Criptografía y seguridad informática

Entregable 1: Sistema de votación online

Grupo 84 - id grupo de prácticas 12

Repositorio de la práctica:

(https://github.com/UC3MLP/criptoprojecto)

Hecho por :

- 100522167@alumnos.uc3m.es - Laura Parrilla Ruiz
- 100522170@alumnos.uc3m.es - Julio Ruiz Sánchez

ÍNDICE:

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

El propósito de la aplicación es crear un sistema de votación online seguro que simule una situación de democracia directa. Este sistema permite a los ciudadanos votar por leyes en línea, asegurando:

1. Elegibilidad: solo permite votar una vez.
2. Confidencialidad: el voto es secreto.
3. Integridad: el voto no puede ser alterado o negado.
4. Autenticación: solo votan usuarios autenticados.

La aplicación está estructurada en 5 módulos lógicos con una interfaz gráfica en python/Tkinter:

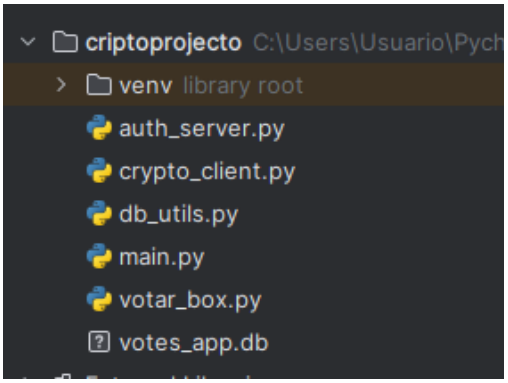
1. *main.py*: proporciona la interfaz de usuario para el registro, login y votación.

2. *crypto_client.py*: Prepara el paquete con el voto que se enviará a la urna (*BallotBox-BB*). Se encarga de la criptografía
    a. Simétrica: Cifrado del voto, generación de clave de sesión.
    b. Asimétrica: Envuelve clave de sesión con clave pública de *BB*.

3. *auth_server.py*: Funciona como un servidor de autenticación y elegibilidad. Emite un token único por elección, firmado con una clave secreta(Kissue con HMAC), para que el ciudadano pueda votar de forma anónima.

4. *votar_ box.py (con BB-BallotBox)*: Realiza la función de una Urna Electrónica; servidor de recepción de votos. Verifica la autenticidad del token y, si es válido, guarda el voto cifrado en la base de datos (tallies) sin identidad para el conteo futuro.

5. *db_utils.py*: Crea la base de datos junto con las tablas necesarias para guardar la información de los usuarios, votos, etc.



b. ¿Cómo se realiza la autenticación de usuarios? ¿Qué algoritmos ha utilizado y por qué? Detalle cómo se gestionan las contraseñas de los usuarios y si se generan claves a partir de éstas.

La autenticación de usuarios se realiza mediante:

a. *Registro*:
   i. Valida el formato del mail (...@gmail.com) y DNI (DNI español).
   ii. Genera **salt** (128 bits) y deriva **pwd_hash** con **PBKDF2-HMAC-SHA-256**.
   iii. Guarda en **users**(email, dni, salt, pwd_hash, iterations).
b. *Login*:
   i. Recupera salt y iterations y re-deriva con **PBKDF2-HMAC-SHA-256**.
   ii. Compara contraseñas con **hmac.compare_digest(a, b)**.

Los algoritmos de autenticación usados son:

1. *PBKDF2-HMAC-SHA-256*: Fortalece contraseñas de forma segura. Evita ataques de fuerza bruta.
2. *salt* por usuario: Evita repeticiones en tablas.
3. hmac.compare_digest(a,b): Reduce vulnerabilidades a ataques de tiempo.

No se genera ninguna clave criptográfica a partir de las contraseñas. El hash derivado solo se utiliza para verificar el login.

```python
2 usages  julio +1
def login_user(email,password):
    """comprobar que tienen la misma contraseña"""
    con = sqlite3.connect(DB_PATH)
    cur = con.cursor()
    row = cur.execute( _sql: """
        SELECT salt, pwd_hash, iterations , dni FROM users WHERE email=?
        """,  _parameters: (email,)).fetchone()
    con.close()
    # fetch la salt, pwd_hash y iterations
    if not row:
        # si no existe, nada
        raise ValueError("Email  no encontrado o incorrecto")

    salt, stored_hash, iterations,dni = row  # la guardo
    test = derive_pwd_hash(password, salt, iterations)
    if hmac.compare_digest(test, stored_hash):
        return True,dni #Éxito, devolvemos true y dni
    else:
        raise ValueError(" Contraseña incorecta")

    # si coinciden ? contraseña correcta y devuelve el dni asociado al correo
```

```python
2 usages  UC3ML +1
def derive_pwd_hash(password, salt, iterations=200_000):
    """transforma contraseña en hash de 32 bytes con PBKDF2"""
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,                          # tamaño normal
        salt=salt,
        iterations=200_000,
    )
    derive = kdf.derive(password.encode()) # pasando a bytes + derivando
    return derive


2 usages  UC3ML +1
def register_user(email, dni, password):
    """registro de usuario"""
    if not password:
        raise ValueError("escribe una contraseña válida")
    if not re.match( pattern: r"^[A-Za-z0-9._%+-]+@gmail\.com$", email):
        # comprobar email - COMPROBACIÓN PUEDE QUE CAMBIE
        raise ValueError("Sólo se admite @gmail.com")
    if not re.match( pattern: r"^[0-9A-Z]{7,10}$", dni):
        # comprobar dni
        raise ValueError("DNI inválido.")
    salt = os.urandom(16)  # 128 bits aleatorios
    iterations = 200_000
    pwd_hash = derive_pwd_hash(password, salt, iterations)
    con = sqlite3.connect(DB_PATH)
    cur = con.cursor()
    cur.execute( _sql: """
        INSERT INTO users(email, dni, salt, pwd_hash, iterations)
        VALUES (?,?,?,?,?)""",
                 _parameters: (email, dni, salt, pwd_hash, iterations))
    con.commit()
    con.close()
```

c. ¿Para qué utiliza el cifrado simétrico/ asimétrico o ambos? ¿Qué algoritmos ha utilizado y por qué? ¿Cómo gestiona las claves? Explique los mismos aspectos si se utiliza cifrado asimétrico para este tipo de cifrado.

Se utiliza un cifrado híbrido para cifrar los votos:

➔ *Simétrico*, para cifrar el contenido del voto con **AES-GCM**.

➔ *Asimétrico*, para envolver la clave **AES** con la clave pública de *BB* usando **RSA-OAEP**.

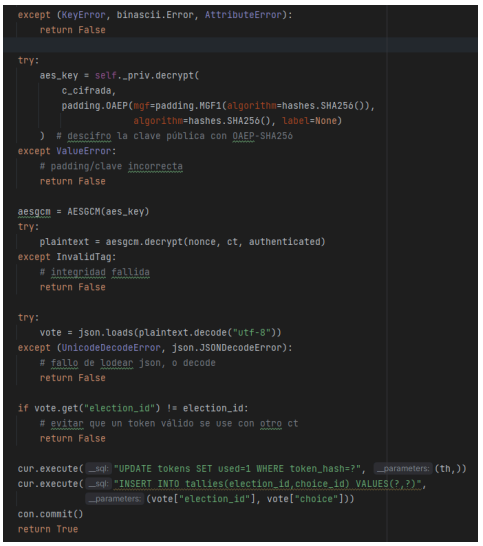
Los algoritmos utilizados son:

1. *AES-GCM*: Da confidencialidad e integridad.
2. *RSA-OAEP*: Envoltura segura de clave simétrica.

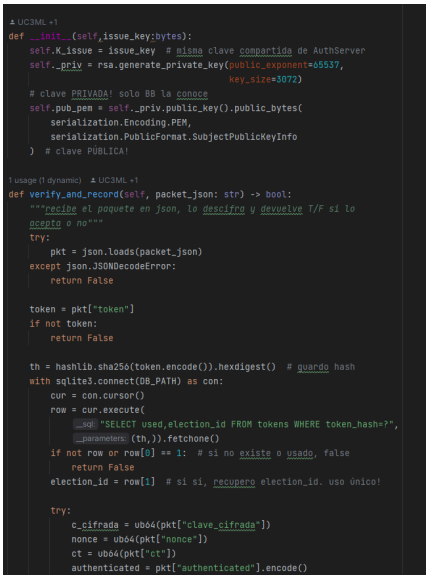
Las gestión de las claves generadas es:

- *Clave AES*: Es generada aleatoriamente por el cliente por voto.
- *Claves RSA*:
    - **BB** genera una clave privada que se queda en el servidor y una pública la cual publica al cliente.
    - Cliente usa la pública para cifrar la clave **AES**, **BB** usa la privada para abrirla.
- *Claves de usuario*: No se crean por usuario.

```
2 usages  UC3ML +1
class ClientCrypto:
    """clase que se encarga de cifrar el paquete para ser enviado a
    votar_box y descifrado"""
    UC3ML
    def __init__(self, bb_pub_pem: bytes):
        # clave pública de ballotbox (NUNCA PRIVADA!)
        self.pub = serialization.load_pem_public_key(bb_pub_pem)

    1 usage  UC3ML +1
    def make_packet(self, election_id: str, choice: str, token: str) -> str:
        """preparar paquete de voto para enviar a ballotbox"""
        aes_key = AESGCM.generate_key(bit_length=256)  # AES-256 aleatoria
        aesgcm = AESGCM(aes_key)
        nonce = secrets.token_bytes(12)  # 96 bits - único por cada clave
        authenticated = f"vote:{election_id}".encode()  # atar cifrado a elección
        payload = json.dumps({  # lo que si se cifra
            "election_id": election_id,
            "choice": choice,
            "client_nonce": b64(secrets.token_bytes(16))
        }).encode()
        ct = aesgcm.encrypt(nonce, payload, authenticated)
        # GCM:conf+integridad
        # ct = ciphertext+tag

        # RSA-OAEP
        c_cifrada = self.pub.encrypt(aes_key,
                                     padding.OAEP(
                                         mgf=padding.MGF1(algorithm=hashes.SHA256()),
                                         algorithm=hashes.SHA256(), label=None))
        # json
        pkt = {"clave_cifrada": b64(c_cifrada),  # clave simétrica con RSA-OAEP
               "nonce": b64(nonce),  # nonce de AES-GCM
               "ct": b64(ct),  # ciphertext || tag
               "authenticated": authenticated.decode(),
               # autenticado ("vote:<id>")
               "token": token}  # token de 1 solo uso
        return json.dumps(pkt)
```



```
except (KeyError, binascii.Error, AttributeError):
    return False

try:
    aes_key = self._priv.decrypt(
        c_cifrada,
        padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
                     algorithm=hashes.SHA256(), label=None)
    )  # descifro la clave pública con OAEP-SHA256
except ValueError:
    # padding/clave incorrecta
    return False

aesgcm = AESGCM(aes_key)
try:
    plaintext = aesgcm.decrypt(nonce, ct, authenticated)
except InvalidTag:
    # integridad fallida
    return False

try:
    vote = json.loads(plaintext.decode("utf-8"))
except (UnicodeDecodeError, json.JSONDecodeError):
    # fallo de lodear json, o decode
    return False

if vote.get("election_id") != election_id:
    # evitar que un token válido se use con otro ct
    return False

cur.execute( __sql: "UPDATE tokens SET used=1 WHERE token_hash=?",  __parameters: (th,))
cur.execute( __sql: "INSERT INTO tallies(election_id,choice_id) VALUES(?,?)",
             __parameters: (vote["election_id"], vote["choice"]))
con.commit()
return True
```



```
UC3ML +1
def __init__(self,issue_key:bytes):
    self.K_issue = issue_key  # misma clave compartida de AuthServer
    self._priv = rsa.generate_private_key(public_exponent=65537,
                                          key_size=3072)
    # clave PRIVADA! solo BB la conoce
    self.pub_pem = self._priv.public_key().public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )  # clave PUBLICA!

1 usage (1 dynamic)  UC3ML +1
def verify_and_record(self, packet_json: str) -> bool:
    """recibe el paquete en json, lo descifra y devuelve T/F si lo
    acepta o no"""
    try:
        pkt = json.loads(packet_json)
    except json.JSONDecodeError:
        return False

    token = pkt["token"]
    if not token:
        return False

    th = hashlib.sha256(token.encode()).hexdigest()  # guardo hash
    with sqlite3.connect(DB_PATH) as con:
        cur = con.cursor()
        row = cur.execute(
            __sql: "SELECT used,election_id FROM tokens WHERE token_hash=?",
            __parameters: (th,)).fetchone()
        if not row or row[0] == 1:  # si no existe o usado, false
            return False
        election_id = row[1]  # si sí, recupero election_id. uso único!

        try:
            c_cifrada = ub64(pkt["clave_cifrada"])
            nonce = ub64(pkt["nonce"])
            ct = ub64(pkt["ct"])
            authenticated = pkt["authenticated"].encode()
```

d. ¿Para qué utiliza las funciones de códigos de autenticación de mensajes (MAC)? ¿Qué algoritmos ha utilizado y por qué? ¿Cómo gestiona la clave/s? Explicite si utiliza algoritmos de cifrado autenticado y las ventajas que esto ofrece.

Las funciones MAC se utilizan en nuestro programa para tener **tokens de elegibilidad**, siendo *AuthServer* quien emite un token **HMAC-SHA-256** sobre (dni | election_id | nonce | ts). Luego, el DB guarda el **hash del token** y **used** para uso único.

Los algoritmos utilizados son:

➔ *HMAC-SHA-256*: Clave compartida K_issue.

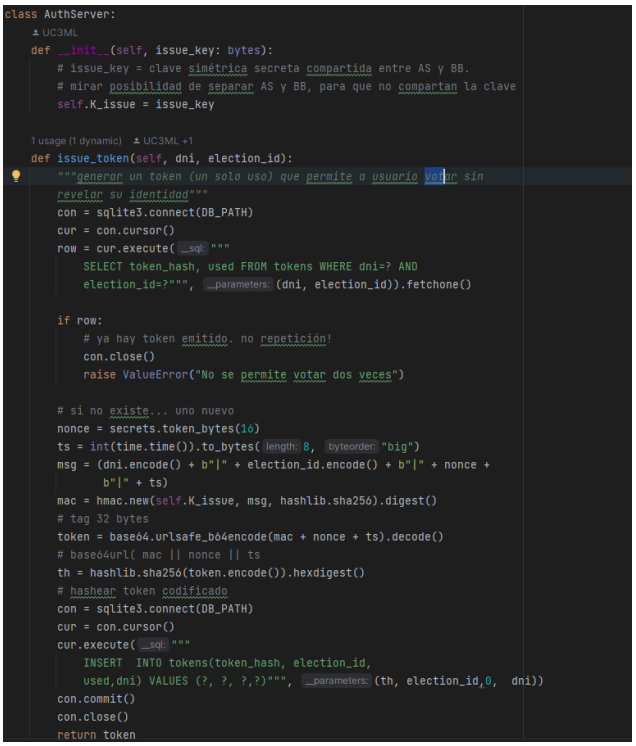
➔ *AES-GMC*: Confidencialidad e integridad. Aquí, ya es cifrado autenticado (**AEAD**).

La gestión de claves es:

- *K_issue* (256 bits): clave secreta de AuthServer. **BB** también la conoce.

Sí, se utilizan algoritmos de cifrado autenticado (**AEAD**). Sus ventajas son:

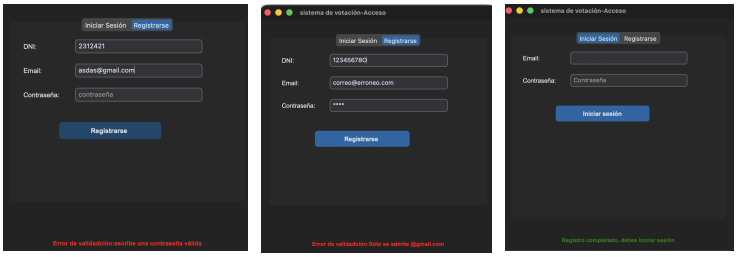
1. Unifica *confidencialidad* + *integridad* sin **MAC** adicional.
2. Se añade *AAD* (Additional Authenticated Data - llamado *authenticated* en el código) para atar criptográficamente el mensaje a la elección y evitar la reutilización del voto entre elecciones.



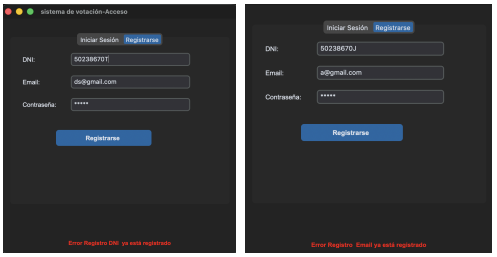
```
class AuthServer:
    UC3ML
    def __init__(self, issue_key: bytes):
        # issue_key = clave simétrica secreta compartida entre AS y BB.
        # mirar posibilidad de separar AS y BB, para que no compartan la clave
        self.K_issue = issue_key

    1 usage (1 dynamic)  UC3ML +1
    def issue_token(self, dni, election_id):
        """generar un token (un solo uso) que permite a usuario votar sin
        revelar su identidad"""
        con = sqlite3.connect(DB_PATH)
        cur = con.cursor()
        row = cur.execute( __sql: """
            SELECT token_hash, used FROM tokens WHERE dni=? AND
            election_id=?""",  __parameters: (dni, election_id)).fetchone()

        if row:
            # ya hay token emitido. no repetición!
            con.close()
            raise ValueError("No se permite votar dos veces")

        # si no existe... uno nuevo
        nonce = secrets.token_bytes(16)
        ts = int(time.time()).to_bytes( length: 8,  byteorder: "big")
        msg = (dni.encode() + b"|" + election_id.encode() + b"|" + nonce +
               b"|" + ts)
        mac = hmac.new(self.K_issue, msg, hashlib.sha256).digest()
        # tag 32 bytes
        token = base64.urlsafe_b64encode(mac + nonce + ts).decode()
        # base64url( mac || nonce || ts
        th = hashlib.sha256(token.encode()).hexdigest()
        # hashear token codificado
        con = sqlite3.connect(DB_PATH)
        cur = con.cursor()
        cur.execute( __sql: """
            INSERT  INTO tokens(token_hash, election_id,
            used,dni) VALUES (?, ?, ?,?)""",  __parameters: (th, election_id,0,  dni))
        con.commit()
        con.close()
        return token
```

e. Indique las pruebas realizadas para garantizar la calidad del código

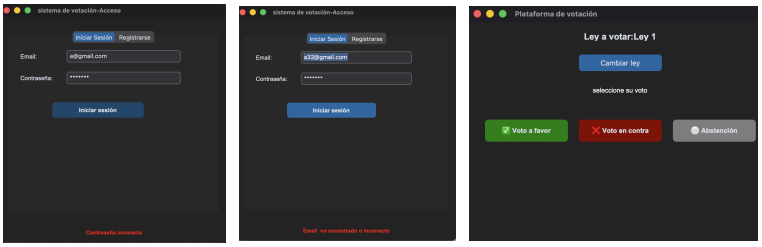
1. Al registrarse, si no pones contraseña o email correcto (mail@gmail.com), o un dni real de 9 números y una letra, saltará un error y no te dejará registrarse. Por el contrario, si haces todo correctamente sí lo hará.



2. Si ya hay un email o dni registrado saltará un error.



3. Si inicias sesión con un mail no existente o contraseña incorrecta saltará un error. En cambio, si escribes un mail existente y contraseña correcta te envía a la pestaña de votos, donde puedes elegir varias leyes y varias opciones de voto.



4. En la zona de voto, si ya has votado no te dejará votar ninguna opción y saltará un error para avisarte, sin embargo, si eliges otra ley podrás votar ya que no has gastado tu voto para esa ley.

f. Anexo: README.md

Esto es una pequeña sección que copia lo escrito en el archivo *markdown* **README.md**, con las pautas de instalación para que el programa funcione correctamente:

Sistema de Votación Online Seguro

Este proyecto implementa un sistema de votación electrónica utilizando criptografía. La interfaz gráfica ha sido desarrollada con *CustomTkinter*.

Requisitos

Para ejecutar esta aplicación, necesitarás tener instalado:

1. *Python* (cualquier versión superior o igual a la 3.8).
2. Las dos librerías listadas a continuación.

Librerías Necesarias

Las dos librerías externas son

A. *CustomTkinter*: para la interfaz.
B. *Cryptography*: para los algoritmos criptográficos como AES-GCM, RSA-OAEP, PBKDF2 y HMAC.

En caso de no tener **python 3.8 o superior** instalado, en el terminal, ejecuta el siguiente comando:

```bash
pip install python3.14
```

O ve a la página principal de python **python.org/downloads** y sigue las instrucciones para instalarlo dependiendo de tu sistema operativo. Para verificar la instalación puedes usar este comando en la terminal:

```bash
pip python3 --version
```

Teniendo instalado python pondremos los siguientes comandos también en el terminal para instalar *CustomTkinter* y *Cryptography.*

➔ *CustomTkinter*:

```bash
pip install customtkinter
```

Si ese comando no funciona, prueba usando este otro:

```bash
python3 -m pip install customtkinter
```

➔ *Cryptography*:

```bash
pip install  cryptography
```

Si ese comando no funciona, prueba usando este otro:

```bash
python3 -m pip install  cryptography
```