema se genera una petición con la clave privada del sistema que tendrá que enviarse a la entidad raíz AC1. Esta parte se realiza a mano, es decir, se arrastra el Areq.pem al directorio de solicitudes de AC1 openssl/AC1/solicitudes. Una vez AC1 tiene la petición en OpenSSL se ejecuta el siguiente comando:

openssl ca -in ./solicitudes/Areq.pem -notext -config ./openssl_AC1.cnf

Si todo es correcto, AC1 generará un certificado llamado 01.pem, que se corresponde al número de serie del certificado. El certificado estará firmado por AC1, y como la solicitud se crea con la clave privada del sistema, este certificado contendrá la clave pública utilizada para la verificación de la firma.

En el sistema hasta entonces se almacenaba la clave pública serializada en un archivo pku.pem, pero se sustituye por el certificado creado por AC1, de modo que la carpeta correspondiente a las claves contiene ahora pkv.pem y Acert.pem. La parte correspondiente al almacenamiento de la clave pública se elimina, ya que es innecesario.

Es importante tener en cuenta que. si la clave privada se modifica o se vuelve a generar, se debe volver a crear la solicitud del certificado y el certificado correspondiente al sistema.

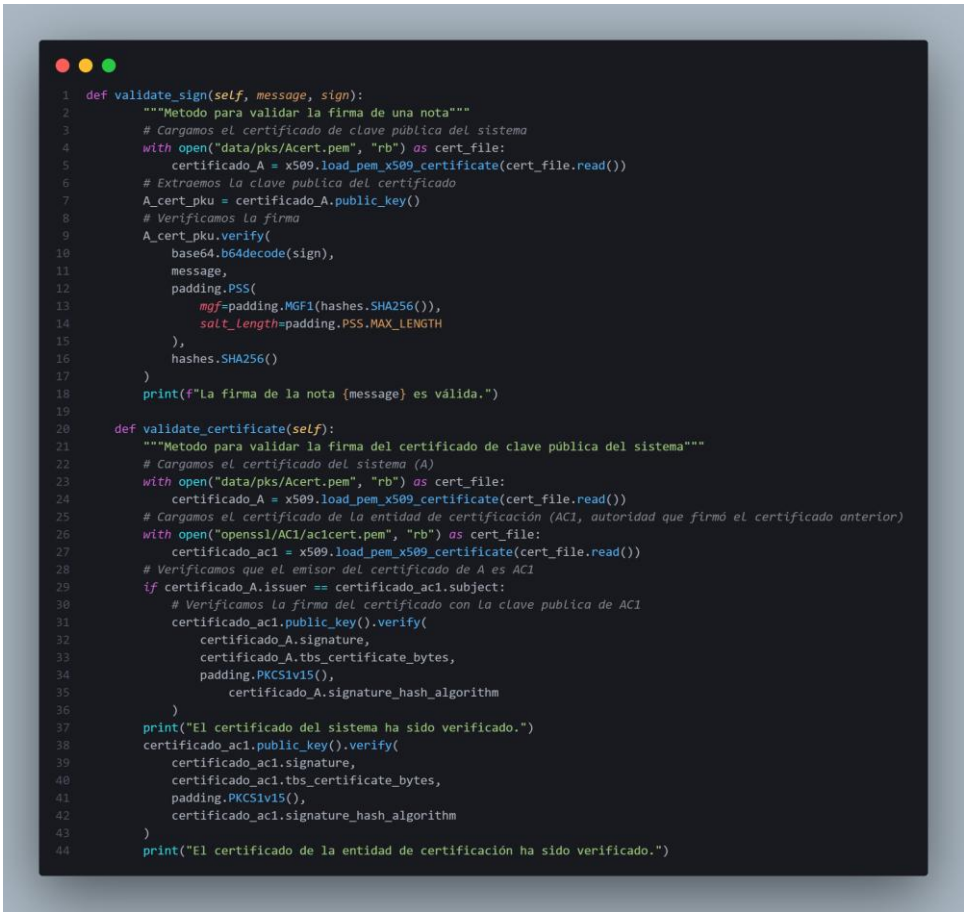
4.4. Autenticación de la firma. Verificación de certificados del sistema y raíz.

Finalmente, la aplicación verifica la firma creada usando el resultado de las partes explicadas antes de llegar a este paso. La verificación se divide en dos métodos establecidos en el módulo sign_validate.py.

En primer lugar, el método validate_sign() valida la firma de una nota inmediatamente después de haberse realizado. Debido a que ya no se tiene la clava pública serializada y almacenada en un archivo .pem independiente, debemos obtener el certificado de clave pública creado en OpenSSL, y extraer la

clave pública que almacena. Una vez tenemos esto, podemos verificar la firma (convirtiéndola primero a base 64), e imprimimos un log de depuración advirtiendo de que todo ha salido bien. En caso de que la firma no sea correcta, antes de llegar a ese log se lanzará una excepción.

Sin embargo, no hemos terminado nuestra tarea, ya que debemos validar también el certificado que incluye la clave pública del sistema. Este se verifica con el método validate_certificate(), incluido a continuación del anterior, el cual obtiene tanto el certificado de clave pública del sistema como el certificado raíz (de la entidad de certificación) que ha firmado dicho certificado. Primeramente, se comprueba si el emisor del certificado de clave pública del sistema es el mismo que el sujeto del certificado de la entidad raíz, y una vez comprobado, se realiza la verificación de la firma. Esta verificación de certificado se realiza inmediatamente después de la verificación de la firma mencionada anteriormente. Una vez hecho esto, hacemos la validación del certificado de la entidad de la misma manera.



```python
def validate_sign(self, message, sign):
        """Metodo para validar la firma de una nota"""
        # Cargamos el certificado de clave pública del sistema
        with open("data/pks/Acert.pem", "rb") as cert_file:
            certificado_A = x509.load_pem_x509_certificate(cert_file.read())
        # Extraemos la clave publica del certificado
        A_cert_pku = certificado_A.public_key()
        # Verificamos la firma
        A_cert_pku.verify(
            base64.b64decode(sign),
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        print(f"La firma de la nota {message} es válida.")

    def validate_certificate(self):
        """Metodo para validar la firma del certificado de clave pública del sistema"""
        # Cargamos el certificado del sistema (A)
        with open("data/pks/Acert.pem", "rb") as cert_file:
            certificado_A = x509.load_pem_x509_certificate(cert_file.read())
        # Cargamos el certificado de la entidad de certificación (AC1, autoridad que firmó el certificado anterior)
        with open("openssl/AC1/ac1cert.pem", "rb") as cert_file:
            certificado_ac1 = x509.load_pem_x509_certificate(cert_file.read())
        # Verificamos que el emisor del certificado de A es AC1
        if certificado_A.issuer == certificado_ac1.subject:
            # Verificamos la firma del certificado con la clave publica de AC1
            certificado_ac1.public_key().verify(
                certificado_A.signature,
                certificado_A.tbs_certificate_bytes,
                padding.PKCS1v15(),
                    certificado_A.signature_hash_algorithm
            )
        print("El certificado del sistema ha sido verificado.")
        certificado_ac1.public_key().verify(
            certificado_ac1.signature,
            certificado_ac1.tbs_certificate_bytes,
            padding.PKCS1v15(),
            certificado_ac1.signature_hash_algorithm
        )
        print("El certificado de la entidad de certificación ha sido verificado.")
```

5. Conclusión

Este proyecto en general ha sido una tarea sencilla debido al amplio conocimiento de Python adquirido durante los anteriores años de la carrera, además del que hemos obtenido acerca del contenido correspondiente de la asignatura en las clases magistrales y reducidas. Además, hemos podido aprender por nuestra parte un caso de uso simple de esta rama de la Ingeniería Informática.

Creemos que, al igual que en el desarrollo de la primera práctica, los resultados obtenidos son correspondientes con lo pedido en el enunciado; y consideramos que el resultado global de la práctica, tanto en contenido como en aprendizaje, ha sido amplio, correcto y gratificante.

6. Anexo

RSA — Cryptography 45.0.0.dev1 documentation. (s. f.).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#generation

RSA — Cryptography 45.0.0.dev1 documentation. (s. f.-b).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#key-serialization

Key Serialization — Cryptography 45.0.0.dev1 documentation. (s. f.).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/

RSA — Cryptography 45.0.0.dev1 documentation. (s. f.-c).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#key-loading

Key Serialization — Cryptography 45.0.0.dev1 documentation. (s. f.-b).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#cryptography.hazmat.primitives.serialization.load_pem_public_key

Key Serialization — Cryptography 45.0.0.dev1 documentation. (s. f.-c).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#cryptography.hazmat.primitives.serialization.load_pem_private_key

RSA — Cryptography 45.0.0.dev1 documentation. (s. f.-d).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#signing

RSA — Cryptography 45.0.0.dev1 documentation. (s. f.-e).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#verification

RSA — Cryptography 45.0.0.dev1 documentation. (s. f.-e).
https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#verification

X.509 CERT Buddy - IntelliJ IDEs Plugin | Marketplace. (s. f.). JetBrains Marketplace.
https://plugins.jetbrains.com/plugin/20571-x-509-cert-buddy

OpenSSL. (s. f.). OpenSSL.

https://openssl.org/

PRÁCTICA: Firma Digital y PKI. OpenSSL