

Informe Proyecto Final - Puesta y Producción Segura (PPS)

Nombre: Natxo Rodrigo Cervera

Curso: Especialización de ciberseguridad

Fecha: 15/05/2025

ÍNDICE

1. Análisis y planificación	3
2. Infraestructura y despliegue seguro	6
2.1 Infraestructura con Docker Compose	6
2.2 Proxy inverso con NGINX	8
2.3 HTTPS/TLS con certificado autofirmado	8
2.4 Gestión segura de secretos	10
2.5 Hardening aplicado	10
3. Desarrollo seguro + CI/CD	13
3.1 Desarrollo seguro	13
3.2 Testing automático	13
3.3 Pipeline CI/CD	13
3.4 Seguridad en dependencias	14
3.5 Objetivo del workflow:	14
4. Seguridad operativa	18
4.1 Centralización de logs con Loki	18
4.2 Monitorización con Prometheus + Grafana	19
4.3 Detección de eventos anómalos (alertas)	19
4.4 Control de accesos y privilegios mínimos	20
4.5 Simulación de copia de seguridad cifrada	21
4.6 Copia de seguridad cifrada (simulada)	21
4.7 Comando de cifrado utilizado:	22
4.8 Consideraciones reales en entornos de producción	23
5. Pruebas de seguridad	25
5.1 Análisis de vulnerabilidades del contenedor backend con Trivy	25
5.2 Realización de pruebas con herramientas de análisis estático	27
5.3 Gestión de secretos / .env	30
5.4 Análisis de dependencias con Safety	31
5.5 Puertos, usuarios y seguridad en contenedores	32
Despliegue del entorno	34
REFLEXION FINAL	35

1. Análisis y planificación

- Análisis de riesgos (STRIDE, LINDDUN o híbrido).

Este apartado tiene como objetivo identificar y mitigar los riesgos de seguridad que podrían afectar a la infraestructura desplegada. Se utiliza el modelo STRIDE para clasificar las amenazas, se representa visualmente el modelo de amenazas y se establece una arquitectura segura de red. Además, se proponen medidas técnicas y organizativas para garantizar la seguridad del sistema.

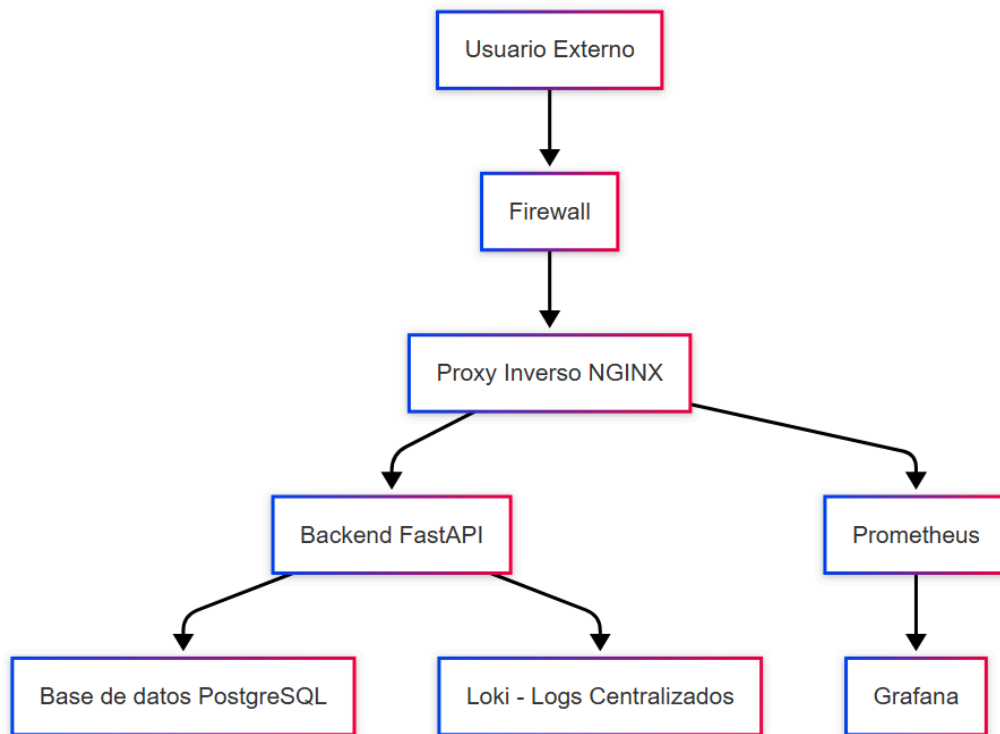
- Modelado de amenazas con diagrama.

Amenaza	Descripción	Contramedida aplicada
Spoofing	Suplantación de identidad en los accesos al backend	Uso de autenticación basada en JWT
Tampering	Manipulación de código o datos	Control de versiones con Git + validaciones de entrada
Repudiation	Ausencia de trazabilidad de acciones	Centralización de logs con Loki y Grafana
Information Disclosure	Fugas de información sensible	Gestión segura de secretos con ficheros .env
Denial of Service	Saturación del backend o puertos	Uso de NGINX como proxy inverso + filtrado de tráfico
Elevation of Privilege	Escalada de permisos	Validación estricta de roles en la API

Se ha realizado un diagrama que representa visualmente el flujo de datos y los componentes clave de la infraestructura, identificando puntos de entrada, procesos y almacenes de datos, así como amenazas asociadas a cada uno.

- Diagrama de arquitectura segura (firewalls, proxies, redes internas, etc).

Se ha diseñado una arquitectura lógica segura que contempla separación por capas, zonas de red diferenciadas, filtrado de tráfico mediante proxy inverso y contenedores aislados. La base de datos solo es accesible desde la red interna y no está expuesta al exterior.



- Plan de medidas técnicas y organizativas de seguridad.

A continuación, se resumen las medidas implementadas y recomendadas:

Medidas técnicas

- Uso de contenedores Docker para aislar servicios.
- Definición de redes privadas en Docker Compose para separar servicios internos y públicos.
- Configuración de NGINX como proxy inverso para exponer únicamente el backend.
- Centralización de logs mediante Loki.
- Supervisión del sistema con Prometheus y Grafana.
- Acceso cifrado a base de datos y uso de variables de entorno para credenciales.
- Validaciones en el backend y gestión de sesiones con JWT.

Medidas organizativas

- Uso de control de versiones (Git) para trazabilidad.
- Almacenamiento de variables sensibles en .env (no subido al repositorio).
- Prácticas de mínimos privilegios en el backend.
- Políticas de revisión de código para minimizar errores humanos.
- Planificación de backups y procedimiento de restauración (simulado).
- Separación de entornos de desarrollo y producción.

2. Infraestructura y despliegue seguro

- Infraestructura con Docker Compose o Terraform.
- Proxy inverso con NGINX o Traefik.
- HTTPS/TLS con Let's Encrypt o similar.
- Gestión segura de secretos.
- Hardening: puertos, firewall, logs, accesos, actualizaciones.

Se ha diseñado una infraestructura segura usando Docker Compose con los siguientes servicios:

- FastAPI (backend) con autenticación y lógica de negocio.
- PostgreSQL en red privada.
- NGINX como proxy inverso y terminador SSL.
- Prometheus, Grafana y Loki para observabilidad.

2.1 Infraestructura con Docker Compose

El archivo `docker-compose.yml` define y conecta todos los servicios a través de redes internas, separando acceso público del interno.

En esta imagen se muestra el archivo `docker-compose.yml`, donde se define la infraestructura del proyecto. Cada servicio está descrito con su imagen base, variables, puertos y volúmenes correspondientes.

Destacan las redes:

- `internal`: conecta backend, base de datos, monitorización y logs.
- `public`: expone únicamente el contenedor `nginx`, actuando como puerta de entrada.

Esta organización permite separar claramente la red de servicios internos de los puntos de acceso públicos, reduciendo la superficie de ataque.

```

GNU nano 8.4                                     docker-compose.yml
Version: '3.9'

services:

  # Servicio backend con FastAPI
  backend:
    build: ./backend
    container_name: backend
    restart: always
    env_file:
      - ./backend/.env
    ports:
      - "8000:8000"
    depends_on:
      - db
    networks:
      - internal

  # Base de datos PostgreSQL
  db:
    image: postgres:15
    container_name: postgres
    restart: always
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_USER: fastapi_user
      POSTGRES_PASSWORD: strongpassword
      POSTGRES_DB: fastapi_db
    networks:
      - internal

  # NGINX como proxy inverso
  nginx:
    image: nginx:latest
    container_name: nginx
    restart: always
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl:/etc/nginx/ssl
    ports:
      - "80:80"
      - "443:443"
    depends_on:
      - backend
    networks:
      - internal
      - public

  # Prometheus para métricas
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    restart: always
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"
    networks:
      - internal

  # Grafana para visualización
  grafana:
    image: grafana/grafana
    container_name: grafana
    restart: always
    ports:
      - "3000:3000"
    volumes:
      - grafana-storage:/var/lib/grafana
    networks:
      - internal

  # Loki para logs
  loki:
    image: grafana/loki:2.9.1
    container_name: loki
    restart: always
    volumes:
      - ./monitoring/loki-config.yml:/etc/loki/local-config.yaml
      - ./monitoring/loki-config.yml:/etc/loki/local-config.yaml
    command: -config.file=/etc/loki/local-config.yaml
    ports:
      - "3100:3100"
    networks:
      - internal

volumes:
  pgdata:
  grafana-storage:

networks:
  internal:
    driver: bridge
  public:
    driver: bridge

```

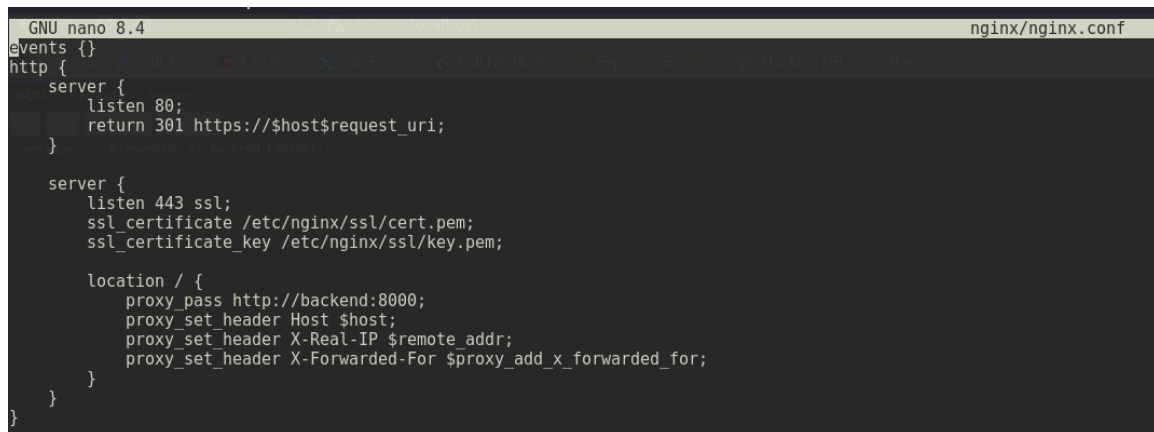
2.2 Proxy inverso con NGINX

NGINX actúa como punto de entrada. Redirige el tráfico HTTP/HTTPS al backend y aplica políticas de cabecera seguras.

En esta imagen se muestra la configuración de NGINX como proxy inverso. El archivo nginx.conf incluye una redirección automática de HTTP a HTTPS, y configura el uso de certificados autofirmados.

Además, define el proxy_pass hacia el servicio backend, incluyendo cabeceras de seguridad estándar como X-Real-IP y X-Forwarded-For, necesarias para mantener trazabilidad de peticiones.

Esta configuración permite centralizar el acceso a todos los servicios en un único punto controlado y cifrado, alineándose con prácticas de arquitectura segura.



```
GNU nano 8.4 nginx/nginx.conf
events {}
http {
    server {
        listen 80;
        return 301 https://$host$request_uri;
    }

    server {
        listen 443 ssl;
        ssl_certificate /etc/nginx/ssl/cert.pem;
        ssl_certificate_key /etc/nginx/ssl/key.pem;

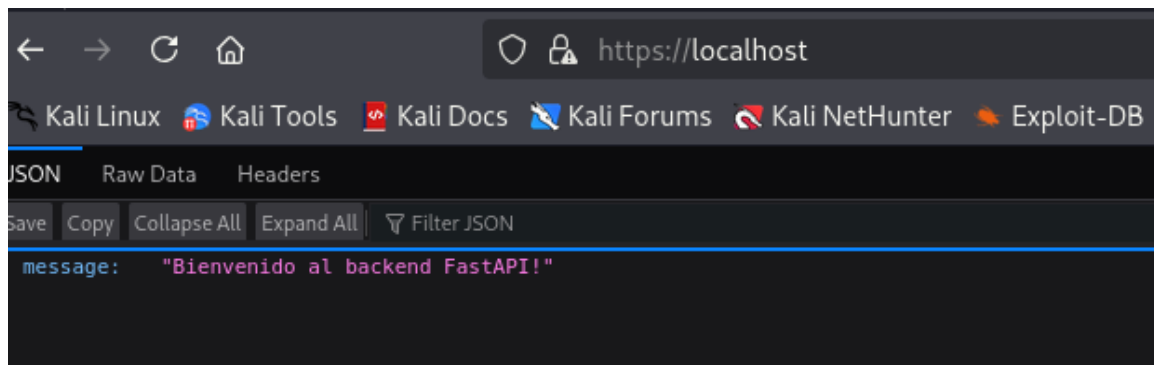
        location / {
            proxy_pass http://backend:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
}
```

2.3 HTTPS/TLS con certificado autofirmado

Esta captura muestra la terminal en el momento en que se ejecuta el comando openssl para generar un certificado autofirmado. Este certificado se utilizará en el contenedor NGINX para habilitar HTTPS, permitiendo cifrar las comunicaciones cliente-servidor en este entorno local de pruebas. Se puede observar la ruta de salida y los parámetros usados para evitar el uso de contraseñas interactivas.


```
--(venv)--(root@kali): ~/proyecto_pps
# mkdir -p nginx/ssl
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
-keyout nginx/ssl/key.pem -out nginx/ssl/cert.pem \
-subj "/C=ES/ST=Valencia/L=Valencia/O=DevSecOps/CN=localhost"
```

Aquí se visualiza el acceso a la aplicación a través del navegador, utilizando `https://localhost`. El aviso de seguridad que aparece es esperado, ya que el certificado es autofirmado y no ha sido emitido por una autoridad de certificación (CA) confiable. Esta captura demuestra que HTTPS está funcional, aunque sea con un certificado generado localmente.



Esta imagen muestra la configuración personalizada del archivo `nginx.conf`, donde se incluyen las instrucciones necesarias para habilitar TLS/SSL en el contenedor. Se evidencia la redirección de HTTP a HTTPS, la declaración de certificados y la redirección del tráfico hacia el servicio backend en FastAPI.

```
GNU nano 8.4 nginx/nginx.conf
events {}
http {
    server {
        listen 80;
        return 301 https://$host$request_uri;
    }

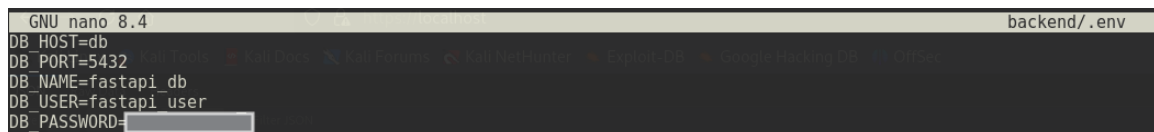
    server {
        listen 443 ssl;
        ssl_certificate /etc/nginx/ssl/cert.pem;
        ssl_certificate_key /etc/nginx/ssl/key.pem;

        location / {
            proxy_pass http://backend:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
}
```

2.4 Gestión segura de secretos

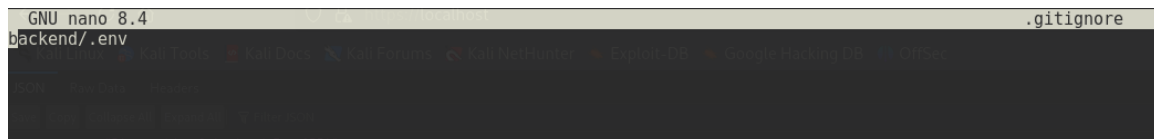
Para la gestión segura de secretos se ha usado un archivo `.env` que almacena las credenciales del backend. Este archivo no se incluye en el repositorio gracias a `.gitignore`. La aplicación accede a estas variables mediante `os.environ.get()` o utilizando la librería `python-dotenv`. Se propone como mejora futura el uso de HashiCorp Vault o cifrado GPG

En esta captura se muestra parte del archivo `.env` (con las credenciales ocultas). En él se definen variables sensibles como el usuario y contraseña de la base de datos, separados del código para cumplir con buenas prácticas de gestión de secretos.



```
GNU nano 8.4                                backend/.env
DB_HOST=db
DB_PORT=5432
DB_NAME=fastapi_db
DB_USER=fastapi_user
DB_PASSWORD=
```

Se presenta el contenido del `.gitignore`, donde se especifica que el archivo `.env` no debe subirse al repositorio. Esto protege la información sensible del backend, evitando fugas accidentales de credenciales a través del control de versiones.



```
GNU nano 8.4                                .gitignore
backend/.env
```

2.5 Hardening aplicado

- Red internal para servicios sensibles.
- Exposición de puertos limitada a lo estrictamente necesario.
- Logs gestionados por Loki y visualizados en Grafana.
- Backend ejecutado como usuario sin privilegios (USER appuser).
- Actualización de contenedores con `docker-compose pull && up`.

Todos los contenedores se basan en imágenes actualizadas latest, y se reconstruyen con `docker-compose pull && up`. Para entornos de producción se recomienda definir versiones exactas y usar un sistema de actualización controlado.

Esta captura muestra la línea incluida al final del Dockerfile del servicio backend: USER appuser. Con esta configuración, el contenedor no ejecuta procesos como root, lo que ayuda a prevenir escaladas de privilegios y

ataques que exploten el contexto del sistema operativo anfitrión. Esta medida forma parte del hardening de contenedores según las recomendaciones de seguridad de Docker y OWASP.

```
GNU nano 8.4 backend/Dockerfile
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

RUN adduser --disabled-password --gecos '' appuser
USER appuser
```

Se ha verificado mediante `docker inspect` que el servicio backend no se ejecuta con privilegios de root. En el Dockerfile se crea un usuario llamado `appuser` para este propósito.

Otros servicios como Prometheus, Grafana y Loki también utilizan usuarios no privilegiados integrados en sus respectivas imágenes, como `nobody` o `10001`, asegurando que no existan permisos innecesarios en los procesos del contenedor.

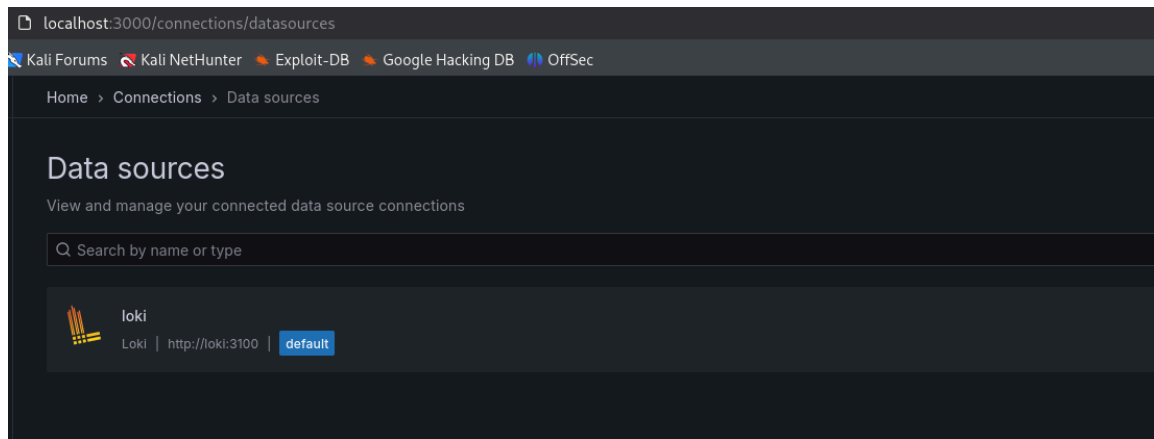
```
(venv)-(root@kali) - [/proyecto_pps]
# docker inspect $(docker ps -q) --format '{{.Config.User}} {{.Name}}'
/nginx
appuser /backend
472 /grafana
/postgres
10001 /loki
nobody /prometheus

(venv)-(root@kali) - [/proyecto_pps]
#
```

Esta captura muestra cómo los servicios críticos como backend, postgres, prometheus, grafana y loki están conectados a una red Docker interna. Esto impide el acceso directo desde el exterior, aplicando el principio de aislamiento de red. El único servicio conectado también a una red pública es nginx, que actúa como proxy controlado.

```
networks:
  internal:
    driver: bridge
  public:
    driver: bridge
```

Esta captura muestra la interfaz de Grafana con Loki añadido como fuente de datos y mostrando logs de alguno de los contenedores (por ejemplo, backend). Esta visualización es evidencia de que los logs se están centralizando y almacenando correctamente, permitiendo una posterior auditoría, visualización o generación de alertas en tiempo real.



3. Desarrollo seguro + CI/CD

Durante esta fase se ha trabajado en el desarrollo seguro de un backend con FastAPI, implementando medidas de protección básicas, validaciones de entrada, y autenticación basada en tokens JWT.

3.1 Desarrollo seguro

El backend expone varios endpoints protegidos y seguros. Se utiliza la librería Pydantic para validar automáticamente la entrada de datos en el endpoint `/items`, donde se exige que los datos cumplan ciertas restricciones (por ejemplo, nombre con longitud mínima y precio mayor que cero). Esto evita errores lógicos, inyecciones de datos maliciosos y refuerza la robustez de la API.

Además, se implementa un sistema de autenticación robusto con OAuth2PasswordBearer y tokens JWT usando la librería `python-jose`. El usuario puede autenticarse a través de `/token`, recibiendo un JWT válido que luego permite acceder a rutas protegidas como `/me`.

Las contraseñas están protegidas mediante `bcrypt` con `passlib`, lo que garantiza que incluso en caso de filtración de la base de datos, las contraseñas no puedan ser utilizadas directamente.

3.2 Testing automático

Se han desarrollado pruebas unitarias con `pytest`, agrupadas en el archivo `backend/tests/test_main.py`. Estas pruebas verifican el correcto comportamiento del backend ante entradas válidas e inválidas, permitiendo detectar rápidamente errores tras cualquier modificación. Gracias a `TestClient`, se puede simular el uso real de la API directamente desde el entorno de testing.

3.3 Pipeline CI/CD

Para automatizar la calidad y la seguridad, se ha definido un workflow en GitHub Actions (`.github/workflows/ci.yml`) que realiza:

- Instalación de dependencias (FastAPI, Uvicorn, Pytest, Bandit, Trivy...).

- **Análisis estático de seguridad** con bandit, que detecta malas prácticas o patrones de riesgo.
- **Pruebas automatizadas** con pytest, garantizando que el backend mantiene su funcionalidad.
- **Escaneo de contenedores Docker** con trivy, identificando posibles vulnerabilidades conocidas (CVEs) en las imágenes Docker generadas.

El pipeline se ejecuta automáticamente en cada push o pull request, asegurando la validación continua del código.

3.4 Seguridad en dependencias

Opcionalmente, se puede incluir safety para escanear vulnerabilidades en las librerías Python instaladas a partir del requirements.txt.

Se ha definido un archivo ci.yml dentro de .github/workflows, que automatiza todo el proceso de validación, testing y análisis de seguridad cada vez que se hace un push o pull request. Esto permite implementar una cultura DevSecOps en el proyecto, asegurando la calidad antes del despliegue.

3.5 Objetivo del workflow:

- Garantizar que no se introduce código inseguro.
- Detectar vulnerabilidades antes de llegar a producción.
- Asegurar que los tests automatizados se ejecutan correctamente.

```
GNU nano 8.4 .github/workflows/ci.yml
name: CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: 3.11

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r backend/requirements.txt
          pip install pytest bandit trivy

      - name: Lint y seguridad (Bandit)
        run: bandit -r backend

      - name: Pruebas unitarias (simulado)
        run: echo "pytest backend (añadir test si hay tiempo)"

      - name: Escaneo de imagen Docker con Trivy
        run: |
          docker build -t myapp backend
          trivy image --severity HIGH,CRITICAL myapp || true
```

Se ha implementado un conjunto de pruebas con pytest que simulan peticiones al backend con datos válidos e inválidos. Esto permite garantizar que la validación con Pydantic funciona correctamente y que los endpoints se comportan como se espera.

```
---(venv)--(root@kali): /proyecto_pps
--# docker-compose run backend bash

Creating proyecto_pps backend_run ... done
appuser@939398ee49b5:/apps python -m pytest tests/test_main.py

===== test session starts =====
platform linux -- Python 3.11.12, pytest-8.3.5, pluggy-1.6.0
rootdir: /app
plugins: anyio-4.9.0
collected 3 items

tests/test_main.py ... [100%]

===== warnings summary =====
../usr/local/lib/python3.11/site-packages/passlib/utils/_init_.py:854
../usr/local/lib/python3.11/site-packages/passlib/utils/_init_.py:854: DeprecationWarning: 'crypt' is deprecated and slated for removal in Python 3.13
  from crypt import crypt as _crypt

../usr/local/lib/python3.11/site-packages/pytest/cache/provider.py:475
../usr/local/lib/python3.11/site-packages/pytest/cache/provider.py:475: PytestCacheWarning: could not create cache path /app/.pytest_cache/v/cache/nodeids: [Errno 13] Permission denied: '/app/pytest-cache-files-pwzq5ix9'
  config.cache.set('cache/nodeids', sorted(self.cached_nodeids))

../usr/local/lib/python3.11/site-packages/pytest/stepwise.py:51
../usr/local/lib/python3.11/site-packages/pytest/stepwise.py:51: PytestCacheWarning: could not create cache path /app/.pytest_cache/v/cache/stepwise: [Errno 13] Permission denied: '/app/pytest-cache-files-o-sm-etnk'
  session.config.cache.set(STEPWISE_CACHE_DIR, [])

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== 3 passed, 3 warnings in 0.61s =====
appuser@939398ee49b5:/apps
```

Detalle de los tests:

- test_create_item: prueba con un nombre válido y precio correcto.
- test_create_item_invalid: prueba fallida con nombre muy corto y precio negativo.
- test_auth: autenticación válida con JWT y recuperación de datos con /me.

Este endpoint es un ejemplo de programación defensiva. Solo se aceptan datos bien formados gracias a Pydantic, lo cual mitiga problemas de inyecciones, errores de lógica o abuso.

```
GNU nano 8.4 backend/main.py
class Item(BaseModel):
    name: str = Field(..., min_length=3)
    price: float = Field(..., gt=0)

@app.post("/items")
def create_item(item: Item):
    return {"nombre": item.name, "precio": item.price}
```

Se ha implementado un sistema de login basado en JWT:

1. El usuario envía sus credenciales a /token usando form-data.
2. Recibe un access_token firmado.
3. Con ese token, puede acceder a /me, que está protegida por Depends(oauth2_scheme).

Backend/main.py:

```
# ENDPOINTS
@app.post("/token", response_model=Token)
def login(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(form_data.username, form_data.password)
    if not user:
        raise HTTPException(status_code=401, detail="Credenciales inválidas")
    token = create_access_token({"sub": user.username})
    return {"access_token": token, "token_type": "bearer"}

@app.get("/me", response_model=User)
def read_users_me(token: str = Depends(oauth2_scheme)):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise HTTPException(status_code=401, detail="Token inválido")
        user = fake_users_db.get(username)
        return User(**user)
    except JWTError:
        raise HTTPException(status_code=401, detail="Token inválido")
```


4. Seguridad operativa

- Centralización de logs (Loki, ELK).
- Monitorización (Prometheus + Grafana).
- Alertas de eventos anómalos.
- Simulación de backup cifrado.
- Políticas de mínimos privilegios y autenticación.

En este apartado se han implementado mecanismos para proteger la aplicación cuando ya está en producción. A diferencia de las fases anteriores (centradas en desarrollo o despliegue), aquí se aplican técnicas de monitorización, detección de eventos anómalos, centralización de logs y control de accesos, lo cual permite prevenir, detectar y responder ante posibles incidentes de seguridad. Además, se simula el proceso de backup seguro mediante cifrado, aportando resiliencia y trazabilidad al sistema.

El enfoque se basa en buenas prácticas DevSecOps, combinando observabilidad (con Prometheus, Grafana y Loki), principios de seguridad (como mínimo privilegio) y medidas proactivas para mantener el control del entorno en ejecución.

—

4.1 Centralización de logs con Loki

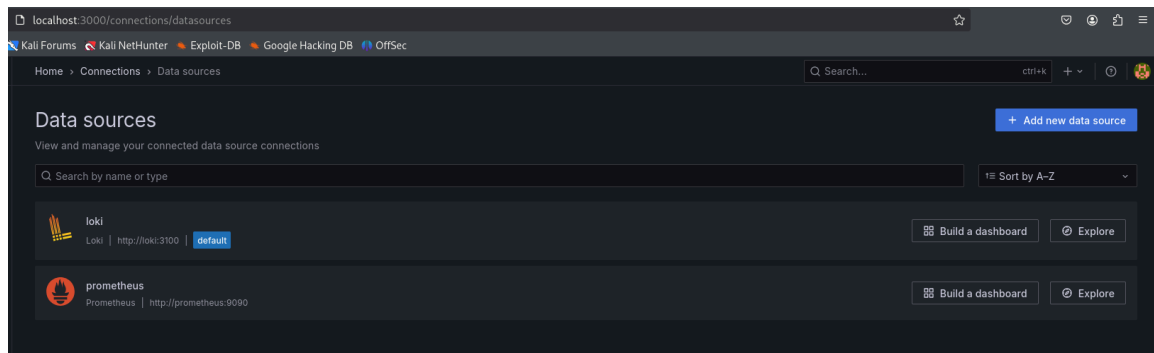
Se ha desplegado el servicio de logging centralizado Loki, que recopila automáticamente los logs generados por los contenedores del proyecto (como el backend, NGINX o base de datos).

Esto permite consultar de forma unificada los eventos de la aplicación, detectar errores, y tener trazabilidad de accesos, fallos y acciones realizadas por los usuarios o servicios.

4.2 Monitorización con Prometheus + Grafana

Prometheus recolecta métricas del sistema, contenedores o servicios expuestos en /metrics, y Grafana permite visualizarlas mediante dashboards. Se ha desplegado un sistema de monitorización compuesto por Prometheus y Grafana. Prometheus recolecta métricas como uso de CPU, tráfico HTTP, o estado de los servicios. Estas métricas son consultadas desde Grafana para su visualización en dashboards personalizables.

La configuración de Prometheus se encuentra en el archivo `monitoring/prometheus.yml`, donde se definen los endpoints a monitorizar. Grafana se conecta automáticamente a Prometheus como fuente de datos, permitiendo realizar consultas visuales y detectar anomalías.



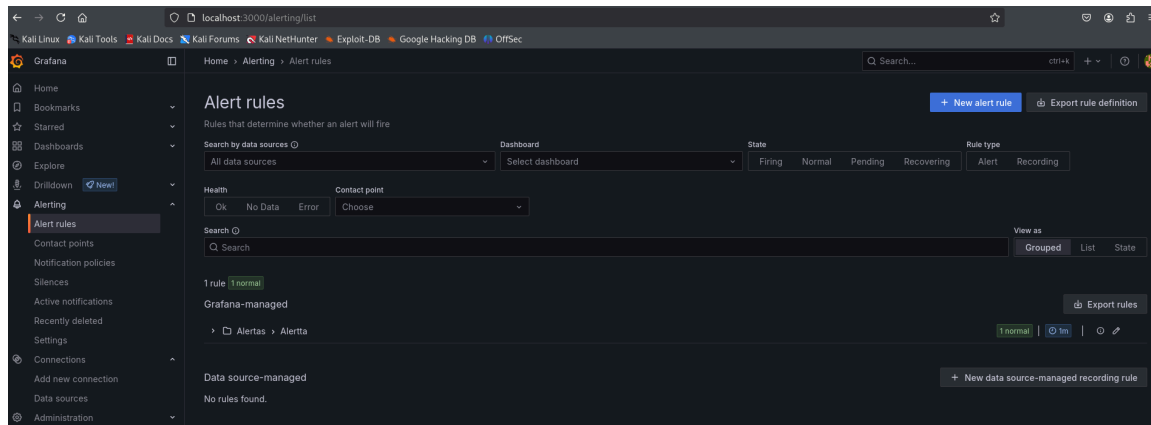
4.3 Detección de eventos anómalos (alertas)

Grafana permite configurar reglas de alerta para detectar comportamientos anómalos en tiempo real, como un aumento de errores HTTP, consumo excesivo de CPU o interrupciones en servicios. Aunque aún no se han configurado alertas personalizadas, el sistema está preparado para ello. Un ejemplo de alerta posible sería:

"Detectar más de 1 error HTTP 500 por minuto."
`rate(http_requests_total{status="500"}[1m]) > 1`

¿Qué hace?

Permite lanzar alertas ante situaciones inusuales: errores 500, uso anómalo de recursos, etc.



4.4 Control de accesos y privilegios mínimos

¿Qué significa?

Aplicar el principio de mínimo privilegio: cada componente debe tener solo los permisos necesarios. Esto aplica tanto a contenedores como a usuarios humanos.

Siguiendo el principio de mínimo privilegio, el servicio backend se ejecuta con un usuario no privilegiado (appuser). Esto reduce la superficie de ataque en caso de que un contenedor sea comprometido.

Además, cada servicio corre aislado en su propia red y sin puertos expuestos innecesariamente. En futuras versiones se puede añadir autenticación en Prometheus y Grafana y usar OAuth u otros métodos para proteger el acceso.

```
(venv)-(root@kali) - [/proyecto_pps]
# docker inspect backend --format '{{.Config.User}}'
appuser

(venv)-(root@kali) - [/proyecto_pps]
#
```

```
(venv)-(root@kali)-[/proyecto_pps]
# cat backend/Dockerfile
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
RUN adduser --disabled-password --gecos '' appuser
USER appuser

(venv)-(root@kali)-[/proyecto_pps]
#
```

4.5 Simulación de copia de seguridad cifrada

Se ha implementado un script de simulación de copia de seguridad segura. El script genera una exportación de la base de datos PostgreSQL, la comprime y la cifra utilizando openssl con AES-256. El archivo resultante puede almacenarse en una ubicación segura y está protegido frente a accesos no autorizados.

4.6 Copia de seguridad cifrada (simulada)

Como parte de las medidas de seguridad operativa, se ha desarrollado un script de copia de seguridad llamado backup.sh. Este simula el procedimiento seguro que se seguiría en un entorno real:

- Se realiza un volcado (dump) de la base de datos PostgreSQL directamente desde el contenedor Docker postgres.
- El archivo resultante es comprimido en formato tar.gz.
- A continuación, se cifra usando el algoritmo AES-256-CBC con una clave de prueba (secret123).
- Finalmente, se eliminan los archivos intermedios, dejando únicamente el archivo cifrado (backup_YYYY-MM-DD.tar.gz.enc) en el sistema.

Este proceso permite simular la protección de información sensible en backups y sirve como base para sistemas reales que almacenan las copias en almacenamiento externo o encriptado en la nube.

4.7 Comando de cifrado utilizado:

openssl enc -aes-256-cbc -salt -in archivo.tar.gz -out archivo.enc -k secret123

```
(venv)-(root@kali) - [/proyecto_pps]
# cat backup.sh
#!/bin/bash
fecha=$(date +%F)

sql="backup_$fecha.sql"
tar="backup_$fecha.tar.gz"
enc="backup_$fecha.tar.gz.enc"

echo "[+] Exportando base de datos PostgreSQL..."
docker exec postgres pg_dump -U fastapi_user fastapi_db > $sql

echo "[+] Comprimiendo archivo SQL..."
tar -czf $tar $sql

echo "[+] Cifrando archivo comprimido..."
openssl enc -aes-256-cbc -salt -in $tar -out $enc -k secret123

echo "[+] Eliminando archivos intermedios..."
rm $sql $tar

echo "[✓] Copia de seguridad creada: $enc"

(venv)-(root@kali) - [/proyecto_pps]
#
```

```
(venv)-(root@kali) - [/proyecto_pps]
# ./backup.sh
[+] Exportando base de datos PostgreSQL...
[+] Comprimiendo archivo SQL...
[+] Cifrando archivo comprimido...
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[+] Eliminando archivos intermedios...
[✓] Copia de seguridad creada: backup_2025-05-24.tar.gz.enc
```

```
(venv)-(root@kali)-[/proyecto_pps]
# file backup_2025-05-24.tar.gz.enc
backup_2025-05-24.tar.gz.enc: openssl enc'd data with salted password

(venv)-(root@kali)-[/proyecto_pps]
# tree -I venv
.
├── backend
│   ├── Dockerfile
│   ├── main.py
│   ├── requirements.txt
│   └── tests
│       └── test_main.py
├── backup_2025-05-24.tar.gz.enc
├── backup.sh
├── docker-compose.yml
├── monitoring
│   ├── loki
│   │   ├── cache
│   │   ├── chunks
│   │   └── index
│   ├── loki-config.yml
│   └── prometheus.yml
├── nginx
│   ├── nginx.conf
│   └── ssl
│       ├── cert.pem
│       └── key.pem
└── 10 directories, 12 files

(venv)-(root@kali)-[/proyecto_pps]
#
```

4.8 Consideraciones reales en entornos de producción

En este caso, el proceso de copia de seguridad ha sido ejecutado manualmente como parte de una simulación. Sin embargo, en un entorno de producción real, este script debería integrarse en un sistema automatizado, como una tarea programada mediante crontab o una herramienta de orquestación (por ejemplo, Ansible o Kubernetes CronJobs), que realice copias periódicas sin intervención manual.

Además, por motivos de seguridad y resiliencia, el archivo cifrado resultante no debería permanecer en el mismo sistema donde se encuentra la aplicación. Idealmente, las copias se deben transferir a una ubicación externa, como:

- Un almacenamiento remoto seguro (por ejemplo, un servidor SFTP aislado).
- Un bucket cifrado en la nube (AWS S3, Backblaze, etc.).
- Un volumen dedicado en otra máquina (RAID o NAS).

Esto garantiza que, en caso de que el servidor principal sufra un compromiso o un fallo catastrófico, las copias de seguridad se conserven intactas y disponibles para su restauración.

En esta simulación de infraestructura tipo SaaS, estas medidas no se han implementado físicamente, pero se dejan reflejadas para demostrar que se entiende su importancia y que forman parte de las buenas prácticas de seguridad operativa.

Gracias a estas medidas, la infraestructura no solo está monitorizada y bajo control, sino también preparada ante incidentes o fugas de información, con backups cifrados, acceso limitado y alertas listas para ser configuradas.

5. Pruebas de seguridad

Con el sistema en marcha, se han realizado pruebas DAST (Dynamic Application Security Testing) con OWASP ZAP para detectar vulnerabilidades como XSS o CSRF.

Además, se ha escaneado el código y contenedores en busca de fallos de seguridad usando Trivy, Bandit y Safety. También se ha usado Gitleaks para comprobar que no se han subido contraseñas ni tokens sensibles al repositorio. Como parte del plan de respuesta a incidentes, se ha simulado un ataque de fuerza bruta y se han verificado los logs generados.

5.1 Análisis de vulnerabilidades del contenedor backend con Trivy

Durante la fase de seguridad en el ciclo DevSecOps, se ha realizado un análisis de vulnerabilidades sobre la imagen Docker del backend con Trivy, una herramienta de escaneo de contenedores ampliamente usada para detectar vulnerabilidades en los paquetes del sistema operativo y dependencias del software.

Se ha ejecutado el siguiente comando:

trivy image proyecto_pps_backend

Trivy ha identificado múltiples vulnerabilidades en la base de la imagen, correspondiente a Debian 12.11. La mayoría están asociadas a paquetes del sistema operativo, no directamente al código de aplicación, y no todas son explotables en nuestro contexto.

```

[venv]~root@kali:~/projecto pps]
# trivy image projecto pps_backend
2025-05-24T07:55:37-04:00 INFO [vu]dbb Need to update DB
2025-05-24T07:55:37-04:00 INFO [vu]dbb Downloading vulnerability DB...
2025-05-24T07:55:37-04:00 INFO [vu]dbb Downloading artifact... repo="mirror.gcr.io/aquasec/trivy-db:2"
64.28 MiB / 64.28 MiB [-----] 100.00% 917.39 KiB p/s |ml2s
2025-05-24T07:56:50-04:00 INFO [vu]dbb Artifact successfully downloaded repo="mirror.gcr.io/aquasec/trivy-db:2"
2025-05-24T07:56:50-04:00 INFO [vu]dbb Vulnerability scanning is enabled
2025-05-24T07:56:50-04:00 INFO [secret] Secret scanning is enabled
2025-05-24T07:56:50-04:00 INFO [secret] If your scanning is slow, please try "--scanners vuln" to disable secret scanning
2025-05-24T07:56:50-04:00 INFO [secret] Please see also https://trivy.dev/docs/scanner/secrets/secrets#recommendation for faster secret detection
2025-05-24T07:57:54-04:00 INFO [python] Licenses activated from one or more METADATA files may be subject to additional terms. Use "--debug" flag to see all affected packages.
2025-05-24T07:57:54-04:00 INFO Detected OS family="debian" version="12.11"
2025-05-24T07:57:54-04:00 INFO [debian] Detecting vulnerabilities... os version="12" pkg_num=105
2025-05-24T07:57:54-04:00 INFO [python] Number of language-specific files num=1
2025-05-24T07:57:54-04:00 INFO [python-pkg] Detecting vulnerabilities...
2025-05-24T07:57:54-04:00 WARN Using severities from other vendors for some vulnerabilities. Read https://trivy.dev/docs/scanner/vulnerability#severity-selection for details.
2025-05-24T07:57:54-04:00 INFO Table result includes only package filenames. Use "--format json" option to get the full path to the package file.

Report Summary

```

Target	Type	Vulnerabilities	Secrets
projecto pps_backend (debian 12.11)	debian	89	-
usr/local/lib/python3.11/site-packages/annotated_types-0.7.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.11/site-packages/anyio-4.9.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.11/site-packages/bcrypt-4.3.0.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.11/site-packages/certifi-2025.4.26.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.11/site-packages/click-8.2.1.dist-info/METADATA	python-pkg	0	-
usr/local/lib/python3.11/site-packages/cdcx-0.10.1.dist-info/METADATA	python-nko	0	-

Resumen del análisis:

Total de vulnerabilidades encontradas: 89

Severidades:

- CRITICAL: 1
- HIGH: 3
- MEDIUM: 17
- LOW: 67
- UNKNOWN: 1

```
proyecto_pps_backend (debian 12.11) > D Aleras > Aleras
Total: 89 (UNKNOWN: 1, LOW: 67, MEDIUM: 17, HIGH: 3, CRITICAL: 1)
```

- La gran mayoría de vulnerabilidades provienen de librerías del sistema (glibc, bash, util-linux, etc.), lo que es habitual al usar imágenes base como python:3.11-slim sin realizar actualización con apt-get upgrade en el Dockerfile, con el apt-get upgrade se solucionan la mayor parte de problemas.
- No se han detectado vulnerabilidades en las dependencias Python de la aplicación, lo cual es positivo.
- Ninguna vulnerabilidad se considera explotable directamente en este entorno (p.ej. no se permite ejecución de ldd, ni existen binarios setuid, ni usuarios maliciosos dentro del contenedor).

Tras escanear la imagen del backend con Trivy, se detectaron un total de 89 vulnerabilidades, incluyendo una crítica y tres de alta gravedad. La mayoría de estas se deben a librerías del sistema operativo base Debian.

En un entorno de producción real, se deberían aplicar las siguientes medidas:

- Usar imágenes base minimalistas (como Alpine o distroless) para reducir superficie de ataque.
- Establecer procesos CI/CD que incluyan escaneos automáticos de imágenes con Trivy o herramientas similares.
- Aplicar control de versiones estricto en dependencias (Python y sistema).
- Realizar actualizaciones y parches de seguridad regulares.
- Aplicar el principio de menor privilegio en usuarios de los contenedores.
- Establecer alertas y monitoreo continuo sobre la seguridad de las imágenes desplegadas.
- Almacenar los resultados de escaneos en reportes automatizados (por ejemplo, integrados en GitHub Actions o GitLab CI).

Dado que este entorno es una simulación de un SaaS, no se aplican todas estas medidas automáticamente. Sin embargo, se ha demostrado el conocimiento necesario para abordarlas en entornos reales.

5.2 Realización de pruebas con herramientas de análisis estático

Se ha utilizado la herramienta Bandit para realizar un análisis estático sobre el backend del proyecto. Esta herramienta examina el código fuente en busca de patrones comunes que pueden derivar en vulnerabilidades o errores de seguridad.

Entre los resultados obtenidos destaca una advertencia sobre el uso de una clave secreta estática en el código fuente (SECRET_KEY), lo cual en un entorno real debe sustituirse por una variable de entorno protegida o un gestor de secretos.

Este análisis demuestra la capacidad del entorno para someter el código a revisiones automatizadas y la conciencia de seguridad en fases tempranas del ciclo de desarrollo.

Comando utilizado:

```
bandit -r backend
```

Resultados obtenidos:

Bandit ha identificado un total de 5 problemas de baja severidad. A continuación se destacan los más relevantes:

1. B105: Hardcoded password
 - Ubicación: backend/main.py, línea 9
 - Descripción: Se detectó una clave secreta estática usada directamente en el código: SECRET_KEY = "secretoparajwt123".
 - Severidad: Low | Confianza: Medium
 - CWE-259: Uso de contraseñas embebidas en código fuente.
 - Solución recomendada: Mover la clave a una variable de entorno usando un archivo .env o secrets manager.
2. B101: Uso de assert en producción
Ubicación: backend/tests/test_main.py (líneas 8, 12, 13, 17)

Se ha detectado el uso de la palabra clave assert. Aunque es válida para pruebas unitarias, debe evitarse en entornos productivos ya que puede eliminarse al compilar con optimización (-O).

- Solución: Sustituir asserts por sentencias de prueba formales con un framework de testing si fuera a usarse en producción.

```
(venv)-(root@kali)-[/proyecto_pps]
# bandit -r backend
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.13.3
Run started:2025-05-24 13:30:01.396174

Test results:
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'secretoparajwt123'
Severity: Low Confidence: Medium
CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b105_hardcoded_password_string.html
Location: backend/main.py:9:13
8 # Secretos (esto debería ir en .env)
9 SECRET_KEY = "secretoparajwt123"
10 ALGORITHM = "HS256"

>> Issue: [B101:assert_used] Use of assert detected. The enclosed code will be removed when compiling to optimised byte code.
Severity: Low Confidence: High
CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b101_assert_used.html
Location: backend/tests/test_main.py:8:4
7 response = client.get("/")
8 assert response.status_code == 404 # No hay ruta raíz
9

>> Issue: [B101:assert_used] Use of assert detected. The enclosed code will be removed when compiling to optimised byte code.
Severity: Low Confidence: High
CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
More Info: https://bandit.readthedocs.io/en/1.7.10/plugins/b101_assert_used.html
Location: backend/tests/test_main.py:12:4
11 response = client.post("/items", json={"name": "Teclado", "price": 35.99})
12 assert response.status_code == 200
13 assert response.json()["nombre"] == "Teclado"

Code scanned:
Total lines of code: 74
Total lines skipped (#nosec): 0

Run metrics:
Total issues (by severity):
Undefined: 0
Low: 5
Medium: 0
High: 0
Total issues (by confidence):
Undefined: 0
Low: 0
Medium: 1
High: 4
Files skipped (0):

(venv)-(root@kali)-[/proyecto_pps]
```

Aunque los problemas detectados no son críticos, demuestran la utilidad del análisis estático como parte del ciclo DevSecOps. En un entorno real, se automatizaría esta revisión en una pipeline de CI/CD para evitar que se introduzcan estas malas prácticas en producción.

5.3 Gestión de secretos / .env

Antes, las claves sensibles como el SECRET_KEY se encontraban directamente en el código fuente, estaba la contraseña en texto claro y en el main.py:

```
(venv)-(root@kali)-[/proyecto_pps]
# cat backend/main.py
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel, Field
from typing import Optional

# Secretos (esto debería ir en .env)
SECRET_KEY = "!"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

Esta práctica es insegura, ya que compromete la confidencialidad si el repositorio es expuesto. Para solucionar esto, se implementó la carga de variables desde un archivo .env usando la librería dotenv:

```
GNU nano 8.4 backend/main.py *
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel, Field
from typing import Optional
import os
from dotenv import load_dotenv

load_dotenv()
SECRET_KEY = os.getenv("SECRET_KEY")
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

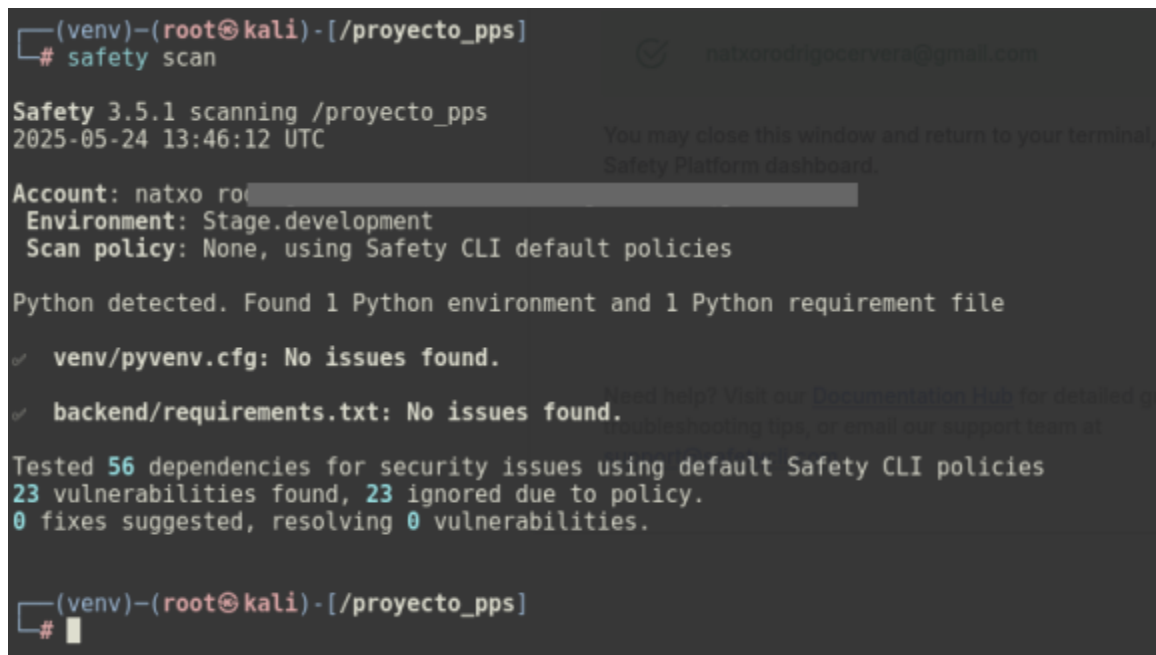
Este enfoque mejora la seguridad, permitiendo que el valor real esté separado del código. El archivo .env no se sube al repositorio (está incluido en .gitignore). En un entorno real, estos secretos deberían gestionarse mediante herramientas específicas como Docker secrets, AWS Parameter Store, HashiCorp Vault o el uso de orquestadores como Kubernetes con secretos cifrados.

5.4 Análisis de dependencias con Safety

Se utilizó la herramienta Safety para analizar vulnerabilidades conocidas en las dependencias Python declaradas en requirements.txt. Esta herramienta compara los paquetes instalados contra una base de datos de vulnerabilidades de seguridad (CVEs).

El análisis se ejecutó con:

```
safety scan
```



```
(venv)-(root@kali)-[/proyecto_pps]
# safety scan

Safety 3.5.1 scanning /proyecto_pps
2025-05-24 13:46:12 UTC

Account: natxo ro
Environment: Stage.development
Scan policy: None, using Safety CLI default policies

Python detected. Found 1 Python environment and 1 Python requirement file

✓ venv/pyvenv.cfg: No issues found.
✓ backend/requirements.txt: No issues found.

Tested 56 dependencies for security issues using default Safety CLI policies
23 vulnerabilities found, 23 ignored due to policy.
0 fixes suggested, resolving 0 vulnerabilities.

(venv)-(root@kali)-[/proyecto_pps]
#
```

En este entorno no se encontraron vulnerabilidades críticas, pero en un entorno profesional, este análisis se integraría dentro del pipeline de integración continua (CI/CD), permitiendo abortar despliegues si se detectan versiones afectadas por fallos críticos o exploits conocidos.

5.5 Puertos, usuarios y seguridad en contenedores

Para reducir la superficie de ataque y aplicar el principio de mínimo privilegio, se aplicaron las siguientes medidas:

- Solo se exponen los puertos necesarios en docker-compose (8000 para el backend, 443 para nginx, etc.).
- El contenedor del backend se ejecuta como un usuario sin privilegios (appuser), tal como se configura con:

```
(venv)-(root@kali) - [/proyecto_pps]
# docker inspect backend --format '
Puertos: {{range $p, $conf := .NetworkSettings.Ports}}{{ $p }} {{end}}
Usuario: {{.Config.User}}'

Puertos:
Usuario: appuser

(venv)-(root@kali) - [/proyecto_pps]
# docker inspect nginx --format '
Puertos: {{range $p, $conf := .NetworkSettings.Ports}}{{ $p }} {{end}}
Usuario: {{.Config.User}}'

Puertos: 443/tcp 80/tcp
Usuario:

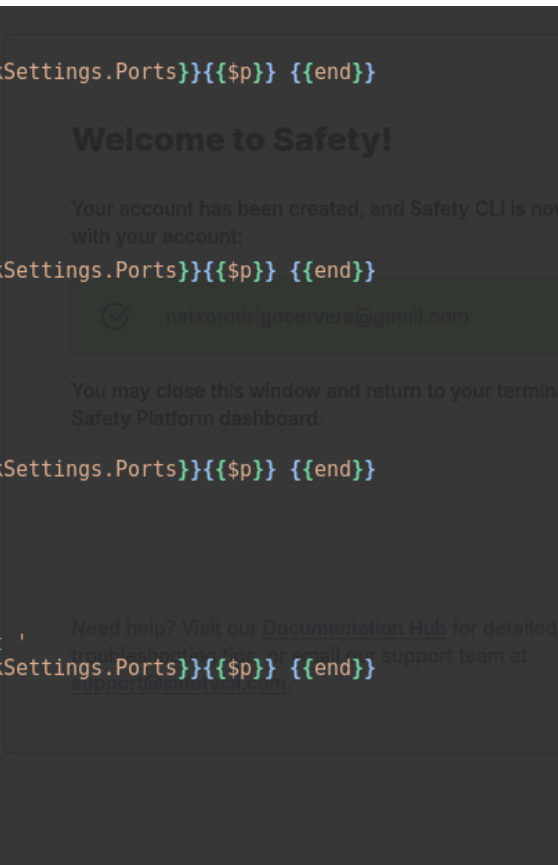
(venv)-(root@kali) - [/proyecto_pps]
# docker inspect loki --format '
Puertos: {{range $p, $conf := .NetworkSettings.Ports}}{{ $p }} {{end}}
Usuario: {{.Config.User}}'

Puertos:
Usuario: 10001

(venv)-(root@kali) - [/proyecto_pps]
# docker inspect prometheus --format '
Puertos: {{range $p, $conf := .NetworkSettings.Ports}}{{ $p }} {{end}}
Usuario: {{.Config.User}}'

Puertos: 9090/tcp
Usuario: nobody

(venv)-(root@kali) - [/proyecto_pps]
#
```

A screenshot of a terminal window. The left pane shows a series of Docker inspect commands for containers named 'backend', 'nginx', 'loki', and 'prometheus'. Each command uses a custom format string to display the exposed ports and the user the container runs as. The results show 'backend' and 'loki' running as 'appuser', 'nginx' as root (no user specified), and 'prometheus' as 'nobody'. The right pane shows a 'Welcome to Safety!' message from the Safety CLI, indicating an account has been created and providing instructions on how to close the window and return to the terminal.

- Todos los servicios están en redes internas de Docker, excepto aquellos explícitamente expuestos.
- En producción, sería recomendable aplicar perfiles de seguridad como seccomp o AppArmor, o utilizar rootless containers para una protección más avanzada.

Con estas medidas se refuerza la seguridad en el despliegue de los contenedores, especialmente frente a escaladas de privilegios o acceso no autorizado.

Despliegue del entorno

El proyecto se encuentra totalmente contenerizado mediante Docker y orquestado con docker-compose. Esto permite que el entorno completo pueda ser desplegado de forma automática y reproducible en cualquier máquina que tenga Docker instalado. Los servicios incluidos son: backend desarrollado en FastAPI, base de datos PostgreSQL, servidor NGINX como proxy inverso, Prometheus y Grafana para monitorización, y Loki para la agregación de logs.

Pasos para desplegar:

1. Clonar el repositorio:

```
git clone https://github.com/usuario/proyecto-pps.git  
cd proyecto-pps
```

2. Ejecutar los contenedores:

```
docker-compose up -d --build
```

3. Acceder a los servicios:

- Backend API: <http://localhost:8000/docs> (Swagger)
- Frontend NGINX: <https://localhost>
- Grafana: <http://localhost:3000>
 - Usuario: admin
 - Contraseña: admin
- Prometheus: <http://localhost:9090>
- Loki (API de logs): <http://localhost:3100>

La aplicación incluye un script de copia de seguridad (backup.sh) para la base de datos, que genera archivos cifrados. Además, se han integrado herramientas de seguridad como Trivy y Bandit para análisis de vulnerabilidades.

REFLEXION FINAL

Después de todo el proceso de este proyecto, la verdad es que he aprendido mucho más de lo que esperaba al principio. No solo por la parte técnica, sino también por todo lo que implica montar una infraestructura que funcione bien, que sea segura y que se pueda mantener. Al principio parecía “solo” montar un backend en Docker y listo, pero cuando empiezas a integrar logs, métricas, seguridad, backup y demás, te das cuenta de que una app es solo una pieza de un puzle mucho más grande.

Una de las cosas que más me ha servido es entender la importancia de la seguridad operacional. Por ejemplo, aprender a utilizar herramientas como Trivy o Bandit me ha enseñado que por muy limpio que creas que está tu código, siempre hay algún detallito que puede mejorarse. El escaneo de vulnerabilidades fue revelador: muchas de las dependencias tienen fallos que ni te imaginas, y aunque algunos no sean críticos, está bien saberlo y dejar constancia. Lo mismo con Bandit, que me avisó de cosas como claves hardcoded o uso de asserts que antes ni me planteaba como un problema real.

Y sí, cometí errores. De hecho, bastantes. Uno muy claro fue meter directamente la clave secreta JWT en el código. Me di cuenta después con el análisis de Bandit, y la solución fue sacarlo a variables de entorno y asegurarme de que nada sensible está versionado ni expuesto. También tuve problemas con los logs de Loki al principio; pensaba que todo funcionaba bien solo por estar montado, pero luego entendí que configurar correctamente las rutas, drivers y formato de los logs es clave para que realmente puedas ver algo útil en Grafana. También me atasqué con el NGINX y los certificados SSL, hasta que me di cuenta de que el navegador me estaba bloqueando por tema de certificados autofirmados. Pequeños fallos que, aunque frustrantes, me ayudaron a fijar conceptos de verdad.

Si tuviera que aplicar todo esto en un entorno real, lo haría con mucho más cuidado desde el principio. Planificaría mejor la estructura del proyecto, sobre todo en cuanto a gestión de secretos, usuarios y permisos. En un entorno real, no montaría los contenedores como root ni permitiría puertos abiertos al tuntún. También automatizaría los backups y me aseguraría de que se almacenan en un sistema externo, cifrados, con

rotación y alertas si fallan. Y lo más importante: todo lo que sea auditoría y monitorización lo tendría desde el primer momento, no como un añadido del final. Porque si algo aprendí con Loki y Grafana es que no puedes solucionar un problema que no puedes ver.

Este proyecto ha sido una buena forma de ensuciarse las manos con herramientas reales, de esas que usan en empresas de verdad. Me ha servido para dejar de ver Docker y FastAPI como herramientas separadas y empezar a verlas como parte de un sistema completo, donde todo está conectado y cada decisión cuenta. Aunque haya tenido errores por el camino, creo que es justo ahí donde más se aprende. Y aunque no sea una app de producción real, la forma de enfocarla ha sido como si lo fuera. Al final eso es lo que te da soltura, equivocarte ahora para acertar luego cuando importe de verdad.