# Análisis estático de seguridad de la aplicación web “Buscador Rick y Morty”

---

**Versión 1.0.0**

**01/2026**

**Pedro Garvich**

**Proyecto: Mini SSDLC aplicado a sistemas web**

# Índice

**1. Informe ejecutivo 3**
**2. Resumen de análisis estático 4**
**3. Detalle de hallazgos 7**
SEC-01 Exposición de clave secreta de Django en código fuente 7
SH-01 Métodos HTTP seguros y no seguros en las vistas de Django 8
SH-02 Configuración DEBUG habilitada en Django 11
SG-01 Uso de datos sin sanitizar desde request POST tras validación de formulario 13
**3. Resumen de remediaciones recomendadas 15**

# 1. Informe ejecutivo

Se realizó un análisis estático de seguridad y calidad de código sobre una aplicación web desarrollada en **Django**, de arquitectura monolítica, orientada a la gestión y visualización de información mediante vistas basadas en templates HTML.

La aplicación fue desarrollada con fines educativos, se ejecuta en un entorno local y no se encuentra desplegada en producción ni expuesta públicamente.

El análisis se llevó a cabo utilizando **herramientas de análisis estático automatizado**, principalmente **SonarCloud**, complementadas con **Semgrep**, y se reforzó mediante revisión manual y análisis contextual de los hallazgos reportados.

Como resultado del análisis, se identificó **una vulnerabilidad de seguridad real y de alto impacto**, relacionada con la **exposición de una clave secreta del framework Django (SECRET_KEY) en el código fuente**, lo que compromete mecanismos críticos como la autenticación, la gestión de sesiones y la protección CSRF de la aplicación. Este hallazgo fue confirmado, contextualizado y documentado con su correspondiente propuesta de remediación.

Asimismo, se detectaron múltiples observaciones clasificadas como **Security Hotspots**, **Reliability Issues** y **patrones inseguros de código**. Tras su análisis detallado, se determinó que la mayoría de estos hallazgos:

- Corresponden a configuraciones propias de entornos de desarrollo
- Representan riesgos potenciales no explotables en el contexto actual
- No están vinculados a prácticas de mantenibilidad y calidad de código sin impacto directo en la seguridad.

El análisis complementario con Semgrep permitió identificar **un patrón de uso inseguro de datos provenientes de solicitudes HTTP**, de severidad media y bajo impacto práctico en el estado actual de la aplicación, así como vulnerabilidades teóricas asociadas a dependencias externas, las cuales fueron evaluadas como no alcanzables o informativas en este contexto.

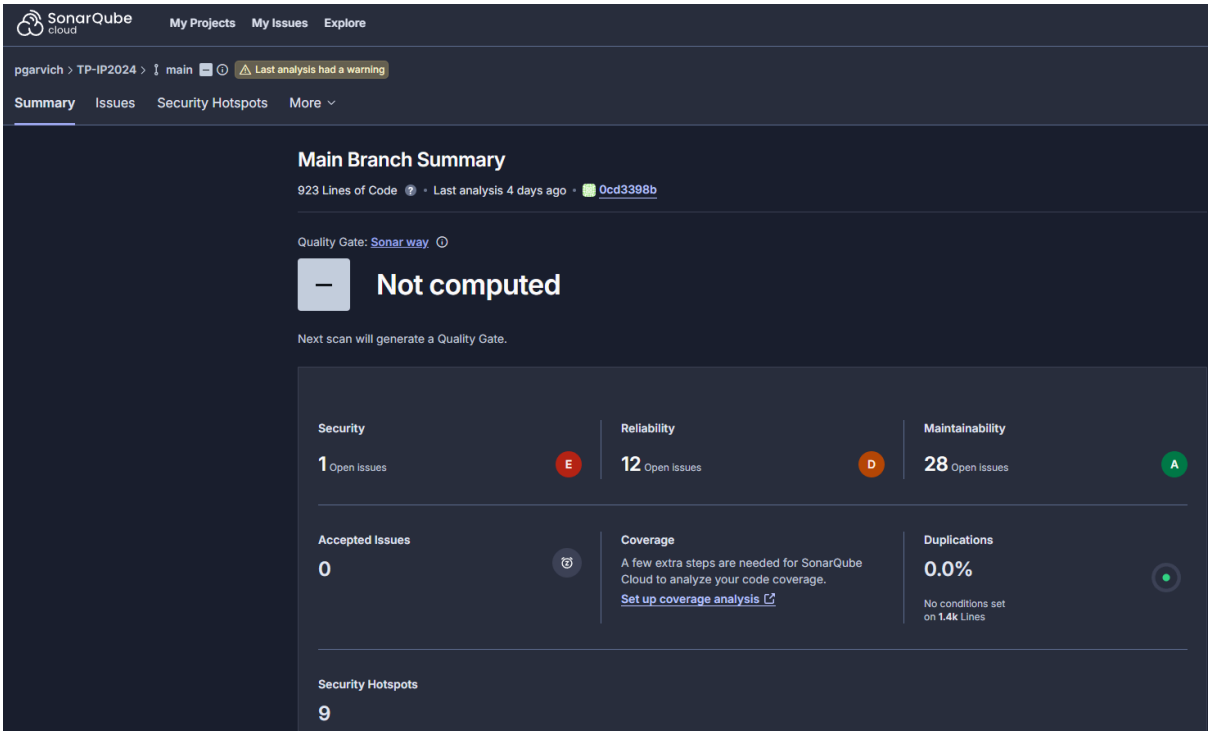
El ejercicio evidencia que **no todo hallazgo reportado por herramientas automáticas constituye una vulnerabilidad real**, y resalta la importancia de aplicar **criterio técnico, análisis contextual y priorización de riesgos** dentro de un proceso de evaluación de seguridad.

En conclusión, el estado general de la aplicación es coherente con su propósito educativo, aunque se identifican prácticas que deberían corregirse antes de un eventual despliegue productivo. El trabajo demuestra la capacidad de **identificar, evaluar, contextualizar y documentar hallazgos de seguridad**, alineándose con los principios de un **Secure Software Development Lifecycle (SSDLC)**.

# 2. Resumen de análisis estático

## SonarCloud

Como primer paso, se realizó un análisis estático del código fuente utilizando **SonarCloud**, con el objetivo de obtener una visión general del estado de seguridad, confiabilidad y mantenibilidad del proyecto.




El dashboard inicial arrojó los siguientes resultados:

- **Security Rating: E**
    - Motivo: existencia de **1 issue de alto impacto**, correspondiente al hallazgo **SEC-01**, el cual se analiza en detalle en la sección de vulnerabilidades de seguridad.
- **Reliability Rating: D**
    - Motivo: presencia de **1 issue de alto impacto** en la categoría Reliability.
    - Este issue corresponde al uso de un selector HTML inválido (`h10`) detectado en una hoja de estilos CSS.
    - Si bien el hallazgo es válido desde el punto de vista de calidad y estándares del código, **no representa un riesgo de seguridad**, ni es explotable.
    - Este caso ilustra que **no todos los issues reportados por herramientas automáticas implican vulnerabilidades de seguridad**, por lo que es necesario aplicar análisis contextual y criterio técnico.

- Adicionalmente, se identificaron **issues de severidad media y baja** relacionados con:
  - La regla **css:S4649 – Font declarations should contain at least one generic font family**, presente tanto en hojas de estilo como en estilos embebidos.
    Estos hallazgos afectan la **compatibilidad y robustez visual** en distintos navegadores, pero **no tienen impacto funcional ni de seguridad**.
  - Reglas de accesibilidad web, como la ausencia del atributo `alt` en etiquetas `<img>` o el uso redundante del término “image” en dicho atributo.

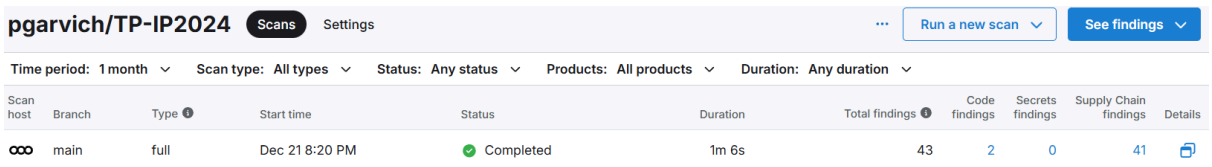
Estas observaciones son de **bajo impacto**, orientadas a mejorar la **accesibilidad, usabilidad y cumplimiento de buenas prácticas**, sin afectar la lógica ni la seguridad de la aplicación.

- **Maintainability Rating: A**
  - La relación de deuda técnica se mantiene por debajo del 5%.
  - No se detectaron duplicaciones de código.
  - Los issues abiertos corresponden a observaciones menores de mantenibilidad.

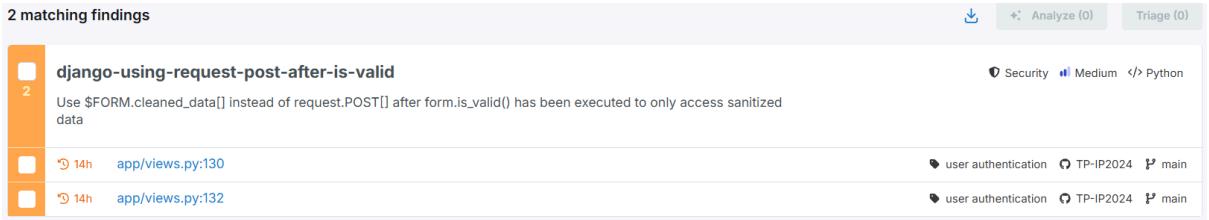
- **Security Hotspots: 9**
  - Todos los hotspots fueron analizados de forma individual.
  - Se clasificó su impacto según el contexto de uso de cada vista y endpoint.
  - Ninguno de los hotspots identificados representa una vulnerabilidad explotable en su estado actual.

### Análisis estático complementario – Semgrep

Como complemento al análisis realizado con SonarCloud, se ejecutó un análisis estático adicional utilizando **Semgrep**, incluyendo tanto reglas de análisis de código como detección de vulnerabilidades en la cadena de suministro (*Supply Chain Security*).

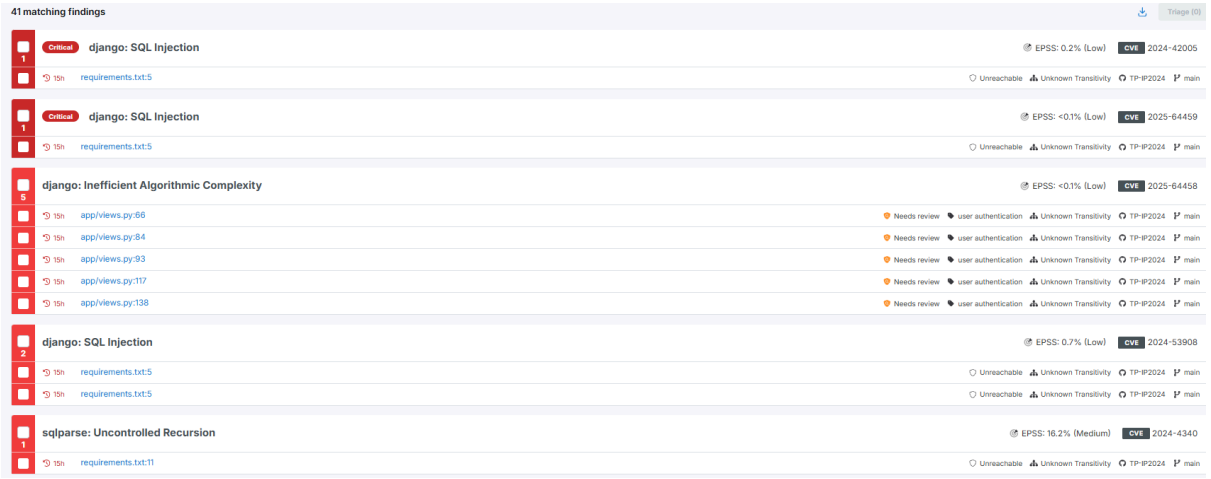



Resultados generales del dashboard:
Code findings: 2
Supply chain findings: 41




Se identificó un único patrón de code finding, relacionado con el uso directo de datos provenientes de request.POST luego de la validación de un formulario Django, utilizado en dos líneas consecutivas.

Este hallazgo fue analizado en detalle y documentado en la sección de Detalle de Hallazgos, confirmándose como un riesgo potencial asociado a validación incorrecta de entradas (CWE-20), con bajo impacto práctico en el contexto actual de la aplicación.




Los *supply chain findings* detectados corresponden mayoritariamente a vulnerabilidades conocidas (CVEs) asociadas a la versión de Django declarada en `requirements.txt`.

En la mayoría de los casos, Semgrep indica explícitamente que los hallazgos son **"Unreachable"**, es decir, que:

- No se identificó código en el proyecto que active las rutas vulnerables.
- Las condiciones necesarias para la explotación no se cumplen.
- El impacto práctico es nulo en el contexto de esta aplicación.

Algunos findings adicionales fueron clasificados como **"Needs review"**, asociados principalmente a vulnerabilidades de tipo *Denial of Service* bajo escenarios específicos (por ejemplo, ejecución en sistemas Windows y uso de redirects con entradas arbitrarias), los cuales no aplican al entorno de ejecución actual del proyecto.

Dado el carácter educativo de la aplicación, su ejecución local y la ausencia de exposición productiva, estos hallazgos se consideran **informativos** y no representan vulnerabilidades explotables en el escenario actual.

No obstante, se reconoce como buena práctica la **actualización periódica de dependencias**, especialmente ante un eventual despliegue en entornos productivos.

# 3. Detalle de hallazgos

## SEC-01 Exposición de clave secreta de Django en código fuente

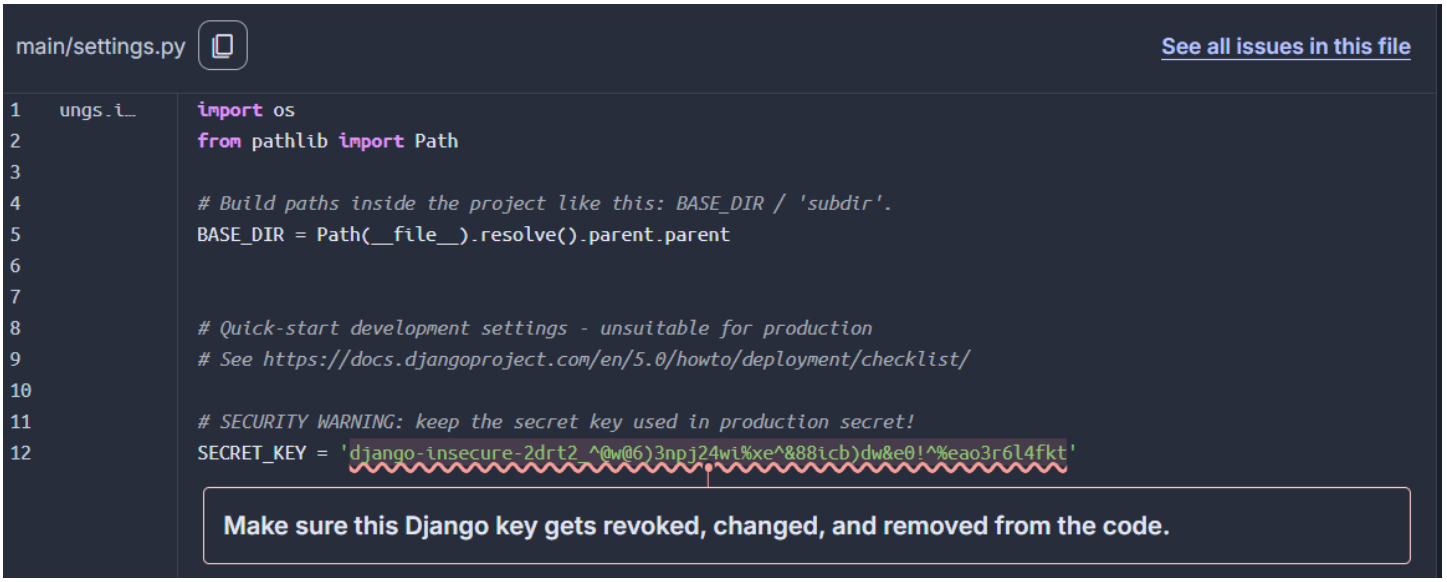
**Herramienta:** SonarCloud
**Categoría:** Security
**Tipo:** Gestión de secretos insegura
**Severidad:** Blocker
**Regla:** Django secret keys should not be disclosed - ID: secrets:S6687
**Estado:** Confirmado
**Ubicación:** main/setting.py - L:12
**Estado de remediación:** Propuesta documentada

### Descripción

Se detectó la presencia de una **clave secreta de Django** (SECRET_KEY) **hardcodeada** directamente en el archivo de configuración settings.py.

En Django, la SECRET_KEY es un componente crítico de seguridad utilizado para firmar **cookies de sesión, tokens CSRF, enlaces de recuperación de contraseña** y otros mecanismos criptográficos internos. La exposición de este valor en el código fuente, especialmente en repositorios públicos, compromete la integridad de los controles de autenticación y sesión de la aplicación.

### Evidencia




### Impacto potencial

La divulgación de la SECRET_KEY puede permitir a un atacante:

- Forjar cookies de sesión válidas.
- Bypassear protecciones CSRF.
- Generar tokens de recuperación de contraseña.
- Comprometer la autenticación y la gestión de sesiones.

Dado que esta clave es utilizada transversalmente por el framework, su exposición afecta a toda la superficie de seguridad de la aplicación, independientemente de que existan

o no vulnerabilidades adicionales en el código.

El impacto es crítico ya que compromete mecanismos centrales de seguridad del framework.

### Análisis contextual

Si bien la aplicación fue desarrollada con fines educativos y no se encuentra desplegada en un entorno productivo, el código fuente se encuentra versionado en un repositorio accesible. En este contexto, la presencia de secretos hardcodeados representa un riesgo real y una mala práctica de seguridad, alineada con escenarios comunes de exposición de credenciales en proyectos reales.

Este hallazgo no constituye un falso positivo, sino una vulnerabilidad legítima asociada a la gestión insegura de secretos.

### Remediación

Se recomienda:

1. Eliminar la SECRET_KEY del código fuente.
2. Externalizar la clave mediante variables de entorno o archivos de configuración no versionados. Ejemplo:
   ‘import os
   SECRET_KEY = os.environ.get("DJANGO_SECRET_KEY")’
3. Rotar la clave existente, invalidando cualquier sesión o token generado previamente.
4. Adoptar prácticas de gestión de secretos alineadas con el SSDLC.

### Referencias

OWASP - Top 10 2021 Category A7 - Identification and Authentication Failures
OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
CWE - CWE-798 - Use of Hard-coded Credentials
CWE - CWE-259 - Use of Hard-coded Password

## SH-01 Métodos HTTP seguros y no seguros en las vistas de Django

**Herramienta:** SonarCloud
**Categoría:** Security hotspot
**Tipo:** Cross-site request forgery (CSRF)
**Severidad:** Media
**Regla:** Allowing both safe and unsafe HTTP methods is security-sensitive - ID:python:S3752
**Estado:** Confirmado
**Ubicación:** app/views.py
Vistas involucradas:

- index_page
- members
- home
- search
- getAllFavouritesByUser
- saveFavourite

- deleteFavourite
- register

**Estado de remediación:** Propuesta documentada

### Descripción

Se identificaron múltiples *Security Hotspots* en el archivo `views.py` relacionados con la regla **python:S3752 – Allowing both safe and unsafe HTTP methods is security-sensitive**.

Las vistas afectadas no definen explícitamente los métodos HTTP permitidos, lo que provoca que acepten tanto métodos seguros (GET) como inseguros (POST, DELETE), independientemente de la operación que realizan.

Esta situación es especialmente relevante en vistas que **modifican el estado de la aplicación**, ya que puede debilitar las protecciones contra ataques como **Cross-Site Request Forgery (CSRF)** y permitir usos indebidos de los endpoints.

### Evidencia

```
@login_required
def getAllFavouritesByUser(request):

    favourite_list = services.getAllFavourites(request)# llama al servicio para obtener favoritos


    return render(request, 'favourites.html', { 'favourite_list': favourite_list })


@login_required
def saveFavourite(request):

    services.saveFavourite(request) #llama al servicio para guardar el favorito

    return redirect(home)



@login_required
def deleteFavourite(request):

    services.deleteFavourite(request) #llama al servicio para eliminar el favorito

    return redirect(getAllFavouritesByUser)
```

```
def index_page(request):

    return render(request, 'index.html')

def members(request):

    return render(request, 'members.html')

# esta función obtiene 2 listados que corresponden a las imágenes de la API y los favoritos del usuario, y los
# si el opcional de favoritos no está desarrollado, devuelve un listado vacío.
def home(request):

    contador=1

    images = services.getAllImages()

    images = Paginator(images,20)

    favourite_list = services.getAllFavourites(request)

    return render(request, 'home.html', {'images': images.page(contador), 'favourite_list': favourite_list})
```

```python
def register(request):

    form = UserCreationForm(request.POST)

    if request.method == 'POST':

        if form.is_valid():

            form.save()

            user_name=request.POST['username']

            user_pwd=request.POST['password1']

            user=authenticate(request,username=user_name,password=user_pwd)

            login(request,user)

            return redirect('home')

        else:

            messages.error(request,"No se pudo crear usuario, intentelo de nuevo")

            return render(request, 'registration/register.html', {'form': form})

    return render(request, 'registration/register.html', {'form': form})
```

### Impacto potencial

**Vistas con modificación de estado (saveFavourite, deleteFavourite):**

Permiten operaciones de escritura y eliminación sin restricción explícita del método HTTP. En un escenario productivo, esto podría facilitar el abuso de endpoints y debilitar controles como CSRF si se incorporan nuevas funcionalidades o configuraciones inseguras. Impacto potencial: medio.

Vista con uso de POST sin persistencia (search):

Procesa datos enviados por POST únicamente para filtrar resultados en memoria, sin almacenamiento ni efectos persistentes. Impacto potencial: bajo a medio.

Vistas de solo lectura (index_page, members, home, getAllFavouritesByUser):

No realizan cambios sobre el estado de la aplicación y se limitan a renderizar información. Impacto potencial: bajo (informativo).

### Análisis contextual

La regla python:S3752 alerta sobre vistas que no restringen explícitamente los métodos HTTP permitidos, lo que podría habilitar el uso de métodos seguros y no seguros sobre una misma operación.

En el contexto de esta aplicación Django:

- Varias vistas solo renderizan templates o muestran información, sin modificar el estado del sistema.
- Django aplica protección CSRF por defecto en formularios que utilizan métodos inseguros (POST).
- No se trata de una API REST expuesta, sino de una aplicación web monolítica de uso educativo.

Tras el análisis:

- No existe riesgo real en las vistas de solo lectura.
- El riesgo es relevante únicamente en las vistas que realizan operaciones de escritura o eliminación (saveFavourite, deleteFavourite).
- En estos casos se recomienda restringir explícitamente el método HTTP.
- El resto de los hotspots se consideran revisados y aceptados, al no representar un impacto de seguridad práctico.

Este hallazgo no constituye una vulnerabilidad explotable per sé, sino un conjunto de configuraciones que requieren diferenciación según el tipo de vista. El riesgo se considera mitigable mediante buenas prácticas de definición de métodos HTTP.

**Remediación**

Definir explícitamente los métodos HTTP permitidos en cada vista, de acuerdo con su propósito, utilizando decoradores provistos por Django:

- Para vistas de solo lectura:

```
from django.views.decorators.http import require_GET
```

- Para vistas que modifican estado:

```
from django.views.decorators.http import require_POST
```

Ejemplo de corrección:

```
from django.views.decorators.http import require_POST

@require_POST
@login_required
def saveFavourite(request):
    services.saveFavourite(request)
    return redirect(home)
```

**Referencias**

OWASP Top 10: A1 - Broken Access Control , A4 - Insecure Design
CWE-352 - Cross-Site Request Forgery (CSRF)
Django - Allowed HTTP Methods

## SH-02 Configuración DEBUG habilitada en Django

**Herramienta:** SonarCloud
**Categoría:** Security hotspot
**Tipo:** Configuración insegura
**Severidad:** Baja

**Regla:** Delivering code in production with debug features activated is security-sensitive ID-python:S4507

**Estado:** Confirmado

**Ubicación:** main/settings.py

**Estado de la remediación:** Hallazgo documentado como **mejora de configuración recomendada**. Riesgo aceptado en el contexto actual del proyecto.

### Descripción

Se detectó que la aplicación tiene habilitado el modo de depuración (`DEBUG = True`) en el archivo de configuración `settings.py`.

El modo DEBUG proporciona información detallada sobre errores y excepciones de la aplicación, lo cual resulta útil durante el desarrollo, pero puede exponer información sensible si se utiliza en un entorno productivo.

### Evidencia


### Impacto potencial

En un entorno de producción, mantener el modo DEBUG activo puede:

- Exponer trazas de error con información interna de la aplicación.
- Revelar detalles sobre la estructura del sistema, rutas internas y configuraciones.
- Facilitar tareas de reconocimiento a un potencial atacante.

### Análisis contextual

En el contexto de este proyecto, el riesgo es **bajo**, ya que se trata de una aplicación desarrollada con fines académicos y de aprendizaje, ejecutada en un entorno local y sin despliegue productivo ni exposición pública.

La configuración observada es coherente con un entorno de desarrollo y no representa una vulnerabilidad explotable en el escenario actual.

### Recomendación

Desactivar el modo DEBUG antes de desplegar la aplicación en un entorno productivo, estableciendo `DEBUG = False` y gestionando esta configuración mediante variables de entorno, de acuerdo con las buenas prácticas recomendadas por Django.

### Referencias

OWASP - Top 10 2021 Category A5 - Security Misconfiguration

OWASP - Top 10 2017 Category A3 - Sensitive Data Exposure
CWE - CWE-489 - Active Debug Code
CWE - CWE-215 - Information Exposure Through Debug Information

## SG-01 Uso de datos sin sanitizar desde request POST tras validación de formulario

**Herramienta:** SemGrep
**Categoría:** Security
**Tipo:** Validación inadecuada de entradas
**Severidad:** Media
**Regla:** django-using-request-post-after-is-valid
**Estado:** Confirmado
**Ubicación:** app/views.py L:130 y 132
**Estado de la remediación:** Propuesta de remediación documentada

### Descripción

Se detectó el uso directo de datos provenientes de request.POST (username y password1) luego de ejecutar form.is_valid(), en lugar de utilizar los valores validados y normalizados disponibles en form.cleaned_data.

Aunque la validación del formulario se ejecuta correctamente, el acceso posterior a request.POST implica el uso de datos crudos del usuario, lo que rompe el modelo de validación esperado del framework Django.

### Evidencia

app/views.py:130 user authentication

```
120 def register(request):
121
122     form = UserCreationForm(request.POST)
123
124     if request.method == 'POST':
125
126         if form.is_valid():
127
128             form.save()
129
130             user_name=request.POST['username']
131
132             user_pwd=request.POST['password1']
133
134             user=authenticate(request,username=user_name,password=user_pwd)
135
136             login(request,user)
137
138             return redirect('home')
```

### Impacto potencial

En el estado actual de la aplicación, el impacto práctico es bajo, ya que:

- `UserCreationForm` aplica validaciones estrictas.
- Django ORM protege contra inyecciones SQL.

- El motor de templates escapa contenido HTML por defecto.

No obstante, este patrón puede introducir riesgos de inyección (XSS, SQLi u otros) si los datos se reutilizan en otros contextos, se relajan validaciones futuras o se incorporan nuevos flujos que asuman datos sanitizados.

### Análisis contextual

La aplicación fue desarrollada con fines educativos y no se encuentra expuesta en producción. Durante pruebas manuales, no se observó comportamiento inseguro ante entradas maliciosas, debido a las protecciones inherentes del framework Django.

Sin embargo, el hallazgo representa una **mala práctica de seguridad** y un **riesgo potencial latente**, correctamente identificado por la herramienta de análisis estático.

## Remediación

Utilizar exclusivamente los datos validados del formulario:

```
user_name = form.cleaned_data['username']
user_pwd = form.cleaned_data['password1']
```

Esto asegura que solo se procesen valores previamente validados y sanitizados por Django, alineándose con las buenas prácticas del framework.

### Referencias

CWE - CWE-20 - Improper Input Validation

# 3. Resumen de remediaciones recomendadas

### 🔴 Alta prioridad (Seguridad crítica)

- **Eliminar secretos hardcodeados del código fuente**
  - Remover la `SECRET_KEY` del archivo `settings.py`.
  - Externalizar la clave mediante variables de entorno.
  - Rotar la clave existente para invalidar sesiones y tokens previos.
  - Evitar versionar archivos que contengan secretos.

### 🟠 Prioridad media (Endurecimiento de seguridad)

- **Restringir explícitamente los métodos HTTP permitidos**
  - Utilizar decoradores como `@require_GET` y `@require_POST` según el propósito de cada vista.
  - Aplicar especialmente en vistas que modifican el estado de la aplicación (`saveFavourite`, `deleteFavourite`).
  - Reducir la superficie de ataque y evitar usos indebidos de endpoints.

- **Utilizar datos validados del formulario**
  - Reemplazar accesos directos a `request.POST` por `form.cleaned_data`.
  - Mantener coherencia con el modelo de validación del framework Django.
  - Prevenir riesgos futuros ante cambios en validaciones o reutilización de datos.

### 🟡 Baja prioridad (Configuración y buenas prácticas)

- **Desactivar DEBUG en entornos productivos**
  - Establecer `DEBUG = False` antes de cualquier despliegue.
  - Gestionar esta configuración mediante variables de entorno.
  - Evitar la exposición de información interna de la aplicación.

- **Actualizar dependencias ante despliegues productivos**
  - Revisar periódicamente versiones del framework y librerías.
  - Priorizar actualizaciones cuando existan CVEs explotables o escenarios alcanzables.
  - Validar impacto real antes de aplicar parches en proyectos educativos.

---

### 🔵 Mejora continua (Calidad y mantenibilidad)

- Corregir observaciones de accesibilidad (atributo `alt` en imágenes).
- Ajustar reglas CSS relacionadas con compatibilidad visual.