CURSO CRIPTOGRAFÍA Y SEGURIDAD INFORMÁTICA

ENTREGA LAB – EVALUACIÓN 1

GRUPO: 80 Fecha de Entrega: 29/10/2025

Alumnos: Daniel Sampedro Eguía (100499944) / José Luis Palacios (100495679)

Correos: 100499944@alumnos.uc3m.es / 100495679@alumnos.uc3m.es

Link al repositorio: https://github.com/josetheengineer/Cripto-entrega-1.git

Índice

1) Propósito de la aplicación y estructura interna

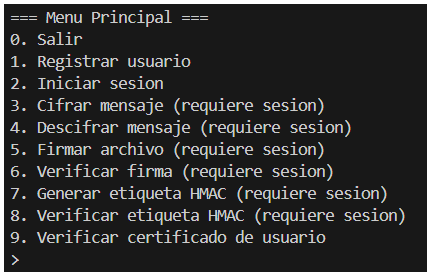
El **propósito** de esta práctica es la aplicación de consola para practicar la criptografía aplicada. Permite el registro y el inicio de sesión de usuarios, cifrar y descifrar mensaje con AES-GCM (AEAD), firmar y verificar archivos con Ed25519, también permite generar y verificar HMAC-SHA256 y por último, generar un PKI local compuesta por: una CA raíz RSA auto firmada, una CA intermedia firmada por la raíz y certificados X.509 de usuario que encapsulan la clave pública Ed25519 del usuario y son firmados por la intermedia. Además, el programa verifica la veracidad de la cadena de confianza Usuario, CA intermedia, CA raíz.

Ahora vamos a hablar de la estructura de las carpetas o archivos principales:

Tenemos un archivo llamado **main.py** que tiene una interfaz de línea de comandos (menú) y orquestación. Por otra parte, tenemos el archivo **crypto_utils.py** que contiene primitivas criptográficas u operaciones criptográficas, están AES, SHA-256 y Ed25519 como operaciones, por otra parte, está las de construcción AES-GCM y HMAC y por último tenemos mecanismos de alto nivel que son los certificados X.509 y la PKI local. El siguiente archivo importante del que vamos a hablar es **user_manager.py**, en este archivo se produce el registro o la autenticación con contraseñas Argon2id persistidas en JSON. Por último, tenemos el archivo **out** que son los artefactos de ejecución y el archivo **tests** de pruebas unitarias pytest.

¿Cómo funciona en el flujo del programa?

1º El usuario se registra guardándose así la contraseña de forma segura, a esta contraseña le aplicamos Argon2id lo que genera un hash (huella digital irreversible). Después guardamos ese hash únicamente en un archivo para que cuando el usuario vuelva a intentar entrar comprobemos que la contraseña genere el mismo hash. 2º Creamos un par de claves para el usuario una privada y una pública, ambas son Ed25519. 3º Emitimos un certificado X.509 para el usuario y tomamos la clave publica del usuario. La CA intermedia firma esa información creando así el certificado X.509. Esto lo hacemos porque el hash Argon2id protege la contraseña, el par de claves permite verificar y firmar archivos, por último, el certificado permite a terceros confiar en esa clave pública. 4º Se inicia sesión: se habilitan operaciones de cifrado, firma y HMAC. 5º Las operaciones guardan los resultados en out y se puede verificar la cadena de certificados.



2) Autenticación de usuarios: algoritmos y gestión de contraseñas

Los algoritmos utilizados dependen de cada función o modulo:

Para la autenticación de contraseñas se utiliza el Argon2id. Para el cifrado de datos utilizamos AES-GCM como cifrado simétrico autenticado. Usamos HMAC-SHA256 para MAC sobre mensajes y archivos. Ed25519 para firmar y verificar archivos. PKI para las CA que firman los certificados X.509 de los usuarios.

Lo siguiente de lo que vamos a hablar es de la autenticación de los usuarios: ¿Qué pasa cuando el usuario se registra?

1º El usuario escribe una contraseña. 2º En user_manager.py llamamos a PasswordHasher.hash(contraseña). 3º Eso devuelve un hash Argon2id que incluye su propia salt y los parámetros de coste. Este hash es irreversible y lo guardaremos en un fichero JSON cuya ruta la configura UserManager y que para cada usuario se guarda su nombre y su password_hash.

En caso de que lo que se realice sea un inicio de sesión entonces, 1º El usuario introduce la contraseña. 2º Cargamos el password_hash del JSON. 3º Llamamos a PasswordHasher.verify(password_hash, contraseña_introducida), que recalcula internamente Argon2id con los mismos parámetros y salt que vienen en el hash almacenado, si coinciden manda un Ok si no coinciden salta una excepción.

Como ya hemos dicho anteriormente en esta práctica utilizamos Argon2id porque es robusto frente a ataque GPU o ASIC, por su coste de memoria y porque es el estándar moderno para el hashing de las contraseñas. Además, evita guardar contraseñas en claro. El cifrado usa claves aleatorias e independiente no se derivan de cifrado a partir de la contraseña.

```
Nombre de usuario: dani
Contrasena:
Confirma la contrasena:
2025-10-27 18:46:47,346 [INFO] Usuario 'dani' registrado correctamente.
Clave privada guardada en: keys\dani_ed25519_private.pem
Clave publica guardada en: keys\dani_ed25519_public.pem
Certificado emitido guardado en: pki\certs\dani_cert.pem
```

```
> 2
Nombre de usuario: dani
Contrasena:
2025-10-27 18:47:00,857 [INFO] Inicio de sesión correcto para 'dani'.
```

```
> 2
Nombre de usuario: dani
Contrasena:
2025-10-27 18:48:33,811 [WARNING] Credenciales inválidas.
```

3) Cifrado simétrico / asimétrico, claves y gestión

¿Cuál es el uso del cifrado simétrico y asimétrico en la app?

En la práctica hemos usado el cifrado simétrico con AES-GCM para proteger los datos de la siguiente manera: se cifra y se descifra con la misma clave secreta, además, el modo GCM añade una etiqueta de integridad que detecta cualquier cambio en el contenido. Al mismo tiempo utilizamos la criptografía asimétrica solo para la autenticidad e identidad, no para cifrar datos: cada usuario contiene un par Ed25519 y la PKI se basa en RSA para las autoridades certificadoras que firman los certificados X.509.

Gestión de claves:

Cada vez que se cifra un mensaje, la aplicación genera una clave AES aleatoria de 256 bits, un nonce único de 12 bytes y produce el cipjertext, ya que, para poder descifrar más tarde se necesita clave + nonce + ciphipertext. La clave se guarda en Base64 en out/key.b64. el nonce se guarda en out/nonce.bin y el ciphipertext en out/ciphipertext.bin. Las claves Ed25519 del usuario se guardan en ficheros keys/…_ed25519_.pem y la PKI mantienen las claves y certificados RSA de la CA raíz y la CA intermedia que se usan para verificar y emitir la cadena de confianza de los certificados X.509

```
Introduce el mensaje o '@ruta' para archivo: mensaje de prueba
2025-10-27 18:57:18,739 [INFO] Mensaje cifrado con AES-GCM.
2025-10-27 18:57:18,739 [INFO] Salida guardada en: out/ciphertext.bin
Clave AES (base64): 117M65ZL/jK6h8Iac/MUkPWzc9NOLUra4RJPPFsgo1M=
Nonce (base64): 2n1ZGv7G/Wxjsjwt
```

```
> 4
Ruta del fichero cifrado (por defecto: out\ciphertext.bin):
Ruta del nonce (por defecto: out\nonce.bin):
Ruta de salida del texto plano (por defecto: out\plain.txt):
Clave AES (base64) (por defecto, leera out\key.b64 si existe):
2025-10-27 19:01:15,447 [ERROR] No se pudo descifrar: autenticación GCM fallida (datos corruptos o clave incorrecta).
```

4) Funciones MAC (HMAC) y cifrado autenticado

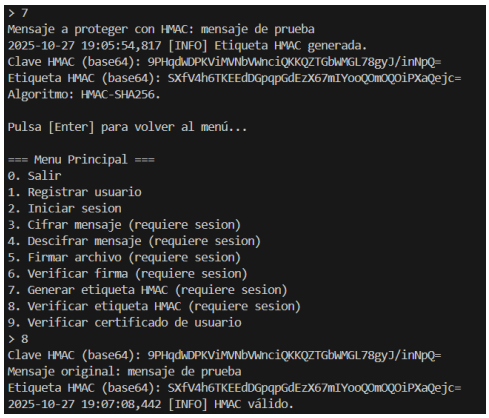
¿Para qué se usa las funciones MAC?

Además de GCM, utilizamos HMAC-256 para etiquetar mensajes o archivos cuando no se desea cifrar.

Para proteger únicamente la integridad del contenido. 1º La aplicación genera una clave HMAC aleatoria de 32 bytes con generate_hmac_key() y usa os.randome(32). 2º Con es clave, calcula la etiqueta HMAC-SHA256 del mensaje utilizando create_hmac(key, mensaje) y la devuelve codificada en Base64 para poder guardarla o enviarla fácilmente. 3º Para comprobar que el mensaje no ha sido modificado se usa verify_hmac(key, mensaje, tag, que recalcula el HMAC

con la misma clave y lo compara en tiempo constante con la etiqueta recibida; si coinciden devuelve True(válido) y, si el mensaje o la clave cambió, aunque sea un byte devolverá False.

En esta práctica usamos HMAC-SHA256 porque es un estándar ampliamente adoptado, robusto frente a colisiones y falsificaciones bajo las suposiciones de SHA-256 y muy sencillo de interoperar entre lenguajes y librerías. Las claves HMAC se generan de forma aleatoria (32 bytes) y pueden tratarse como efímeras o almacenarse si se necesita verificar más tarde. Frente a AEAD que combina confidencialidad e integridad en una sola operación y reduce errores de uso cuando se quiere cifrar, HMAC es preferible ya que cuando solo necesitamos integridad o autenticidad sin cifrar el contenido, o cuando el esquema no es AEAD y se desea añadir verificación aparte.



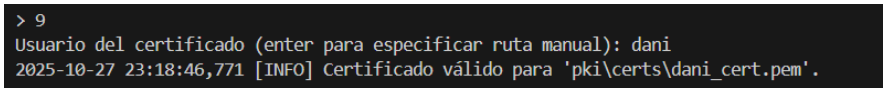
5) Firmas digitales y PKI (certificados X.509)

Estas son las firmas del usuario. La aplicación genera un par de Ed25519 por usuario y permite firmar y verificar los archivos.

En la práctica se realiza una PKI local que está formada por una CA raíz y una CA intermedia ambas con RSA 4096 y SHA-256. La Sub-CA es la que emite los certificados X.509 de los usuarios en los que incluye la clave pública Ed25519 de cada usuario. Esto lo hace para poder verificar firmas. La confianza se comprueba validando la cadena de certificación usuario, Sub-CA, Raíz, si cada certificado está correctamente firmado por el superior, entonces el certificado es valido y confiable dentro de esta PKI

¿Por qué lo hemos hecho así?

RSA sigue siendo bastante común para la autoridad certificadora por su gran y amplio soporte en X.509/TLS. Ed25519 permite firmas rápidas segura y con claves cortas para los usuarios. Separar la CA raíz de la CA subordinada permite revocar o rotar sin tocar la raíz.



6) Pruebas realizadas para garantizar la calidad del código

El Proyecto contiene pruebas unitarias con pytest (en tests/test_crypto_utils.py). Estas pruebas unitarias validan los módulos clave de seguridad. Primero, se verifica el cifrado autentico con AES_GCM realizando un ciclo completo de cifrar y descifrar para comprobar que el texto recuperado es idéntico al original y que cualquier alteración del ciphertext provoca un error de autenticación. Segundo, se hace una prueba para comprobar el sistema de firmas digitales con Ed25519 generando un par de claves, firmando un archivo y validando que la verificación con la clave pública es correcta, mientras que con un contenido modificado con otra clave falla como es debido. Tercero, se comprueba la integridad mediante HMAC-SHA256, creando la etiqueta para un mensaje y verificando que varify_hmac devuelve True o verdadero solo cuando se usan la misma clave y mensaje. Cuarto y última prueba, se ensaya con la PKI local construyendo una CA raíz y Sub-CA y un certificado de usuario, y se verifica la cadena de confianza que ya hemos mencionado anteriormente en esta práctica, asegurando que la cadena válida se acepta y que cualquier manipulación se rechaza. Estas pruebas proporcionan confianza en la correctitud y el comportamiento seguro de las funciones de cifrado firmas, MAC y certificados.