

TECNICATURA UNIVERSITARIA EN CIBERSEGURIDAD

DESARROLLO DE SOFTWARE SEGURO

Trabajo Practico Final

Diseño e Implementación de APIs Seguras con Flask,
FastAPI y OpenAPI – Segunda Entrega

Alumnos :

Cabrera, Ricardo Martin

Deparsia, Ignacio

Quaglia, Tomas

Profesor :

Licenciado Juan Pablo Villalba

Villa Real (CABA), Merlo (San Luis), Mendoza Capital (Mendoza)

Junio 2025

Índice

Resumen	3
Introducción	3
1.Desarrollo – Componente Teórica	
1.1 Estudio detallado de la autenticación por IP	3
1.1.1 Aplicaciones prácticas	3
1.1.2 Restricciones	4
1.1.3 Riesgos	4
1.2 Análisis de la autenticación por API Key	4
1.2.1 Implementación en encabezados o parámetros	4
1.2.2 Medidas de seguridad	4
1.2.3 Riesgos	5
1.2.4 Ejemplo práctico	5
1.3 OAuth 2.0 y JWT: flujos principales y ejemplos en la industria	5
1.3.1 OAuth 2.0	5
1.3.2 JWT (JSON Web Tokens)	6
1.3.3 Ejemplos en la industria	6
1.3.4 Riesgos y mitigaciones	6
1.4 Vulnerabilidades comunes y estrategias de mitigación	6
1.4.1 Inyección SQL	7
1.4.2 Cross-Site Request Forgery (CSRF)	7
1.4.3 Broken Authentication	7
1.4.4 Cross-Site Scripting (XSS)	8
1.4.5 Insecure Deserialization	8
2. Desarrollo - Componente Practica	
2.1 Implementar autenticación por API Key en Flask y FastAPI	9
2.1.1 Comparativa entre Flask y FastAPI	9
2.2 Implementación en Flask	9
2.2.1 Fragmento Del Código Adicional	10
2.3 Implementación en FastAPI	10
2.3.1 Fragmento Del Código Adicional	10
2.4 Configurar autenticación por IP, limitando acceso a direcciones específicas	10
2.4.1 Comparativa entre Flask y FastAPI	11
2.4.2 Implementación en FastAPI	11
2.4.3 Implementación en Flask	11
2.5 Integrar JWT en uno de los frameworks	12
2.5.1 Fragmento del código adicional	12
3. Realizar pruebas básicas de seguridad y documentar resultados	13
3.1 Prueba de código en Flask	13
3.2 Prueba de código en FastAPI	16
4. Actualizar la documentación OpenAPI	19
5. Conclusión	20
6. Referencias Bibliográficas	20
7. Bibliografía	21

Resumen

Este trabajo presenta un análisis detallado de la implementación de los mecanismos avanzados de autenticación aplicables al Desarrollo De Software Seguro. Se describen los usos, riesgos y buenas prácticas de la autenticación por IP, el uso de API Keys, y de los protocolos OAuth 2.0 y JWT. A lo largo del informe se identificaron vulnerabilidades frecuentes asociadas a cada método y se propusieron medidas de mitigación concretas, destacando la importancia de aplicar controles como el uso de HTTPS, validación de tokens, y la combinación de factores de autenticación. Se concluye que una arquitectura segura requiere tanto el diseño cuidadoso de los mecanismos de acceso como una continua evaluación de los vectores de ataque comunes.

Introducción

En el desarrollo de software moderno, la autenticación segura es un componente esencial para proteger los sistemas frente a accesos no autorizados y vulnerabilidades comunes. Esta tarea analiza distintos mecanismos avanzados de autenticación utilizados en APIs y aplicaciones web, como la autenticación por IP, API Keys, OAuth 2.0 y JWT que serán sumadas a nuestra autenticación básica. Además, se estudian las vulnerabilidades más frecuentes asociadas a estos mecanismos, como la inyección SQL, CSRF y robo de tokens, junto con estrategias prácticas para su mitigación. Este enfoque integral busca fortalecer la seguridad de las aplicaciones y garantizar una correcta implementación de políticas de acceso.

1. Desarrollo – Componente Teórica

1.1 Estudio detallado de la autenticación por IP

La autenticación por IP es un mecanismo que controla el acceso a un sistema basándose en la dirección IP del cliente. Es como si le dijeras al sistema: "Solo deja entrar a los que vienen de esta dirección específica".

1.1.1 Aplicaciones prácticas

Se usa mucho en entornos corporativos, como VPNs o sistemas internos, donde se sabe que los usuarios legítimos siempre se conectan desde una red específica (por ejemplo, la red de una oficina con IPs fijas). También es común en APIs privadas o en servicios de nube como AWS para restringir el acceso a recursos específicos, como bases de datos o buckets de S3. Por ejemplo, un servidor puede configurar un firewall para aceptar conexiones solo desde un rango de IPs predefinido, como 192.168.1.0/24.

1.1.2 Restricciones

El principal problema es que las IPs no siempre son estáticas. En redes domésticas o móviles, los proveedores de internet asignan IPs dinámicas que cambian constantemente, lo que complica su uso para autenticar usuarios individuales. Además, no es práctico en aplicaciones públicas porque bloquear por IP puede excluir a usuarios legítimos que comparten la misma IP pública (como en una red de universidad o cafetería).

1.1.3 Riesgos

La autenticación por IP es vulnerable al "IP spoofing", donde un atacante falsifica su dirección IP, por medio del uso de VPNs o proxies, para hacerse pasar por un cliente autorizado. También, si un atacante compromete un dispositivo dentro de la red autorizada, tiene vía libre. Otro riesgo es que las listas de IPs permitidas (whitelists) pueden ser difíciles de mantener en entornos grandes, y un error en la configuración puede bloquear usuarios legítimos o dejar entrar a los no deseados. Para mitigar estos riesgos, se recomienda combinar la autenticación por IP con otros métodos, como certificados o contraseñas, y usar firewalls robustos con monitoreo constante.

1.2 Análisis de la autenticación por API Key

La autenticación por API Key es como darle una llave secreta a un cliente para que pueda usar tu API. Es un string único (a veces un hash largo, como `a1b2c3d4-5678-90ab-cdef-1234567890ab`) que el cliente incluye en sus requests para probar que tiene permiso. Es súper común en APIs públicas, como las de Google Maps o Twitter, donde necesitas identificar quién está haciendo las llamadas sin meterte en procesos complicados.

1.2.1 Implementación en encabezados o parámetros

Hay dos formas principales de pasar una API Key:

- **En encabezados HTTP:** La más común es poner la key en el encabezado `Authorization` o uno personalizado, como `X-API-Key`. Por ejemplo:
`Authorization: Bearer a1b2c3d4-5678-90ab-cdef-1234567890ab`.
Esto es más seguro porque los encabezados no suelen quedar expuestos en logs o URLs.
- **En parámetros de la URL:** En esta otra forma la key va como parte del query string, tipo `https://api.ejemplo.com/data?api_key=a1b2c3d4`. Es más simple, pero menos seguro porque las URLs pueden quedar en historiales de navegadores, logs de servidores o hasta en marcadores.

1.2.2 Medidas de seguridad

Las API Keys son fáciles de implementar, pero también son fáciles de vulnerar si no las proteges bien. Primero, **nunca hay que dejarlas hardcoded en el código fuente**,

porque cualquier desarrollador con acceso al repositorio (o un atacante que lo robe) puede usarlas. Lo conveniente es guardarlas en variables de entorno o en un gestor de secretos como AWS Secrets Manager.

Segundo, es fundamental asegurar que las comunicaciones se realicen siempre mediante HTTPS para que la key no viaje en texto plano.

Tercero, implementar límites de uso (rate limiting) para evitar que un atacante abuse de una key robada. También se podría rotar las keys periódicamente y darles un tiempo de vida corto. Por último, asegurarse de que cada key esté asociada a un cliente específico y con permisos mínimos (principio de menor privilegio).

1.2.3 Riesgos

Si una API Key se filtra y se ve comprometida, el atacante puede usarla hasta que se desactive. También hay riesgos si no se valida bien el origen de las requests: alguien podría intentar un ataque de repetición (replay attack) si no se usan mecanismos como nonces o timestamps. Otro punto crítico son las keys en parámetros de URL, ya que son un blanco fácil para atacantes que intercepten tráfico o accedan a logs.

1.2.4 Ejemplo práctico

Imagina que tenés una API para un servicio de mapas. El cliente incluye su key en el encabezado: X-API-Key: a1b2c3d4. Tu servidor valida la key contra una base de datos, chequea que no esté expirada y que el cliente no haya superado su cuota de requests. Si todo está OK, devolvés los datos; si no, un error 401 (Unauthorized).

1.3 OAuth 2.0 y JWT: flujos principales y ejemplos en la industria

1.3.1 OAuth 2.0

Es el método de autenticación moderna más utilizado hoy en día. OAuth 2.0 es un protocolo que permite que una aplicación acceda a recursos de otra en nombre de un usuario, sin compartir contraseñas. Un ejemplo claro es cuando iniciamos sesión en una app con nuestra cuenta de Google: la app no ve nuestra contraseña, solo recibe un token de acceso. Los flujos principales son:

- **Authorization Code Flow:** Ideal para apps web. El usuario se autentica en el proveedor (como Google), recibe un código temporal, y la app lo canjea por un access token. Ejemplo: Spotify usa este flujo para que conectes tu cuenta a otras apps.
- **Implicit Flow:** Más simple, usado en apps de una sola página (SPAs). El token se devuelve directamente en la URL, pero es menos seguro porque el token queda expuesto. Está cayendo en desuso.

- **Client Credentials Flow:** Para comunicación entre servidores, sin usuario involucrado. Por ejemplo, una API interna de una empresa que autentica servicios backend.
- **Refresh Token Flow:** Usas un refresh token para obtener nuevos access tokens cuando el original expira, sin pedirle al usuario que se autentique de nuevo.

1.3.2 JWT (JSON Web Tokens)

Un JWT es un formato compacto de token que transporta información entre partes de forma segura, mediante firma digital para garantizar su integridad. Está compuesto por tres partes: **Header** (información del algoritmo y tipo de token), **Payload** (datos como identificadores y permisos), y **Signature** (firma que valida el contenido). Su uso es común en conjunto con OAuth 2.0, sirviendo como formato del access token. Se ve así:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c.
```

1.3.3 Ejemplos en la industria

Google usa OAuth 2.0 para casi todos sus servicios: Gmail, Drive, YouTube, etc. Normalmente utilizando el Authorization Code Flow. En el caso de JWT, servicios como Firebase Authentication lo usan para autenticar usuarios en apps móviles. Por ejemplo, cuando inicias sesión en una app de Firebase, el servidor te da un JWT que el cliente usa para probar quién sos en cada request.

1.3.4 Riesgos y mitigaciones

OAuth 2.0 puede ser vulnerable a ataques como redirecciones maliciosas si no se validan bien las URLs de redirección. Con JWT, un error clásico es no validar la firma o utilizar algoritmos débiles (como none). Para mitigar estos riesgos se recomienda:

- Usar siempre HTTPS.
- Validar los tokens del lado del servidor.
- Evitar incluir información sensible en el *payload* del JWT, ya que puede ser fácilmente decodificado.
- Establecer tiempos de vida cortos para los tokens.
- Utilizar refresh tokens para extender sesiones de forma segura.
-

1.4 Vulnerabilidades comunes y estrategias de mitigación

A continuación analizaremos las cinco vulnerabilidades más comunes según estándares como el OWASP Top Ten, con un enfoque en los sistemas de autenticación que estuvimos discutiendo (IP, API Key, OAuth 2.0, JWT).

1.4.1 Inyección SQL

Este ataque ocurre cuando un atacante inserta código SQL malicioso en un campo de entrada, como un formulario de login o un parámetro de API. En el contexto de autenticación, es un problema grave si, por ejemplo, se está validando una API Key o un JWT contra una base de datos y los inputs no están sanitizados. Imaginemos un endpoint de autenticación que toma un parámetro `api_key` y lo pasa directo a una consulta como `SELECT * FROM users WHERE api_key = ' + input + '`. Si el atacante introduce algo como `' OR '1'='1`, podría acceder a datos sensibles o incluso bypassar la autenticación.

- **Mitigación:** Utilizar consultas parametrizadas o prepared statements ayuda a evitar que el input del usuario se interprete como código SQL. Por ejemplo, en lugar de concatenar strings, podríamos usar `SELECT * FROM users WHERE api_key = ?` y pasar el parámetro de forma segura. También debemos sanitizar y validar todos los inputs (por ejemplo, asegurese de que la API Key tenga el formato esperado). Un ORM como Sequelize o Hibernate puede ayudar a evitar errores. Además, se debe limitar los permisos de la base de datos para que el usuario de la DB solo pueda ejecutar las operaciones estrictamente necesarias.

1.4.2 Cross-Site Request Forgery (CSRF)

CSRF engaña a un usuario autenticado para que realice acciones no deseadas en una aplicación. En el contexto de OAuth 2.0, por ejemplo, un atacante podría crear un formulario malicioso que envíe una solicitud a un endpoint protegido por un access token, aprovechando que el usuario ya está autenticado. Si nuestra API no valida el origen de la solicitud, el atacante podría, por ejemplo, cambiar la configuración de una cuenta.

- **Mitigación:** Implementar tokens CSRF en todos los formularios o endpoints que modifiquen datos. Estos tokens son valores únicos por sesión que el servidor valida antes de procesar la solicitud. También configurar las cookies con el atributo `SameSite` (`Strict` o `Lax`) para evitar que se envíen en solicitudes cross-site. Para acciones críticas (como revocar un token OAuth), podríamos pedir autenticación adicional, como una contraseña o un PIN. Y además utilizar métodos HTTP seguros (POST en lugar de GET) para acciones sensibles.

1.4.3 Broken Authentication

Esta vulnerabilidad la vemos cuando los mecanismos de autenticación están mal implementados, permitiendo que un atacante robe sesiones, credenciales o tokens. En autenticación por IP, un atacante podría falsificar una IP permitida (IP spoofing) para

acceder. Con API Keys, si se filtran (por ejemplo, por estar en un repositorio público), cualquiera puede usarlas. En OAuth 2.0 o JWT, errores como no validar firmas de JWT, usar algoritmos débiles (como **none**) o permitir redirecciones maliciosas en el flujo OAuth pueden comprometer todo el sistema.

- **Mitigación:** Para autenticación por IP, podemos combinar con otros métodos (como certificados de cliente) y usar firewalls para filtrar tráfico sospechoso. Para API Keys, podríamos guardarlas en gestores de secretos, rotarlas frecuentemente y aplicar rate limiting. En OAuth 2.0, podemos validar estrictamente las URLs de redirección y usar tokens con tiempo de vida corto. Para JWT, siempre hay que verificar la firma en el servidor, usar algoritmos fuertes (como HS256 o RS256) y evitar almacenar datos sensibles en el payload. HTTPS es obligatorio para todos estos métodos para evitar interceptaciones.

1.4.4 Cross-Site Scripting (XSS)

XSS permite a un atacante inyectar scripts maliciosos en una página web que ejecutan los navegadores de los usuarios. En el contexto de autenticación, un ataque XSS podría robar un JWT o un access token de OAuth 2.0 almacenado en el navegador (por ejemplo, en **localStorage** o una cookie). Por ejemplo, si un endpoint de nuestra API devuelve datos sin sanitizar (como un nombre de usuario que incluye `<script>alert('hackeado')</script>`), el script podría ejecutarse y enviar el token a un servidor controlado por el atacante.

- **Mitigación:** Sanitizar y escapar todos los datos que se muestren en la interfaz (por ejemplo, usando bibliotecas como **DOMPurify**). Configurar los encabezados HTTP como **Content-Security-Policy** (CSP) para limitar qué scripts pueden ejecutarse. Debemos evitar almacenar tokens sensibles en **localStorage**; usar cookies con los atributos **HttpOnly**, **Secure** y **SameSite**. Y también validar los inputs en el servidor para evitar que se inyecten scripts maliciosos en primer lugar.

1.4.5 Insecure Deserialization

Esto ocurre cuando una aplicación de serializa datos no confiables sin validarlos, permitiendo que un atacante ejecute código arbitrario o manipule objetos. En el contexto de JWT, un atacante podría intentar manipular el payload si el servidor no valida correctamente la firma o si usa una clave débil para firmar el token. Por ejemplo, si un servidor acepta un JWT con un algoritmo **none**, el atacante puede crear un token falso y hacerse pasar por un usuario legítimo.

- **Mitigación:** Siempre hay que validar la firma de los JWT en el servidor y usar claves seguras (nunca uses **none** como algoritmo). Para cualquier deserialización (por ejemplo, en APIs que procesan JSON o XML), debemos sanitizar los datos y usar

bibliotecas seguras que limiten la ejecución de código arbitrario. Evitar de serializar datos de fuentes no confiables y, si es posible, utilizar formatos de datos más simples (como JSON puro) en lugar de formatos complejos como XML. También podemos aplicar el principio de menor privilegio para limitar el impacto de un ataque exitoso.

2. Desarrollo - Componente Practica

2.1 Implementar autenticación por API Key en Flask y FastAPI

Si bien esta sección corresponde a la componente práctica, reforzaremos lo expuesto en la componente teórica lo que explyara el trabajo realizado.

Para la implementación practica de protección de rutas de acceso con Autenticación Básica instaurada en la primer entrega, ahora sumaremos API Key, la cual es una técnica común para controlar el acceso a recursos.

La API Key es un identificador único que el cliente debe incluir en cada solicitud, ya sea en los encabezados o como parámetro de URL. Para reforzar la seguridad.

Nosotros la sumamos a la autenticación básica (HTTP Basic Auth), donde el cliente también debe proporcionar un nombre de usuario y contraseña válidos.

Este enfoque doble implementa el principio de defensa en profundidad, agregando una segunda capa de validación a los endpoints sensibles.

2.1.1 Comparativa entre Flask y FastAPI

Característica	Flask	FastAPI
Manejo de API Key	Manual, usando request.headers o request.args	Uso de Security con APIKeyHeader
Manejo de autenticación básica	Manual, con request.authorization	Uso de Depends con HTTPBasic
Decoradores de protección	Decorador personalizado (@autenticacion_doble)	Dependencias declaradas como argumentos
Validación automática	No incluida	Incluida mediante Depends y Security
Integración con Swagger	Usando flask_restx con authorizations	Automática mediante anotaciones y dependencias
Carga de variables .env	python-dotenv + os.getenv()	Igual en ambos frameworks

2.2 Implementación en Flask

Se usó Flask junto con la extensión flask_restx para generar documentación automática y proteger una ruta con API Key y autenticación básica combinadas.

- La API Key se obtiene desde los headers o la URL.
- La autenticación básica se verifica con request.authorization.
- Ambas condiciones se validan en un decorador personalizado @autenticacion_doble.

2.2.1 Fragmento Del Código Adicional

```
def autenticacion_doble(f):
    @wraps(f)
    def decorador(*args, **kwargs):
        api_key = request.headers.get('X-API-Key') or request.args.get('apikey')
        if not api_key or api_key != API_KEY_ESPERADA:
            return {"mensaje": "API Key inválida o ausente."}, 401

        auth = request.authorization
        if not auth or not (auth.username == 'admin' and auth.password == 'secret'):
            return (
                {"mensaje": "Autenticación básica fallida."},
                401,
                {'WWW-Authenticate': 'Basic realm="Login Required"'})
        return f(*args, **kwargs)
    return decorador
```

2.3 Implementación en FastAPI

En FastAPI, se aprovechó la inyección de dependencias y herramientas del módulo fastapi.security para definir una doble validación más estructurada.

- Se usaron HTTPBasic y APIKeyHeader para capturar credenciales.
- La API Key se permite en header o query string.
- Las funciones de validación se integran directamente como dependencias en la ruta.

2.3.1 Fragmento Del Código Adicional

```
@app.get("/api/protegida", response_class=HTMLResponse)
async def ruta_protegida(
    api_key: str = Security(verificar_api_key),
    credenciales: HTTPBasicCredentials = Depends(verificar_credenciales)
):
```

2.4 Configurar autenticación por IP, limitando acceso a direcciones específicas

Para el caso del Control de Acceso por IP en APIs con Flask y FastAPI, implementamos una medida de seguridad que restringe el acceso a una aplicación solo a direcciones IP específicas. Esta técnica es útil para limitar el acceso a recursos sensibles, permitiendo solo a usuarios o sistemas autorizados interactuar con la API.

2.4.1 Comparativa entre Flask y FastAPI

Característica	Flask	FastAPI
Obtención de IP del cliente	request.remote_addr	request.client.host
Implementación del control	Decorador @before_request o funciones personalizadas	Dependencias (Depends) en las rutas
Carga de IPs permitidas	Desde archivo .env utilizando python-dotenv	Desde archivo .env utilizando python-dotenv
Manejo de múltiples IPs	Lista o conjunto (set) de IPs permitidas	Lista o conjunto (set) de IPs permitidas
Respuesta ante IP no permitida	Retorno de error 403 con mensaje personalizado	Lanzamiento de HTTPException con código 403 y detalle del error

2.4.2 Implementación en FastAPI

```

from fastapi import Request, HTTPException, Depends
from starlette.status import HTTP_403_FORBIDDEN

ALLOWED_IPS = {"127.0.0.1", "192.168.1.100"}

async def verificar_ip(request: Request):
    client_ip = request.client.host
    if client_ip not in ALLOWED_IPS:
        raise HTTPException(status_code=HTTP_403_FORBIDDEN,
                            detail=f"Acceso denegado para IP {client_ip}")
    return client_ip

@app.get("/ruta-protegida")
async def ruta_protegida(client_ip: str = Depends(verificar_ip)):
    return {"mensaje": f"Acceso concedido desde IP {client_ip}"}

```

2.4.3 Implementación en Flask

```

from flask import Flask, request, abort
app = Flask(__name__)

ALLOWED_IPS = {"127.0.0.1", "192.168.1.100"}

@app.before_request
def limitar_ip():
    client_ip = request.remote_addr
    if client_ip not in ALLOWED_IPS:
        abort(403, description=f"Acceso denegado para IP {client_ip}")

@app.route("/ruta-protegida")
def ruta_protegida():
    return {"mensaje": f"Acceso concedido desde IP {request.remote_addr}"}

```

2.5 Integrar JWT en uno de los frameworks (Flask o FastAPI, a elección).

Para la integración de JWT elegimos el framework de FastAPI. JWT (JSON Web Token) es un estándar abierto (RFC 7519) que permite el intercambio seguro de información como un objeto JSON. Es ampliamente utilizado en sistemas de autenticación y autorización, ya que permite que un servidor genere un token firmado digitalmente que puede ser usado para identificar al usuario sin necesidad de consultar una base de datos en cada solicitud.

Son varios los beneficios del uso de JWT, los más importantes son :

Seguridad : el token está firmado y tiene tiempo de expiración.

Escalabilidad : se puede usar el mismo JWT para múltiples endpoints.

Comodidad para el usuario : el token puede guardarse localmente y reutilizarse.

Separación de responsabilidades clara entre autenticación inicial y validación posterior.

Implementación en el Proyecto

En este proyecto con el mecanismo JWT se implementa de la siguiente manera :

1. El usuario accede a la ruta /api/protegida, donde se valida su IP, una API Key y credenciales HTTP básicas.
2. Si todas las verificaciones son exitosas, se genera un JWT con su nombre de usuario y una expiración de 60 minutos. El **tiempo de vida promedio** de un token JWT depende del contexto y del nivel de seguridad requerido.
3. Este token se presenta al usuario junto con opciones para copiarlo, guardarlo en el navegador (localStorage), o utilizarlo para acceder a una segunda ruta.
4. La ruta /api/con_token está protegida y requiere que el cliente envíe un JWT válido en el encabezado Authorization.
5. Si el token es válido y no ha expirado, se permite el acceso y se muestra el nombre del usuario.

2.5.1 Fragmento del código adicional

```
from fastapi import Depends, HTTPException
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from datetime import datetime, timedelta
import os
```

```
# Cargar variables necesarias
JWT_SECRET = os.getenv("JWT_SECRET")
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# Función para crear el token JWT
def crear_token_jwt(data: dict, expires_delta: timedelta = None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode["exp"] = expire
    encoded_jwt = jwt.encode(to_encode, JWT_SECRET, algorithm=ALGORITHM)
    return encoded_jwt

# Función para verificar el token JWT
async def verificar_token_jwt(token: str = Depends(oauth2_scheme)):
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=[ALGORITHM])
        username = payload.get("sub")
        if username is None:
            raise HTTPException(status_code=401, detail="Token inválido")
        return username
    except JWTError:
        raise HTTPException(status_code=401, detail="Token inválido o expirado")
```

3. Realizar pruebas básicas de seguridad y documentar resultados

Para ambos casos, las pruebas de soportes adicionales de seguridad se fueron agregando una a una. Como resultado, tenemos un código que integra todas las validaciones requeridas.

Nos valimos del uso de un archivo .env para guardar allí los datos sensibles y evitar que los mismos se encuentren harcodeados en el código fuente.

También hicimos uso del .env para quitar las credenciales de usuario y contraseña del código fuente tanto en FastAPI como en FLASK.

3.1 Prueba de código en Flask

Para el funcionamiento correcto del código en FLASK, debemos tener instaladas las siguientes dependencias, se muestran también las versiones recomendadas :

```
flask==3.0.3
flask-restx==1.3.0
python-dotenv==1.0.1
```

Y el archivo .env con los siguientes datos :

```
API_KEY=supersecreta123
JWT_SECRET=secretojwt123
```

ALLOWED_IPS=127.0.0.1,192.168.1.10,192.168.1.98
ADMIN_USER=admin
ADMIN_PASSWORD=secret

Este archivo .env se incluirá en GitHub solo a título de referencia, ya que no es una buena práctica dejarlo disponible al tener todos los datos sensibles. Este archivo puede ubicarse en :

https://github.com/Rimaca1973/UGR_Desarrollo_De_Software_Seguro/blob/main/.env

Luego cargaremos por consola el código `tp002-flask.py` (que se encuentra en el repositorio de GitHub) con :

Python tp002-flask.py

Lo que nos dará este resultado :

```
c:\Users\Rimaca\Desktop>python tp002-flask.py
* Serving Flask app 'tp002-flask'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.98:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 722-800-951
```

Luego en el navegador ingresamos :

127.0.0.1:5000/protegida

El resultado será :



Lo cual es correcto, ya que no se indicó la APIKEY. Nuevamente nos dirigimos al navegador e ingresamos la dirección seguida de la APIKEY en el query. Si bien esta no es la mejor implementación, para la prueba de laboratorio es suficientemente útil.

<http://127.0.0.1:5000/protegida/?apikey=supersecreta123>

El resultado será :

Acceder
http://127.0.0.1:5000
Nombre de usuario
Contraseña

Si se ingresan credenciales incorrectas, volverá a presentarse el mismo cuadro de dialogo pidiendo nuevamente las credenciales.

Si ingresamos en nombre de usuario admin y contraseña secret, el resultado será :

 **Acceso Concedido**
 **Universidad del Gran Rosario**

Usuario: admin
Contraseña: secret
API Key: supersecreta123
IP origen: 127.0.0.1
IPs permitidas: 127.0.0.1, 192.168.1.10, 192.168.1.98
Usted ingresó a las: 20:51 del 01-06-2025

⚠ Datos proporcionados a efectos de ejemplo en laboratorio

En este caso el acceso se hizo con la dirección local. También probamos con otra dirección, la física de la máquina que es 192.168.1.98, el resultado es :

 **Acceso Concedido**
 **Universidad del Gran Rosario**

Usuario: admin
Contraseña: secret
API Key: supersecreta123
IP origen: 192.168.1.98
IPs permitidas: 127.0.0.1, 192.168.1.10, 192.168.1.98
Usted ingresó a las: 20:54 del 01-06-2025

⚠ Datos proporcionados a efectos de ejemplo en laboratorio

Para forzar el error de acceso por IP, en la lista del archivo .env solo dejaremos la dirección 192.168.1.10 activa.

Al ingresar en el navegador

<http://127.0.0.1:5000/protegida/>

El resultado será :



3.2 Prueba de código en FastAPI

Para el funcionamiento correcto del código en FastAPI, debemos tener instaladas las siguientes dependencias, se muestran también las versiones recomendadas :

```
fastapi==0.110.0
uvicorn==0.29.0
python-dotenv==1.0.1
python-jose[cryptography]==3.3.0
```

Y el archivo .env con los siguientes datos :

```
API_KEY=supersecreta123
JWT_SECRET=secretojwt123
ALLOWED_IPS=127.0.0.1,192.168.1.10,192.168.1.98
ADMIN_USER=admin
ADMIN_PASSWORD=secret
```

Este archivo .env se incluirá en GitHub solo a título de referencia, ya que no es una buena práctica dejarlo disponible al tener todos los datos sensibles. Este archivo puede ubicarse en :

https://github.com/Rimaca1973/UGR_Desarrollo_De_Software_Seguro/blob/main/.env

Luego cargaremos por consola con uvicorn el código [tp002-flastapi.py](#) (que se encuentra en el repositorio de GitHub) con :

```
uvicorn tp002-fastapi:app --reload
```

Lo que nos dará este resultado :

```
c:\Users\Rimaca\Desktop>uvicorn tp002-fastapi:app --reload
INFO: Will watch for changes in these directories: ['c:\\Users\\Rimaca\\Desktop']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [9104] using WatchFiles
INFO: Started server process [5288]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Luego en el navegador ingresamos :

<http://127.0.0.1:8000/api/prottegida>

Lo que nos dará como resultado :

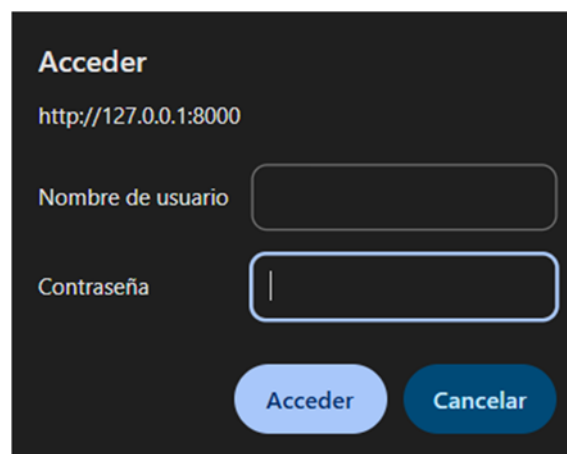
```
{"detail": "API Key inválida"}
```

Lo cual es correcto, ya que no se ingresó la APIKEY correspondiente.

Nuevamente nos dirigimos al navegador e ingresamos la dirección seguida de la APIKEY en el query. Si bien esta no es la mejor implementación, para la prueba de laboratorio es suficientemente útil.

http://127.0.0.1:8000/api/prottegida?api_key=supersecreta123

El resultado será :



Acceder

http://127.0.0.1:8000

Nombre de usuario

Contraseña

Si ingresamos las credenciales invalidas, obtendremos el aviso :

```
{"detail": "Credenciales inválidas"}
```

Refrescamos la página para que nos vuelva a solicitar las credenciales, y si ingresamos en nombre de usuario admin y en contraseña secret, el resultado será

¡ Acceso concedido !

- **Usuario :** admin
- **Contraseña :** secret
- **API Key :** supersecretal23
- **JWT Secret :** secretojwt123
- **Su IP :** 127.0.0.1
- **IPs Permitidas :**
 - 192.168.1.10
 - 192.168.1.98
 - 127.0.0.1
- **JWT generado :** eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbSIiSwV4cCI6MTc0ODgzNDkyMn0.eyJFei03bGCoC4n_jlNe0tYsvRIJe5ez0tBtvJr5ypj8

Copiar Token Guardar Token Usar Token en /api/con_token

🔑 **Token almacenado en localStorage:**

(vacío)

Cerrar sesión

Podemos probar con los distintos botones :

COPIAR TOKEN

127.0.0.1:8000 dice

Token copiado al portapapeles!

Aceptar

GUARDAR TOKEN

127.0.0.1:8000 dice

Token guardado en localStorage!

Aceptar

USAR TOKEN EN /API/CON_TOKEN

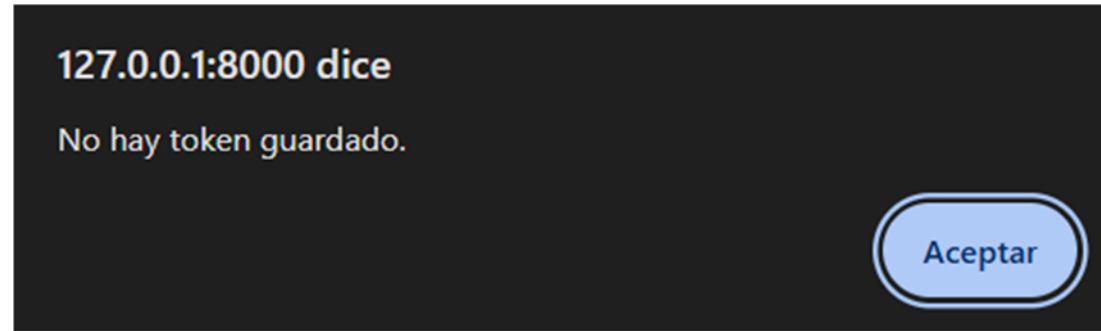
✅ **Hola admin, accediste con un JWT válido!**

Esta es una página protegida usando JSON Web Token.

Cerrar sesión

PRUEBAS DE ERROR

USAR TOKEN EN /API/CON_TOKEN sin haber guardado token



Para forzar el error de acceso por IP, en la lista del archivo .env solo dejaremos la dirección 192.168.1.10 activa. Al ingresar en el navegador nuevamente con :

http://127.0.0.1:8000/api/prottegida?api_key=supersecreta123

Obtendremos :

```
{"detail": "Acceso denegado para IP 127.0.0.1"}
```

De esta manera verificamos los errores que confirman nuestros medios de validación activos.

4. Actualizar la documentación OpenAPI

Documentación de Flask API

Podemos acceder a la información auto documentada de Swagger desde el navegador a partir de la dirección :

<http://127.0.0.1:5000/docs>

Esta información esta almacenada en el repositorio de GitHub como el archivo [tp002-flask-swagger.pdf](#)

Documentación de FastAPI

Podemos acceder a la información auto documentada de Swagger desde el navegador a partir de la dirección :

<http://127.0.0.1:8000/docs>

Esta información esta almacenada en el repositorio de GitHub como el archivo [tp002-fastapi-swagger.pdf](#)

Adicionalmente tenemos acceso también a la información auto documentada de ReDocs desde el navegador a partir de la dirección :

<http://127.0.0.1:8000/redocs>

Esta información esta almacenada en el repositorio de GitHub como el archivo [tp002-fastapi-redocs.pdf](#)

5. Conclusión

La implementación adecuada de mecanismos de autenticación avanzada es fundamental para garantizar la seguridad en el desarrollo de software. Cada método —ya sea por IP, API Key, OAuth o JWT— aporta ventajas según el contexto, pero también implica riesgos específicos si no se aplica con buenas prácticas. La clave está en no confiar exclusivamente en un único método, sino en combinar capas de seguridad, aplicar validaciones rigurosas, y adoptar mecanismos de control como la rotación de claves, uso de HTTPS y políticas de expiración. Asimismo, conocer las vulnerabilidades más comunes y sus mitigaciones permite reducir significativamente la superficie de ataque de nuestras aplicaciones. El desarrollo seguro es, en definitiva, una responsabilidad constante que requiere tanto conocimiento técnico como disciplina en su aplicación.

6. Referencias Bibliográficas

- Gemini (Modelo de lenguaje de IA). (2025). Interacción con el usuario sobre el análisis y mejora de código Python para una aplicación IVR. [Fecha de la interacción: 9 de abril de 2025]. <https://gemini.google.com/app/983078de7faa96e7?hl=es&pli=1>
- Josefsson, S. (2006). The Base16, Base32, and Base64 Data Encodings (RFC 4648). Internet Engineering Task Force. Recuperado de <https://datatracker.ietf.org/doc/html/rfc4648>
- Mozilla Foundation. (s.f.). Base64 – MDN Web Docs. Recuperado de <https://developer.mozilla.org/en-US/docs/Glossary/Base64>
- Open Web Application Security Project. (2023). Authentication Cheat Sheet. Recuperado de https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
- Python Software Foundation. (s.f.). base64 – Base16, Base32, Base64, Base85 Data Encodings. Recuperado de <https://docs.python.org/3/library/base64.html>

7. Bibliografía

- Google. (2024). Gemini (versión 1.5) [Modelo de lenguaje]. <https://gemini.google.com/>
- OpenAI. (2024). ChatGPT (versión GPT-4) [Modelo de lenguaje]. <https://chat.openai.com/>