Evidencia de aplicación de principios OWASP Top 10, autenticación JWT u OAuth2 y cifrado de contraseñas.

Integrantes:

Eberson Guayllas - Wilmer Pardo - Santiago Tamayo

1. Propósito

Evaluar la capacidad del estudiante para diseñar, implementar y documentar el backend del sistema multiplataforma, aplicando buenas prácticas de arquitectura, control de versiones, seguridad web y documentación profesional.

2. Desarrollo de la actividad

2.1 Vulnerabilidades del OWASP Top 10 (2021)

A continuación, se presenta un resumen de las vulnerabilidades críticas identificadas en el proyecto Codium, clasificadas según el estándar OWASP 2021:

- A02:2021 – Cryptographic Failures (Fallas Criptográficas)
  Riesgo en Codium: Las contraseñas están guardadas en texto plano y no cuentan con encriptación hash.
  Descripción de la Amenaza: Si las contraseñas se almacenan en un texto plano, una filtración de datos podría revelar todas las contraseñas de los usuarios. Además, el manejo del token_refresco en texto plano en la Base de Datos (BD) también representa un riesgo de seguridad.
- A03:2021 – Injection (Inyección)
  Riesgo en Codium: Uso potencial de consultas SQL no parametrizadas, exponiendo el sistema a ataques de Inyección SQL.
  Descripción de la Amenaza: Si las consultas SQL no usaran la sustitución de variables segura del conector de MySQL (es decir, concatenando directamente las entradas del usuario a la query), un atacante podría enviar comandos SQL maliciosos a través de las rutas API. Esto podría permitir obtener información sensible de la tabla ‘persona’.
- A04:2021 – Insecure Design (Diseño Inseguro)
  Riesgo en Codium: La configuración de la base de datos (credenciales y contraseña) está almacenada directamente en el código fuente (Backend/src/database/db.py).
  Descripción de la Amenaza: El uso directo de credenciales de base de datos (DB_CONFIG) dentro del archivo de código fuente es una práctica de diseño insegura. Si el código fuente se filtrara, las credenciales de acceso total a la BD quedarían expuestas. Además, se observa una falta de patrones de diseños seguros.
- A05:2021 – Security Misconfiguration (Fallo de Configuración de Seguridad)
  Riesgo en Codium: Uso de CORS abierto (CORS(app)) en Flask, permitiendo a cualquier dominio realizar peticiones al backend.
  Descripción de la Amenaza: El uso de CORS(app) sin restringir dominios en Backend/app.py permite que cualquier sitio web acceda al API. Esto puede facilitar que un sitio malicioso envíe solicitudes en nombre del usuario (si existieran sesiones activas), provocando posibles ataques CSRF (Falsificación de Solicitudes en Sitios Cruzados).
- A08:2021 – Software and Data Integrity Failures (Fallo de Integridad de Software y Datos)
  Riesgo en Codium: Falta de validación y control de la data recibida en el backend, permitiendo la inyección de datos inesperados.

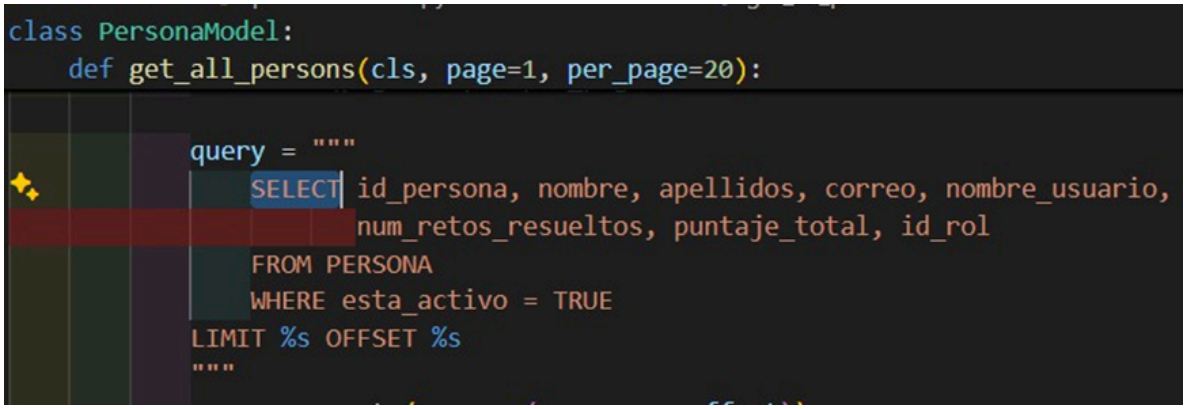
Descripción de la Amenaza: En el modelo persona, especialmente al realizar un POST, si no se controlan los campos a actualizar, un atacante podría intentar modificar campos críticos. Esto podría resultar en un cambio de roles o implantación de identidad.

Para la resolución de aquellas vulnerabilidades se ha planteado las siguientes medidas de mitigación:

- A02:2021 Medida Planificada: Uso de hashing seguro
  Explicación: La vulnerabilidad de almacenar contraseñas en texto plano se resolvió implementando el hashing seguro de contraseñas mediante la librería werkzeug.security de Flask.

```python
class PersonaModel:
    def create_person(cls,

        try:
            # Hashear la contraseña ANTES de la consulta
            # Recibimos texto plano, guardamos el hash
            hashed_password = generate_password_hash(contrasena_plana)

            # Query actualizada:
            # - Sin id_persona
            # - Sin num_retos_resueltos
            # - Sin puntaje_total
            query = """
                INSERT INTO persona (nombre, apellidos, correo, contraseña_hash, nombre_usuario, token_refresco, id_rol)
                VALUES (%s, %s, %s, %s, %s, %s, %s)
            """

            # Tupla de valores actualizada
            cursor.execute(query, (nombre, apellidos, correo, hashed_password, nombre_usuario, token_refresco, id_rol))

            # Obtener el ID que SÍ se generó
            new_person_id = cursor.lastrowid

            conn.commit()

            # Retornar el ID real
            return jsonify({"message": "Persona creada exitosamente", "id_persona": new_person_id}), 201
```

- A03:2021 Medida planificada: Uso exclusivo de consultas parametrizadas
  Explicación: Se ha implementado en personaModel todas las queries que utilizan el formado '%s' y la tupla de datos separada en el método cursor.execute(query, (data,)). Esto garantiza que los datos del usuario se envíen por separado del comando SQL, mitigando la Inyección SQL.



```python
class PersonaModel:
    def get_all_persons(cls, page=1, per_page=20):

            query = """
                SELECT id_persona, nombre, apellidos, correo, nombre_usuario,
                       num_retos_resueltos, puntaje_total, id_rol
                FROM PERSONA
                WHERE esta_activo = TRUE
            LIMIT %s OFFSET %s
            """
```

```
class PersonaModel:
    def delete_person(cls, id_persona):
        query = "UPDATE persona SET esta_activo = FALSE WHERE id_persona = %s"
        cursor.execute(query, (id_persona,))
```

- A04:2021 Medida Planificada: Incorporar seguridad desde el diseño (roles).
  Explicación: Seguir el principio de menor privilegio para el usuario DB app_user. Se limitaron los permisos del usuario que se conecta desde el backend, este se encuentra al final del archivo schema.sql

```
schema.sql
236  CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'S@ntiagoñ2002';
237  GRANT SELECT, INSERT, UPDATE, DELETE ON codium_db.* TO 'app_user'@'localhost';
238  FLUSH PRIVILEGES;
239
240
```

- A05:2021 Medida Planificada: Restricción estricta de CORS.
  Explicación: Cuando la aplicación se despliegue se restringirá para que solo la acceda la IP específica. Esto se realizara en la próxima unidad al integrar el frontend
- A08:2021 Medida Planificada: Control de campos actualizables
  Explicación: En personaModel se utiliza una lista blanca. Solo los campos de esta lista se utilizan para construir la query UPDATE.

```
Backend > src > models > personaModel.py > PersonaModel > update_person
  6  class PersonaModel:
134      def update_person(cls, id_persona, update_data):
139          # --- INICIO DE CORRECCIÓN DE SEGURIDAD ---
140          # Lista blanca de campos permitidos para actualización
141          allowed_fields = ['nombre', 'apellidos', 'nombre_usuario', 'token_refresco']
142
143          fields_to_update = []
144          values = []
145
146          for key, value in update_data.items():
147              if key in allowed_fields:
148                  fields_to_update.append(f"{key} = %s")
149                  values.append(value)
150              else:
151                  # Opcional: registrar intento de actualizar campo no permitido
152                  print(f"ADVERTENCIA: Intento de actualizar campo no permitido: {key}")
153
154          if not fields_to_update:
155              return jsonify({"error": "No se proporcionaron datos válidos para actualizar"}), 400
156          # --- FIN DE CORRECCIÓN DE SEGURIDAD ---
```

2.2 Autenticación JWT u OAuth2

Para implementar un mecanismo de autenticacion y autorizacion en el backend, primeramente se realizó una revisión teórica de los mecanismos de seguridad (JWT y OAuth2), con el fin de seleccionar el método más adecuado para una API REST desarrollada en flask; JWT es un estándar que define un formato compacto y seguro para representar información entre dos partes como un objeto JSON firmado digitalmente, mientras que, OAuth2 es un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a recursos protegidos en nombre de un usuario.

Luego se procedió a configurar el entorno de trabajo, instalando las dependencias necesarias para el manejo de tokens y control de accesos El desarrollo se centró en la creación de las rutas /auth/login y /auth/register, encargadas de la validación de tokens JWT, los cuales incluyen información como el identificador del usuario, su rol y el tiempo de expiración. Asimismo, se implementaron validaciones de entrada y políticas CORS que limitan los orígenes permitidos para las solicitudes, fortaleciendo la seguridad del sistema. Finalmente, se realizaron pruebas de funcionalidad con Postman, verificando el acceso a rutas protegidas y documentando las respuestas HTTP obtenidas.
Autenticacion Google:

```python
@auth_bp.route('/auth/google', methods=['POST'])
def google_auth():
    data = request.get_json()

    # validación
    required_fields = ['nombre', 'apellidos', 'correo', 'contrasena_plana', 'nombre_usuario']
    for field in required_fields:
        if not data or field not in data:
            return jsonify({"error": f"Falta el campo de autenticación de Google: {field}"}), 400

    correo = data['correo'].strip()

    # buscar por email si el usuario ya existe
    try:
        persona = authModel.get_person_by_email(correo)
    except Exception as e:
        print(f"Error al buscar persona por email: {e}")
        return jsonify({"error": "Error al buscar usuario en la base de datos"}), 500

    # Se logea o se registra
    if persona:
        # USUARIO EXISTE: LOGIN
        token_acceso = create_access_token(identity=str(persona['id_persona']))

        return jsonify({
            "message": "Inicio de sesión de Google exitoso",
            "id_persona": persona['id_persona'],
            "nombre_usuario": persona['nombre_usuario'],
            "token": token_acceso
        }), 200
    else:
        # USUARIO NO EXISTE: REGISTRAR (CREATE)
        try:
            cleaned_nombre = data['nombre'].strip()
            cleaned_apellidos = data['apellidos'].strip()
            cleaned_nombre_usuario = data['nombre_usuario'].strip()
```

2.3 Cifrado de contraseñas

El cifrado de contraseñas se implementa en el authModel usando la función generate_password_hash de la librería Werkzeug. Esta función toma la contrasena_plana recibida del usuario y la transforma en un hash criptográfico seguro. Este hash (y no la contraseña original) es el que finalmente se almacena en la columna contraseña_hash de la base de datos.

```
try:
    hashed_password = generate_password_hash(contrasena_plana)

    query = """
        INSERT INTO persona (nombre, apellidos, correo, contraseña_hash, nombre_usuario, token_refresco, id_rol)
        VALUES (%s, %s, %s, %s, %s, %s, %s)
    """

    cursor.execute(query, (nombre, apellidos, correo, hashed_password, nombre_usuario, token_refresco, id_rol))

    new_person_id = cursor.lastrowid

    conn.commit()


    return {"message": "Persona creada exitosamente", "id_persona": new_person_id}, 201
```

3. Bibliografía

[1] OWASP Foundation (2023). OWASP Top 10 – Web Application Security Risks.

[2] Auth0. JWT Handbook. https://auth0.com/learn/json-web-tokens

[3] Spring Security / Django Auth / Express JWT Documentation.

[4] PlantUML / Mermaid C4 Model Reference.\

[5] Top 10 most pressing web security challenges: OWASP TOP TEN. Top 10 Most Pressing Web Security Challenges