

# Desarrollo Seguro de Aplicaciones -

## Práctica 3

### Autores - Grupo 'Error 404'

- Juan Cruz Cassera Botta, 17072/7
- Brian Llamocca, 02037/9
- Franco Camacho, 16194/1

### Notas

- **Fecha de entrega: 29/05/2025 a las 17:59.**
- Términos relacionados:
  - OWASP top 10 (2021): A03 (**inyección**).
  - Inyección SQL (**SQLI**).
  - Ejecución remota de comandos (**RCE**).
  - Server side template injection (**SSTI**).
  - Cross Site Scripting (**XSS**).

### Guía para hacer la práctica

- Continuaremos utilizando el repositorio grupal de código en GitHub y **Docker con docker-compose**.
- Ejecute ***git pull origin main*** en la carpeta del repositorio para bajar los últimos cambios.
- Ejecute los ejercicios accediendo a la carpeta **practica3** y corriendo el comando ***./run.sh***
- Para acceder vía web a cada reto deberá ingresar al puerto que corresponda, por ejemplo, **para el reto python\_sqli**, **http://localhost:13001**

# Ejercicio 1 - Explotación y fix

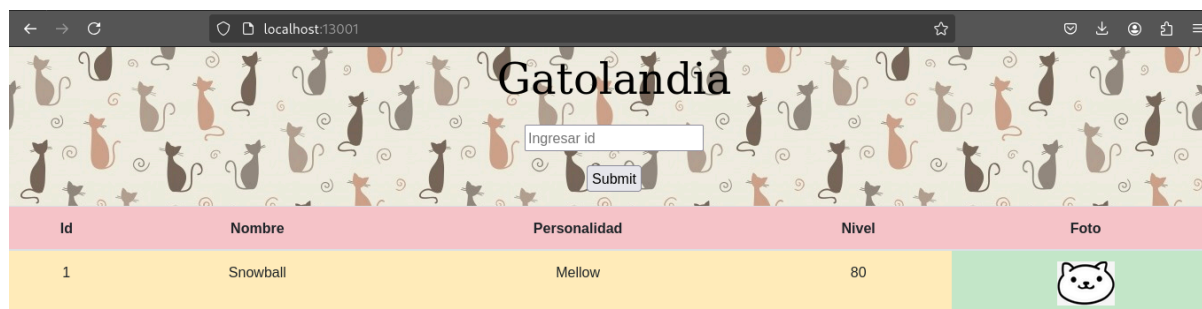
Para cada ejercicio explote la vulnerabilidad y genere un parche con el código que arregle la vulnerabilidad editando el código fuente de cada ejercicio en la carpeta **www**. Realizar un push en el branch **"fix-practica3"**.

**Nota: Evitar soluciones basadas en la confianza ciega de los datos proporcionados por el usuario sin una validación adecuada. Su solución no debe bloquear o restringir ciertos inputs de usuario. No depender únicamente de medidas de seguridad en el frontend, sino también implementar controles de seguridad en el backend.**

## Reto python\_sqli (<http://localhost:13001>)

### Cómo se explotó la vulnerabilidad

Para la siguiente vulnerabilidad se accedió a la página web dada por la cátedra, en la cual se observa que se tiene un input el cual nos invita a ingresar a un id, y una tabla que muestra los datos del gato con ID igual a 1. Por lo tanto, al ingresar un id, se mostrarán los datos de un gato particular por su ID. Esto se logra realizando una query SQL a la base de datos de la aplicación.



Dicho esto, es lógico pensar que la flag la posee alguno de todos los gatos. Pero como no sabemos cuál de todos, no es nada práctico ir probando IDs uno por uno, sobre todo porque podrían haber miles de gatos en la DB.

Es por esto que se consideró un **ataque de inyección SQL que devuelva los datos de todos los gatos de la DB a la vez, para rápidamente scrollear en la tabla y encontrar la flag.**

Para este ataque, el objetivo es ingresar en el input un código SQL que se concatene con el código SQL de la consulta existente a la DB de la aplicación. Dicho código debería manipular a la consulta de manera tal que devuelva todas

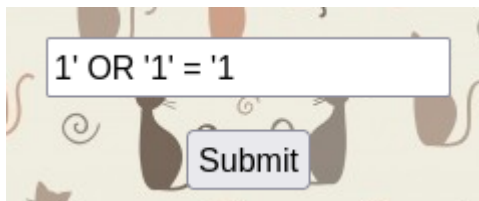
las filas de la tabla. Como tenemos una consulta que busca gatos por ID, esto en SQL se resuelve trivialmente: **SELECT \* FROM gatos WHERE id = 'id\_enviado';**. Por lo tanto, tenemos que inyectar algo luego del signo igual (=) de forma tal que el WHERE evalúe siempre a TRUE. La forma más fácil de que una condición sea siempre verdadera es usando un OR de forma tal que la segunda mitad de la condición doble sea siempre verdadero (por ejemplo  $1 = 1$ ) lo cual hará a toda la condición verdadera.










Sabiendo esto, primero pensamos inyectar: **1 OR 1 = 1**. Sin embargo, esto no funciona porque las comillas simples lo rompen: **SELECT \* FROM gatos WHERE id = '1 OR 1 = 1';** no es válido porque toda la condición se interpreta como un string.

Esto lo solucionamos haciendo que la comparación se haga con strings del número 1: **1' OR '1' = '1'** lo cual se convierte en (al inyectarse):

**SELECT \* FROM gatos WHERE id = '1' OR '1' = '1';**

Como se explicó, la condición siempre es verdadera lo cual provoca que se traigan todas las filas de la tabla.



localhost:13001/?numero=1'+OR+'1'+%3D+'1					
4	Tabitha	Leisurely	40		
5	Bandit	Wild at Heart	180		
6	Pepper	Shy	165		
7	Patches	Jealous	80		
8	Princess	Ditzy	125		
9	Ginger	Bashful	91		
10	Peaches	Capricious	45		
16161	Wally	flag{Bieeen_esta_eslaflag}	250		

Al poder ver todos los gatos de la DB gracias a la inyección, se encuentra la flag:

***flag{Bieeen\_esta\_eslaflag}***

## Cómo se generó un parche para solucionar la vulnerabilidad

Para solucionar esta vulnerabilidad de inyección SQL se hicieron varios cambios:

En el frontend, se le agregó al tag input en la línea 14:

**input name="numero" placeholder="Ingresar id" id="numero" required>**

el atributo **type = "number"** para que no se pueda ingresar texto en el input. Esto no nos parece una restricción ya que un ID de un gato no debería ser nunca texto, si no que siempre es un número entero.

Desde el backend, se usó la técnica de **consultas parametrizadas** para evitar la inyección directa del input dentro de la query SQL. Esta técnica se hace de distintas formas según la librería de base de datos que se esté usando. En el caso de esta aplicación, como usa sqlalchemy, se logra con tres pasos:

- 1) Importar "text" de sqlalchemy.
- 2) Encuadrar la consulta SQL en la función text: sql = text("SELECT \* ...")
- 3) Al hacer connection.execute(), se le manda además de la variable sql anterior, un diccionario con el nombre de la columna como **clave**, y el

parámetro que ingresó el usuario como **valor**. Así, si el usuario intenta inyectar código SQL, no tendrá éxito.

## Reto python\_rce (<http://localhost:13002>)

### Cómo se explotó la vulnerabilidad

La aplicación realiza un ping a una IP que ingresa el usuario, usando el comando ping. Si el usuario ingresa una IP no válida, la aplicación no realiza el ping e informa el error.

🛡️ 📄 localhost:13002

# Chequeador de Ping

Ingresá una IP para que nosotros le hagamos ping :)

Resultado:

PING 8.8.8.8 (8.8.8.8): 56 data bytes  
64 bytes from 8.8.8.8: seq=0 ttl=118 time=39.587 ms

--- 8.8.8.8 ping statistics ---  
1 packets transmitted, 1 packets received, 0% packet loss  
round-trip min/avg/max = 39.587/39.587/39.587 ms

Sin embargo, como el input del usuario pasará a formar parte de la escritura de un comando Linux, se puede explotar esto para hacer que la aplicación ejecute cualquier otro comando, haciendo uso del punto y coma: si el usuario ingresa por ejemplo 8.8.8.8; ls, el comando final será:

**ping 8.8.8.8; ls**

Lo cual provocará que se ejecuten dos comandos (primero el ping y luego el ls), en vez de uno.

# Chequeador de Ping

Ingresá una IP para que nosotros le hagamos ping :)



Resultado:

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=118 time=7.946 ms
```

```
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 7.946/7.946/7.946 ms
    bin
    dev
entrypoint.sh
    etc
    home
    lib
    media
    mnt
    proc
    root
```

Dicho de otra manera, se puede explotar esta vulnerabilidad para ejecutar remotamente cualquier comando que se quiera en el sistema operativo del servidor.

Para encontrar la flag, se procede a acceder al directorio `www`, donde se puede encontrar el archivo **secrets.txt**. Para acceder al mismo, se utilizó la siguiente serie de comandos, en el input: `8.8.8.8; cd www; cat secrets.txt`

# Chequeador de Ping

Ingresá una IP para que nosotros le hagamos ping :)



Resultado:

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=118 time=7.021 ms
```

```
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 7.021/7.021/7.021 ms
    flag{Tremend0polis!}
```

Así, se encuentra que la flag es:

*flag{Tremend0polis!}*

## Cómo se generó un parche para solucionar la vulnerabilidad

Para poder solucionar el problema mencionado, se modificó el método **subprocess.check\_output()**, el cual ejecuta el comando ingresado por argumento, espera a que se ejecute, devuelve el resultado del mismo y en caso de error, devuelve una excepción.

Teniendo en cuenta el funcionamiento de ese método, los cambios a realizar fueron los siguientes:

- 1) Se le pasa por parámetro una lista en la cual sus elementos son:
  - a) El nombre del comando.
  - b) Su primer argumento.
  - c) El valor para ese argumento.
- 2) Ya no se le pasa por parámetro shell=True (por defecto es False) para que nada de lo que ingresa el usuario sea evaluado a comandos de shell, si no solo argumentos.
- 3) Se agrega el parámetro stderr=subprocess.STDOUT, el cual redirige el error estándar al stdout, para capturarlo todo junto.

De esta manera, cada vez que se intente inyectar un comando con **dirección IP; otro comando**, mostrará el mensaje de error dado por la excepción.

# Chequeador de Ping

Ingresa una IP para que nosotros le hagamos ping :)

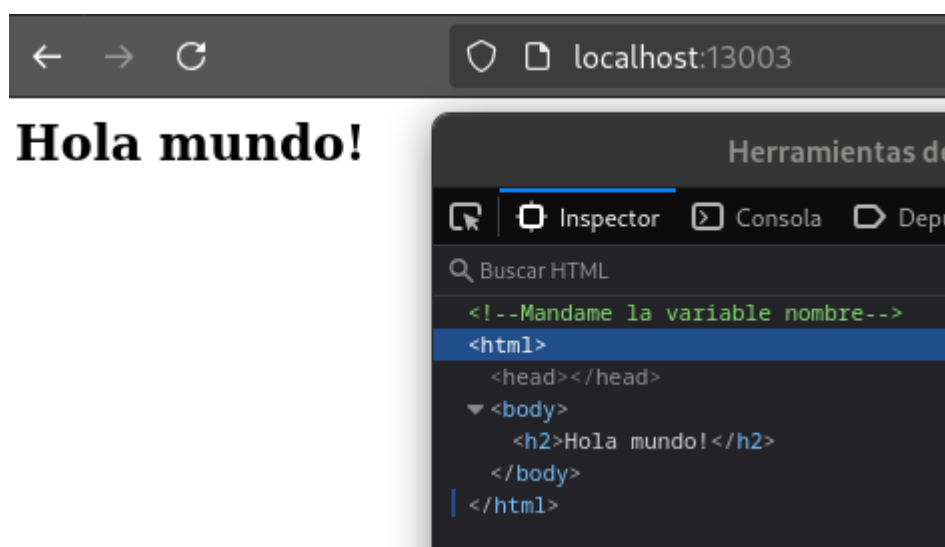
Resultado:

El host con la IP ingresada no responde

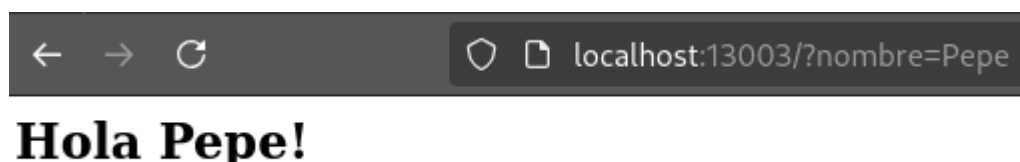
## Reto python\_ssti (<http://localhost:13003>)

### Cómo se explotó la vulnerabilidad

La página de este reto muestra un “Hola mundo!”. Al inspeccionarla, vemos que en el código HTML hay un comentario que indica que hay que mandar la variable “nombre”.



Al enviar por ejemplo “Pepe”, el saludo cambia dinámicamente a “Hola Pepe!”



Esto nos indica que el servidor de la página está haciendo algo con el valor de la variable nombre que le enviamos. Como el nombre del ejercicio está relacionado a Template Injection y sabemos que estos ejercicios están usando Python + Flask, el lenguaje de plantillas típico de este framework es Jinja. Lo que hace Jinja es embebir código Python (asignaciones, llamadas a funciones, modificar variables, etc) dentro del código HTML, para generar código HTML dinámico.

La sintaxis que usa Jinja son las doble llaves “{{ }}”. Es decir, Flask llama a una plantilla que posee código HTML y le pasa datos (variables o funciones principalmente). En este caso, se puede asumir que el nombre que enviamos por parámetro se procesa por Python y luego se lo envía a la plantilla para que renderice el HTML con ese nombre.





**Hola 2\*2!**

En este caso, el argumento se interpreta como texto plano, Python lo entiende como un string y nada más.



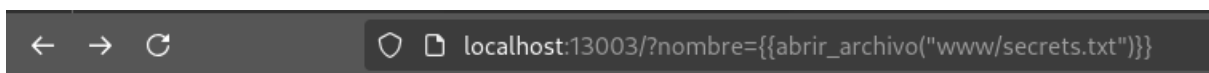
**Hola 4!**

En cambio en este caso, como le agregamos las llaves, Python interpreta ese código, y realiza la multiplicación.

Inspeccionando el código interno (app.py) de la aplicación, vemos que posee una función **abrir\_archivo(f\_name)** que abre el archivo pasado como parámetro y devuelve su contenido como texto.

Entonces lo único que tenemos que hacer es enviar un argumento que, dentro de las llaves, llame a esa función enviándole como parámetro la ruta al archivo secrets.txt que posee la flag.

[http://localhost:13003/?nombre={{abrir\\_archivo\('www/secrets.txt'\)}}](http://localhost:13003/?nombre={{abrir_archivo('www/secrets.txt')}})



**Hola ['flag{alto\_hacker\_sos!\_firmame\_un\_autografo}']!**

Entonces la flag es: ***flag{alto\_hacker\_sos!\_firmame\_un\_autografo}***

## Cómo se generó un parche para solucionar la vulnerabilidad

Para evitar SSTI en este caso, se colocaron las llaves dentro del código HTML mismo y se pasa como argumento el valor que se envió como parámetro, el cual no será tratado como código si no como texto plano, debido a que ya no se

inserta directamente en el template string si no que se lo envía a Jinja al llamar a `render_template()`.

De esta forma, como se puede ver debajo, ya no se puede inyectar código Python a las templates porque las llaves no hacen nada, se tratan como texto.



**Hola {{2\*2}}!**



**Hola {{abrir\_archivo("www/secrets.txt')}}!**

## Reto python\_xss (<http://localhost:13004>)

### Cómo se explotó la vulnerabilidad

En este caso, se tiene una aplicación que muestra información de películas, con un buscador de películas por nombre.

Si buscamos una película que existe en la “base de datos” de la aplicación, se muestra la info de la misma sin problema:



## Resultados de búsqueda

Resultados para 'ESe oscuro':

Imagen de la película

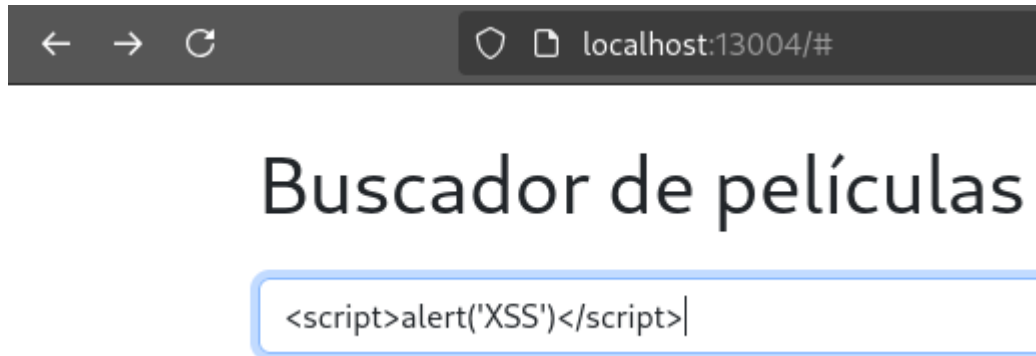
**Ese oscuro objeto del deseo**

Descripción de la película

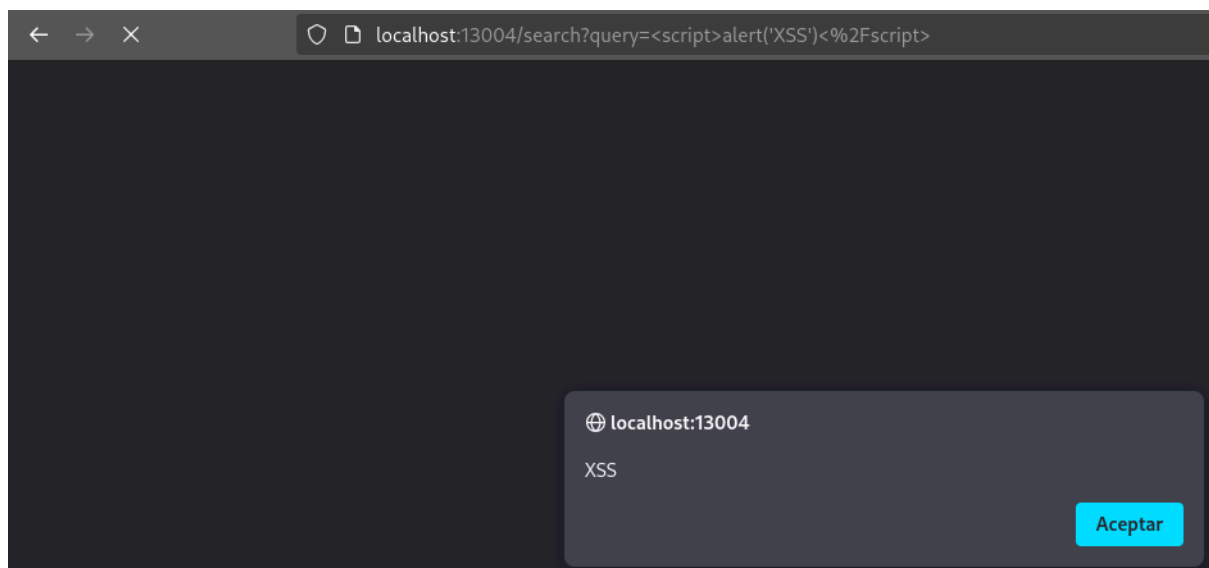
[Ver detalles](#)

En la imagen anterior, sin embargo, podemos notar que lo que nosotros escribimos en el buscador se imprime en el HTML: ingresamos “ESe oscuro” y dentro del HTML resultado se puede ver “Resultados para ‘ESe oscuro’”. Esto nos indica que el input del usuario quizá puede ser explotado haciendo que, al ingresar como “nombre de película” un código HTML, éste se inyecte dentro del código HTML de la página misma y modifique su comportamiento.

Por ejemplo, si ingresamos `<script>alert('XSS')</script>`



Se puede ver que efectivamente, ese input modifica el HTML de la página haciendo que se ejecute el código HTML/JavaScript que queramos.



Otro ejemplo:



## Buscador de películas



### Resultados de búsqueda

Resultados para '

**XSS**

':

### Cómo se generó un parche para solucionar la vulnerabilidad

Inspeccionando el código de la aplicación, el problema principal que tiene la misma y que causa que esté vulnerable a XSS está en la línea 10 del archivo **search.html**. La directiva “ | safe” en Jinja indica que la variable {{ variable }} que se le pasa al template es segura, y no es necesario escaparla. Pero esto está mal en este caso, ya que no se requiere del uso de caracteres especiales, además de que es peligroso confiar ciegamente en el input del usuario.

Entonces, para solucionar el problema, borramos el “ | safe”, lo que causa que ahora, si se pasa código HTML/JavaScript como input, será tratado como texto plano ya que Jinja escapará de forma automática los caracteres peligrosos (principalmente los “<” “>” de los tags):



# Buscador de películas

<h1>XSS</h1>

Como se puede ver debajo, el código HTML ya no se inyecta en el HTML de la página.

## Resultados de búsqueda

Resultados para '<h1>XSS</h1>':