MEMORIA CRIPTOGRAFÍA

AUTORES:
Guillermo Sánchez-Pérez García-Marcos 100522341
100522341@alumnos.uc3m.es
Mohamed Rida Chahdaoui Moujib 100522245
100522202@alumnos.uc3m.es

GRUPO: 82

ÍNDICE

Guillermo Sánchez-Pérez García-Marcos 100522341
Mohamed Rida Chahdaoui Moujib 100522202

Introducción - propósito de la aplicación y estructura

Nuestro trabajo simula una aplicación cliente-servidor, donde tenemos un hotel que gestiona varias reservas hechas por diferentes usuarios.

Además, existe un usuario administrador con permisos adicionales para consultar reservas de cualquier usuario. Este se crea con la primera ejecución del programa, cuando la base de datos de usuarios (*usuarios.json)* está vacía. Y una interfaz que permite realizar todas las operaciones de manera sencilla para el usuario.

De esta forma, tenemos diferentes módulos donde implementamos las funcionalidades necesarias para el programa, con un módulo *main.py* que ejecuta todo el programa. Aunque en realidad es el módulo *ui.py* el que implementa la interfaz y contiene las llamadas a las funciones, que se encuentran en los módulos *booking_management.py, hash_functions.py, key_management.py, user_management.py* y *login.py*.

Tenemos dos bases de datos que se encuentran en el directorio *database*: *reservas.json* y *usuarios.json*. En el caso de las reservas, las guardamos cifradas, pero en el caso de los usuarios hemos prescindido de cifrarlos porque no necesitan recuperar su información en ningún momento de la ejecución del programa. No obstante, esta funcionalidad habría añadido una capa de seguridad extra que hemos omitido por mantener la simplicidad, aunque las contraseñas de dichos usuarios sí que se guardan hasheadas.

Por último, las claves de cada usuario (pública y privada) se guardan por separado en archivos .pem, dentro del directorio *claves*. Por ejemplo, para el usuario admin, tenemos los archivos *admin_private.pem* y *admin_public.pem.*

Autenticación de usuarios - algoritmos utilizados

La autenticación de nuestra aplicación se basa en **contraseñas**, una información conocida por el usuario. Estas contraseñas nunca se almacenan en texto claro, sino que son **hasheadas con Argon2id**, una KDF (*key derivation function)* moderna resistente a ataques de rainbow tables y de fuerza bruta. Lo implementamos en la siguiente función:

```python
def hash_text(password: str) -> str:
    salt = os.urandom(16)

    kdf = Argon2id(
        salt=salt,
        length=32,
        iterations=2,
        lanes=1,
        memory_cost=32 * 1024,
        ad=None,
        secret=None
    )

    key = kdf.derive(password.encode())

    # Guardamos sal + clave derivada en base64
    return base64.b64encode(salt + key).decode("utf-8")
```

Justificamos los valores de los parámetros a continuación:

- Salt de 16 bytes: creemos que es una longitud suficiente para garantizar que el hash sea único para cada usuario sin hacer que ocupe demasiado almacenamiento.
- Length de 32 bytes: de la misma forma que con el salt, creemos que es un tamaño equilibrado entre seguridad y almacenamiento. De forma que aseguremos protección pero también una ejecución rápida.
- Iterations = 2: escogimos este valor para garantizar la fluidez del programa, aunque podríamos utilizar un número mayor para más seguridad.
- Lanes = 1: podría aumentarse para explotar varios núcleos del sistema a la vez.
- Memory_cost = 32 * 1024 = 32 MiB: consideramos que es suficiente para penalizar ataques masivos.
- Ad y sercret se dejan a none porque complicarían mucho la implementación de nuestro trabajo y no son necesarios.

En resumen, hemos buscado tener cierta seguridad pero manteniendo la fluidez (aunque en nuestro caso, la aplicación es tan pequeña que podríamos haber aumentado los parámetros, pero hemos imaginado una aplicación con mucho más tráfico). Como apunte, mencionar que devolvemos en base64 porque los JSON esperan texto en UTF-8, no bytes.

Habiendo explicado esto, podemos pasar por fin a explicar la autenticación de usuarios, que se hace en el módulo *login.py*, concretamente en la función *login_usuario*:

```python
def login_usuario(usuario_name:str, password):
    """Método que guarda un usuario. Recibe su usuario y su contraseña para
    comprobar que existe y si existe, compara la contraseña recibida con la
    almacenada (que está hasheada)."""
    usuarios = cargar_usuarios()

    if usuario_name not in usuarios:
        return False, "Usuario no existe"

    hash_guardado = usuarios[usuario_name]["password_hash"]

    if not hash_functions.verify_hash(password, hash_guardado):
        return False, "Contraseña incorrecta"

    rol = usuarios[usuario_name]["rol"]
    return True, f"Bienvenido, {usuario_name}. Rol: {rol}"
```

El proceso de autenticación de un usuario consiste por tanto en:

1. Verificar si el usuario está en la base de datos (*cargar_usuarios()* simplemente devuelve todos los objetos del json) y recuperar el usuario junto a la contraseña hasheada.
2. Verificar si la contraseña hasheada coincide con el hash de la contraseña recibida por la función *login_usuario* (que es precisamente la contraseña que ingresa el usuario al iniciar sesión. Cosa que se hace mediante la función *verify_hash*:

```python
def verify_hash(text: str, encoded: str) -> bool:
    """
    salt = b""
    stored_key = b""
    try:
        data = base64.b64decode(encoded.encode("utf-8"))
        salt = data[:16]
        stored_key = data[16:]

        kdf = Argon2id(
            salt=salt,
            length=32,
            iterations=2,
            lanes=1,
            memory_cost=32 * 1024,
            ad=None,
            secret=None
        )

        # Esta línea lanza InvalidKey si la verificación falla
        kdf.verify(text.encode(), stored_key)
        print(
            f"[DEBUG] hash_functions: Argon2id verificación correcta (memoria=32MB,
            iteraciones=2, salida={len(stored_key)} bytes, salt={len(salt)} bytes)."
        )
        return True
    except InvalidKey:
        print("[WARNING] hash_functions: Verificación Argon2id fallida (entrada no coincide).")
        return False
```

Mientras que hash_text sirve para crear y almacenar de forma segura el hash de la contraseña cuando el usuario se registra, verify_hash hace la operación necesaria para comparar con el hash generado: toma la contraseña introducida en el login, recalcula el hash utilizando la misma salt almacenada y comprueba si ambos coinciden, permitiendo autenticar al usuario sin necesidad de conocer ni almacenar su contraseña real.

Cifrado simétrico/asimétrico

Nuestra aplicación emplea un **sistema híbrido de cifrado,** ya que combinamos tanto el cifrado simétrico (AES-GCM) como el asimétrico (RSA).

Para las reservas, cumpliendo con nuestro objetivo de mantener la rapidez de la ejecución, utilizamos AES-GCM, ya que pueden ser bastante pesadas si llegamos a tener muchas. Además, añadimos el atributo *associated_data*, garantizando así la **autenticidad del mensaje** sin usar HMAC. El papel de RSA consiste en proteger la clave AES, asegurando que solo los usuarios autorizados (el admin y el propio usuario) puedan descifrar la información.

El flujo completo del cifrado híbrido se realiza en la función *cifrar_reserva()*, dentro de la clase *Booking* (reserva) en *booking_management.py:*

```python
def cifrar_reserva(self) -> dict:
    datos_completos = [self.usuario_asociado, self.fecha_asociada, self.datos]

    # Además, guardaremos el usuario hasheado para que luego sea más fácil buscar sus reservas
    usuario_hasheado = hash_functions.hash_text(self.usuario_asociado)

    # Codificamos los datos como bytes para que AES pueda usarlos
    datos_byte = json.dumps(datos_completos).encode()
    usuario_bytes = self.usuario_asociado.encode()

    #Generamos la clave AES aleatoria
    aes_clave = AESGCM.generate_key(bit_length=256) #32 bytes

    #Creamos un objeto AESGCM con la clave anterior
    aesgcm = AESGCM(aes_clave)

    #Generamos un nonce de 12 bytes (semilla única para cada cifrado)
    nonce = os.urandom(12)
    print(f"[DEBUG] booking_management: -- Inicio cifrado reserva para {self.usuario_asociado} --")
    print(
        f"[DEBUG] booking_management: Clave AES-GCM de 256 bits generada (len={len(aes_clave)} bytes) y nonce de {len(nonce)} bytes para {len
        (datos_byte)} bytes de datos."
    )

    #Ciframos los datos de la reserva con AES-GCM: solo será válida para el usuario asociado: AUTENTICACIÓN
    reserva_cifrada = aesgcm.encrypt(nonce, datos_byte, associated_data=usuario_bytes)
    print(
        f"[DEBUG] booking_management: AES-GCM completado -> ciphertext={len(reserva_cifrada)} bytes (incluye etiqueta de 128 bits)."
    )
```

```python
#Ahora debemos cifrar esta clave AES con la clave pública

#Cargamos la clave pública desde su archivo.pem usando key_management
clave_publica = key_management.cargar_clave_publica(self.usuario_asociado)
aes_clave_cifrada = key_management.rsa_oaep_encrypt(clave_publica, aes_clave)
print(
    f"[DEBUG] booking_management: Clave AES protegida con RSA-OAEP 2048 bits para {self.usuario_asociado}."
)

# Cargamos también la clave pública del administrador
clave_publica_admin = key_management.cargar_clave_publica("admin")
aes_clave_cifrada_admin = key_management.rsa_oaep_encrypt(clave_publica_admin, aes_clave)
print("[DEBUG] booking_management: Clave AES duplicada para admin usando RSA-OAEP 2048 bits.")

# Ciframos también el nombre del usuario con la clave pública del admin para mantener la privacidad
usuario_cifrado_admin = key_management.rsa_oaep_encrypt(
    clave_publica_admin,
    self.usuario_asociado.encode()
)
print("[DEBUG] booking_management: Usuario asociado cifrado para admin con RSA-OAEP 2048 bits.")
print("[DEBUG] booking_management: -- Fin cifrado reserva --")

#Devolvemos una tupla con los datos cifrados y lo necesario para descifrarlos
#Guardamos el nonce, los datos y las claves cifradas

return {"usuario_hasheado": usuario_hasheado,"reserva_cifrada":reserva_cifrada,
        "aes_clave_cifrada": aes_clave_cifrada, "aes_clave_cifrada_admin": aes_clave_cifrada_admin,
        "nonce": nonce, "usuario_original_cifrado": usuario_cifrado_admin}
```

Flujo:

1. Se prepara la información de la reserva (email, teléfono, dni, fecha) y se pasa a bytes para poder cifrarla con AES.
2. Se genera una clave AES aleatoria de 256 bits y un nonce aleatorio de 12 bytes, que evita que se puedan encontrar patrones.
3. Los datos se cifran con **AES-GCM** y usando associated_data como ya explicamos.
4. Después, ciframos la clave AES tanto con la clave del usuario dueño como con la del admin utilizando RSA, todo ello para que solo ellos puedan descifrar la reserva y la clave simétrica nunca se almacene ni se transmita en claro.
5. Se cifra también el identificador del usuario con la clave pública del administrador para proteger su privacidad incluso a nivel de metadatos, de manera que un tercero con acceso a las reservas no pueda inferir a quién pertenece cada reserva únicamente observando los campos no cifrados.
6. Se devuelve un diccionario con todos los datos que se almacenarán en reservas.json, algunos como la clave aes cifrada son necesarios para descifrar.

Para descifrar, hacemos el proceso inverso en *descifrar_reserva*: decode en vez de encode, decrypt en vez de encrypt. Estas son decrypt y encrypt, asociadas a **RSA**:

```python
def rsa_oaep_encrypt(clave_publica, datos: bytes) -> bytes:
    """Cifra datos con RSA-OAEP usando SHA-256."""
    ciphertext = clave_publica.encrypt(
        datos,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return ciphertext


def rsa_oaep_decrypt(clave_privada, datos_cifrados: bytes) -> bytes:
    """Descifra datos con RSA-OAEP usando SHA-256."""
    plaintext = clave_privada.decrypt(
        datos_cifrados,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return plaintext
```

Autenticación de mensajes

Utilizamos AES-GCM como cifrado autenticado, por lo que el código de autenticación de mensajes (MAC) se genera en el propio algoritmo en forma de un *authentication tag* (dentro de *cifrar_reserva*, cuando llamamos a encrypt). Esto nos permite garantizar la autenticidad de los datos cifrados, detectando cualquier modificación durante el descifrado sin necesidad de usar mecanismos adicionales como HMAC.

Además, hacemos uso del parámetro associated_data, de forma que los datos quedan vinculados criptográficamente al usuario propietario y un cambio en dicho campo invalida la autenticación. Por último, la clave AES empleada en el proceso se protege mediante cifrado asimétrico RSA con las claves públicas del usuario y del administrador, lo que asegura que solo los dos autorizados puedan descifrar la reserva sin que la clave simétrica se almacene nunca en claro.

Pruebas realizadas

1. REGISTRO CORRECTO: el usuario debe completar todos los campos, una vez registrado, se almacena en la base de datos correctamente.

2. INICIO DE SESIÓN CORRECTO: solo se inicia sesión si la contraseña introducida coincide con la de registro (que se comprueba con funciones hash como ya hemos explicado anteriormente).

3. CREACIÓN DE RESERVAS Y VISUALIZACIÓN DE LAS MISMAS: el usuario puede efectivamente crear reservas y acceder a todas sus reservas.

4. RESERVAS ALMACENADAS CORRECTAMENTE: aparecen los datos cifrados esperados.

5. USUARIOS ALMACENADOS CORRECTAMENTE: aparece hasheada la contraseña en usuarios.json.

6. CLAVES ALMACENADAS CORRECTAMENTE: aparece la clave privada encriptada en el .pem y la clave pública sin encriptar en otro .pem. Ambos asociados al usuario.

7. PRUEBAS USUARIO ADMIN: hemos comprobado que efectivamente solo se crea en la primera ejecución (database vacía, o en su defecto sin usuario admin, pero eso no ocurriría en un caso real), y que puede ver las reservas de TODOS los usuarios.