

## INYECCION SQL



En el ámbito de la ciberseguridad, cuando trabajamos como Blue Team y Red Team adoptamos dos roles complementarios que nos permiten comprender un sistema desde perspectivas opuestas. Como Blue Team, nos centramos en proteger, reforzar y asegurar la aplicación, aplicando buenas prácticas y anticipándonos a posibles ataques. Como Red Team, en cambio, asumimos la mentalidad del atacante: buscamos fallos, analizamos puntos débiles y tratamos de explotar vulnerabilidades para demostrar qué tan lejos podría llegar un adversario real. Esta dualidad nos ofrece una visión completa del ciclo de seguridad.

Cuando estudiamos la inyección SQL, entendemos que se trata de una vulnerabilidad que aparece cuando una aplicación mezcla instrucciones SQL con datos proporcionados por el usuario. Si esa mezcla no se controla adecuadamente, un atacante puede introducir fragmentos de código SQL dentro de los campos de entrada, alterando la consulta original y obteniendo acceso a información o funciones que no deberían estar disponibles. Es una de las técnicas más conocidas y peligrosas en aplicaciones que interactúan con bases de datos.

La forma de evitar esta vulnerabilidad consiste en separar estrictamente el código SQL de los datos que introduce el usuario. Cuando utilizamos consultas parametrizadas, el motor de la base de datos recibe primero la estructura de la consulta y después los valores, tratándolos siempre como texto sin capacidad de modificar la lógica. De esta manera, incluso si alguien intenta introducir un payload malicioso, la base de datos lo interpreta como un simple dato y no como una instrucción ejecutable. Esta separación es la base de una defensa sólida frente a la inyección SQL.

## FASE 1: DESARROLLO Y PREPARACIÓN DEL PROTOTIPO (BLUE TEAM)

**OBJETIVO:** Diseñar y construir dos versiones de una aplicación (vulnerable y segura) en Python + SQLite, con datos realistas y documentación clara.

### HERRAMIENTAS CLAVE:

- **Visual Studio Code** → desarrollo del código
- **Python 3** → lenguaje principal
- **SQLite** → base de datos local
- **Faker** → generación de datos realistas
- **Markdown** → documentación en README.md
- **Linux Desktop (VBox)** → entorno de desarrollo controlado
- **OWASP** → guía de buenas prácticas y referencias de seguridad

### ENTREGABLES EN LA ESTRUCTURA DEL PROYECTO:

```
Linux Desktop (Documentos)
    — README.md
    — app_vulnerable.py
    — app_segura.py
    — database.sql
    — generar_datos.py
    — requirements.txt
```

### BLUE TEAM

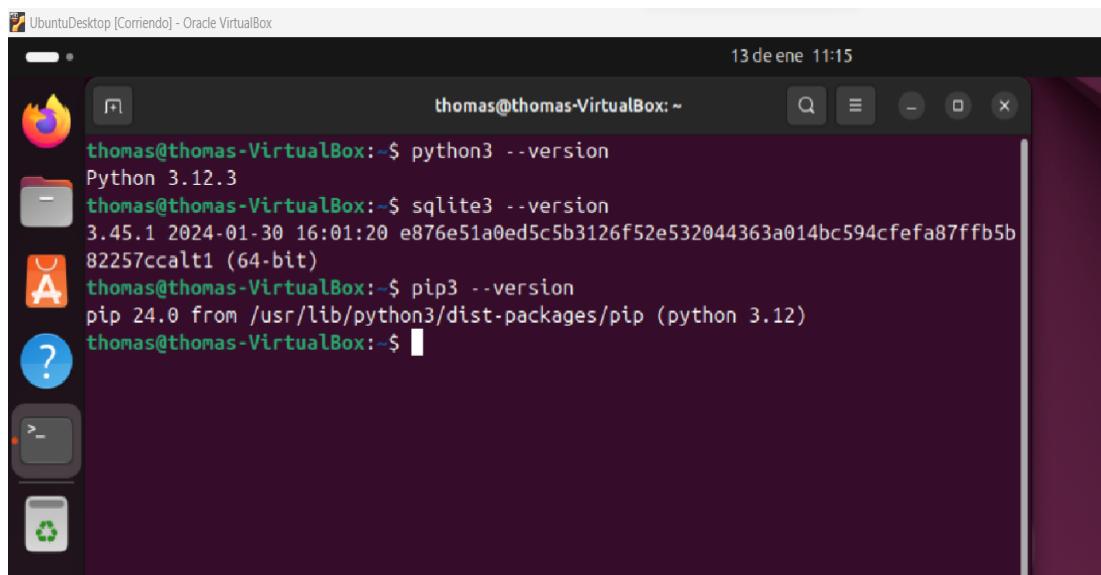
En este proyecto, cuando actuamos como Blue Team nos encargamos de diseñar y construir la aplicación desde cero, definiendo la base de datos, implementando las funciones principales y preparando tanto la versión vulnerable como la versión segura. Aplicamos buenas prácticas de desarrollo, generamos datos realistas y dejamos el sistema listo para ser evaluado y atacado por otros equipos, asegurándonos de que todo esté correctamente documentado y estructurado.

Después, cuando recibimos el informe del Red Team, analizamos cada vulnerabilidad explotada, identificamos su origen y reforzamos el código aplicando defensas como consultas parametrizadas, validación de entradas, límites de longitud, sanitización y manejo seguro de errores. Validamos que la versión segura resista todos los ataques, realizamos pruebas cruzadas en entornos controlados y documentamos cada mejora aplicada. Nuestro objetivo final es transformar una aplicación débil en un sistema robusto y demostrar una comprensión completa del ciclo de seguridad.

```
# Instala el gestor de BD SQLite3 y el gestor de paquetes pip para Python 3
sudo apt install sqlite3 python3-pip -y
```

- sqlite3: instala SQLite3, un sistema de base de datos ligero basado en archivos.
- python3-pip: instala pip, el gestor de paquetes para Python 3.

```
python3 --version
sqlite3 --version
pip3 --version
```



The screenshot shows a terminal window titled "UbuntuDesktop [Corriendo] - Oracle VirtualBox". The window title bar includes the date and time: "13 de ene 11:15". The terminal prompt is "thomas@thomas-VirtualBox:~\$". The user runs three commands:  
1. "python3 --version" which outputs "Python 3.12.3"  
2. "sqlite3 --version" which outputs "3.45.1 2024-01-30 16:01:20 e876e51a0ed5c5b3126f52e532044363a014bc594cfefa87ffb5b82257ccalt1 (64-bit)"  
3. "pip3 --version" which outputs "pip 24.0 from /usr/lib/python3/dist-packages/pip (python 3.12)"

```
# Crea el directorio InyeccionSQL dentro de Documentos
mkdir -p ~/Documentos/InyeccionSQL
```

```
# Crea el punto de montaje InyeccionSQL en /mnt con privilegios de administrador
sudo mkdir -p /mnt/InyeccionSQL
```

```
# Monta la carpeta compartida en /mnt/InyeccionSQL
sudo mount -t vboxsf InyeccionSQL /mnt/InyeccionSQL
```

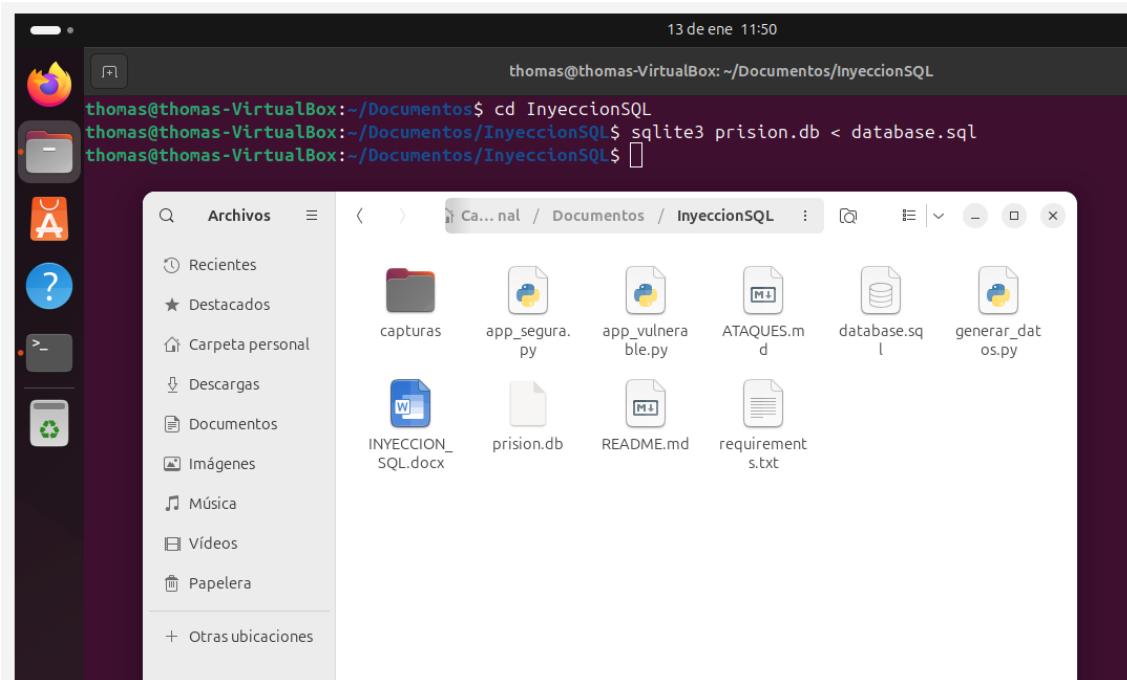
- mount: comando que permite montar un sistema de archivos en el sistema.
- -t vboxsf: especifica el tipo de sistema de archivos VirtualBox Shared Folder (carpeta compartida de VirtualBox).

```
# Copia todo el contenido de la carpeta montada al directorio del usuario
cp -r /mnt/InyeccionSQL/* ~/Documentos/InyeccionSQL/
```

```
# Accede al directorio InyeccionSQL dentro de Documentos
cd ~/Documentos/InyeccionSQL
```

```
# Ejecuta el script database.sql para crear y cargar la base de datos prision.db
sqlite3 prision.db < database.sql
```

- sqlite3: línea de comandos para trabajar con bases de datos SQLite.
- prision.db: nombre del archivo de BD SQLite que se va a crear o usar.
  - Si no existe, SQLite lo crea automáticamente.
  - Si existe, se reutiliza.
- <: operador de redirección de entrada estándar.
- database.sql: archivo con sentencias SQL (CREATE, INSERT, etc.).



```
# Actualiza la lista de paquetes disponibles del sistema
sudo apt update
```

```
# Instala el módulo venv para crear entornos virtuales en Python 3
sudo apt install python3-venv -y
```

- python3-venv: instala el módulo venv, que permite crear entornos virtuales de Python 3 para aislar dependencias de proyectos.

El objetivo es aislar las dependencias de cada proyecto. Esto es importante porque distintos proyectos pueden necesitar versiones diferentes de las mismas librerías, y sin entornos virtuales se generaría conflictos al instalar paquetes de forma global. Al usar venv, cada proyecto tiene su propio espacio de librerías y configuración, lo que mejora la seguridad, estabilidad y orden del sistema, evita romper otros proyectos y facilita la reproducibilidad del entorno en prácticas, laboratorios o despliegues.

```
# Crea un entorno virtual de Python llamado "venv"  
python3 -m venv venv
```

```
# Activa el entorno virtual para aislar las dependencias del proyecto  
source venv/bin/activate
```

- `source`: ejecuta un script en la sesión actual de la terminal (no abre un nuevo proceso).
- `venv/bin/activate`: script que pertenece al entorno virtual `venv` y configura el entorno.
  - Modifica la variable `PATH` para que se use el Python y pip del entorno virtual.
  - Cambia el prompt de la terminal para indicar que el entorno está activo.
  - El entorno virtual `venv` queda activado, y cualquier comando `python` o `pip` que ejecutes usará exclusivamente las librerías de ese entorno, sin afectar al sistema global.

```
# Instala la librería Faker dentro del entorno virtual  
pip install faker
```

- `pip`: gestor de paquetes de Python que permite instalar librerías desde PyPI (Python Package Index).
- `faker`: nombre del paquete a instalar; es una librería que permite generar datos falsos o de prueba, como nombres, direcciones, emails, fechas, etc.

```
# Ejecuta el script Python que genera datos de prueba en la base de datos  
python3 generar_datos.py
```

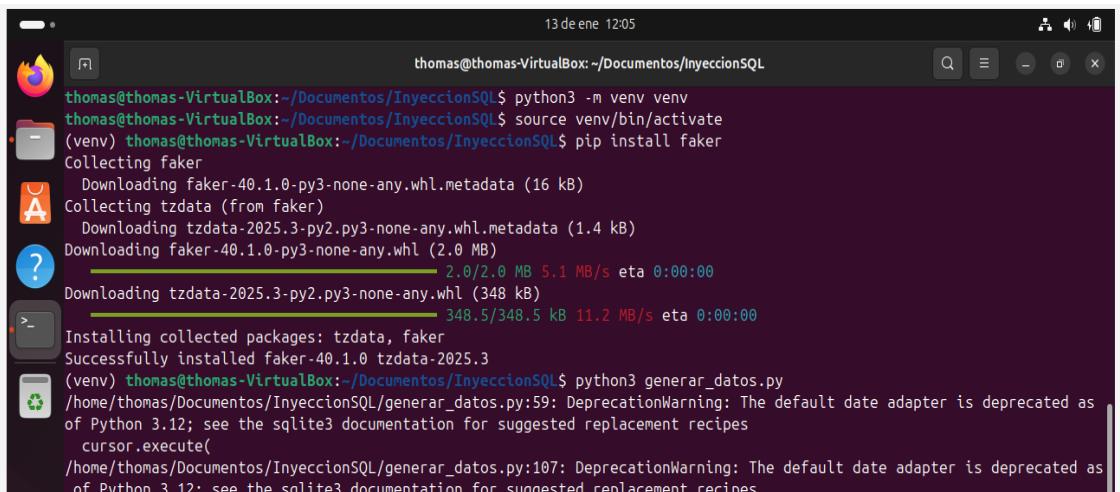
La línea `fake = Faker()` inicializa la librería Faker y crea un generador capaz de producir datos falsos pero realistas, como nombres, fechas, usuarios o contraseñas. Funciona cargando un conjunto amplio de plantillas y algoritmos que imitan patrones reales de información, permitiéndonos generar datos variados sin inventarlos manualmente. Lo usamos porque necesitamos poblar la base de datos con información creíble, coherente y suficiente para que nuestras pruebas —tanto de ataque como de defensa— se parezcan lo máximo posible a un entorno real.

Otro punto importantes es instrucción `cursor.execute("PRAGMA foreign_keys = ON;")` ya que activa el uso de claves foráneas en SQLite, algo que este motor no habilita por defecto. Al hacerlo, garantizamos que todas las relaciones entre tablas se respeten: no podemos insertar condenas para presos inexistentes, ni visitas para IDs que no están en la tabla, ni guardias vinculados a usuarios que no existen. En otras palabras, esta línea obliga a la base de datos a mantener su coherencia interna y evita errores silenciosos que podrían romper la lógica del proyecto.

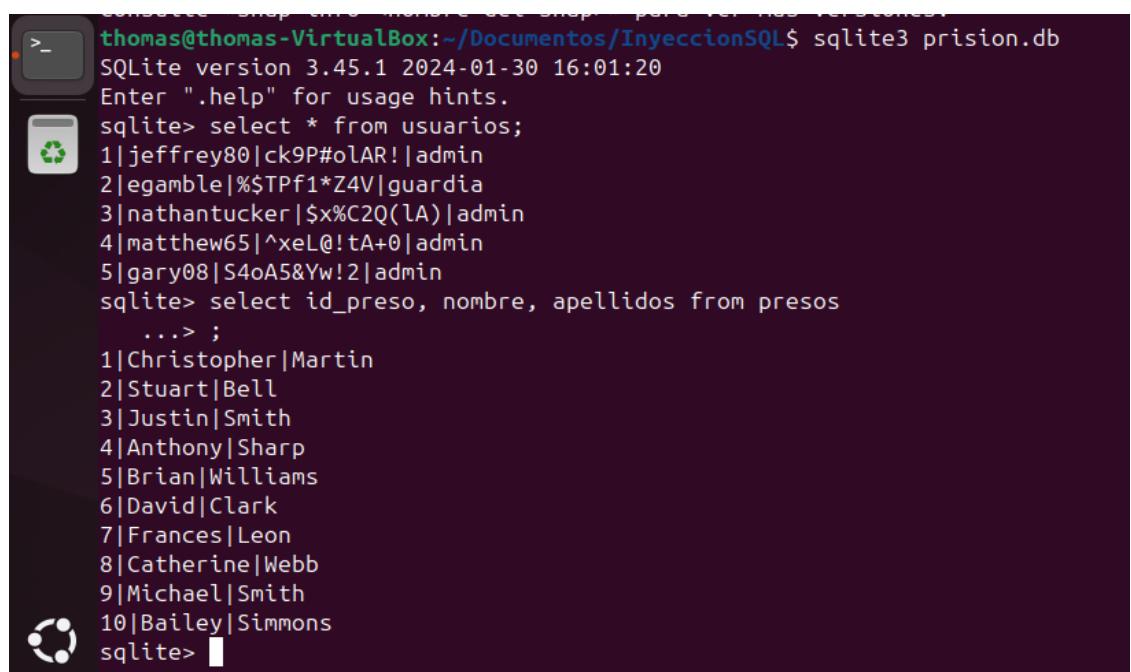
```
# Desactiva el entorno virtual y vuelve al entorno global del sistema
deactivate
```

```
# Abre la base de datos SQLite prision.db en modo interactivo
sqlite3 prision.db
```

```
SELECT * FROM usuarios;
SELECT id_preso, nombre, apellidos FROM presos;
```



```
thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ python3 -m venv venv
thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ source venv/bin/activate
(venv) thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ pip install faker
Collecting faker
  Downloading faker-40.1.0-py3-none-any.whl.metadata (16 kB)
Collecting tzdata (from faker)
  Downloading tzdata-2025.3-py2.py3-none-any.whl.metadata (1.4 kB)
Downloaded faker-40.1.0-py3-none-any.whl (2.0 MB)
  2.0/2.0 MB 5.1 MB/s eta 0:00:00
Downloaded tzdata-2025.3-py2.py3-none-any.whl (348 kB)
  348.5/348.5 kB 11.2 MB/s eta 0:00:00
Installing collected packages: tzdata, faker
Successfully installed faker-40.1.0 tzdata-2025.3
(venv) thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ python3 generar_datos.py
/home/thomas/Documentos/InyeccionSQL/generar_datos.py:59: DeprecationWarning: The default date adapter is deprecated as of Python 3.12; see the sqlite3 documentation for suggested replacement recipes
    cursor.execute(
/home/thomas/Documentos/InyeccionSQL/generar_datos.py:107: DeprecationWarning: The default date adapter is deprecated as of Python 3.12; see the sqlite3 documentation for suggested replacement recipes
```



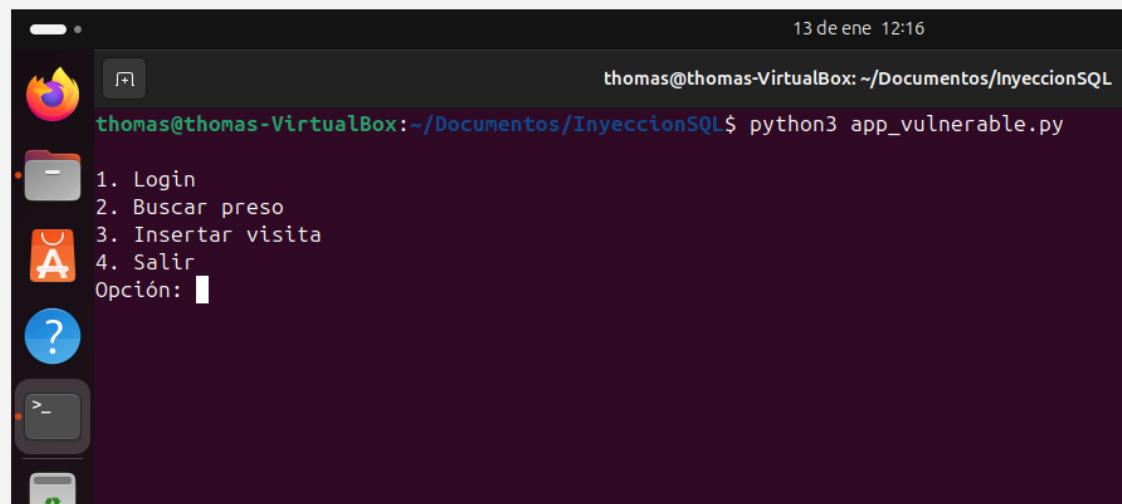
```
thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ sqlite3 prision.db
SQLite version 3.45.1 2024-01-30 16:01:20
Enter ".help" for usage hints.
sqlite> select * from usuarios;
1|jeffrey80|ck9P#oLAR!|admin
2|egamble|%$TPf1*Z4V|guardia
3|nathantucker|$x%C2Q(lA)|admin
4|matthew65|^xeL@!tA+0|admin
5|gary08|S4oA5&Yw!2|admin
sqlite> select id_preso, nombre, apellidos from presos
...> ;
1|Christopher|Martin
2|Stuart|Bell
3|Justin|Smith
4|Anthony|Sharp
5|Brian|Williams
6|David|Clark
7|Frances|Leon
8|Catherine|Webb
9|Michael|Smith
10|Bailey|Simmons
sqlite>
```

Estas instrucciones nos permiten abrir la base de datos prision.db en modo interactivo y consultar directamente su contenido, algo muy útil para verificar que los datos generados por nuestro script se han insertado correctamente. Al ejecutar sqlite3 prision.db accedemos a la consola de SQLite, y con consultas como SELECT \* FROM usuarios; o SELECT id\_preso, nombre, apellidos FROM presos; comprobamos de forma inmediata que las tablas contienen información realista y coherente.

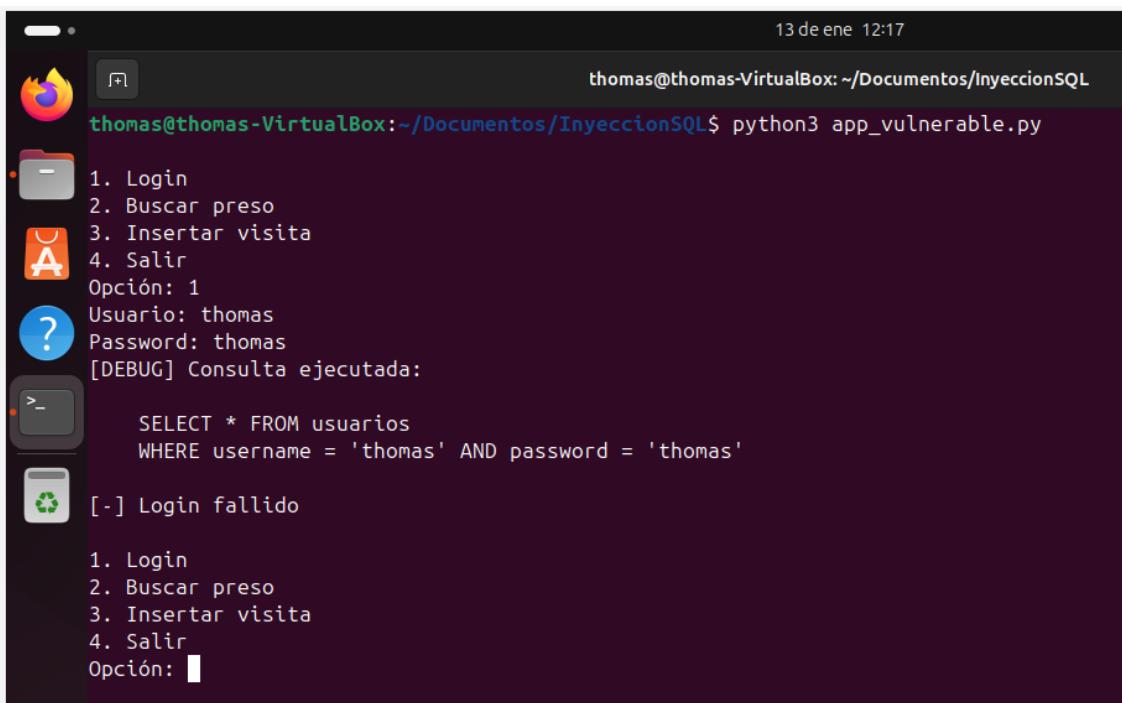
```
# Ejecutamos la aplicación vulnerable escrita en Python 3
python3 app_vulnerable.py
```

[DEBUG] Consulta ejecutada:

```
SELECT * FROM usuarios
WHERE username = 'lo_que_escribiste' AND password = 'lo_que_escribiste'
```



```
thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ python3 app_vulnerable.py
1. Login
2. Buscar preso
3. Insertar visita
4. Salir
Opción: 1
```



```
thomas@thomas-VirtualBox:~/Documentos/InyeccionSQL$ python3 app_vulnerable.py
1. Login
2. Buscar preso
3. Insertar visita
4. Salir
Opción: 1
Usuario: thomas
Password: thomas
[DEBUG] Consulta ejecutada:
SELECT * FROM usuarios
WHERE username = 'thomas' AND password = 'thomas'
[-] Login fallido

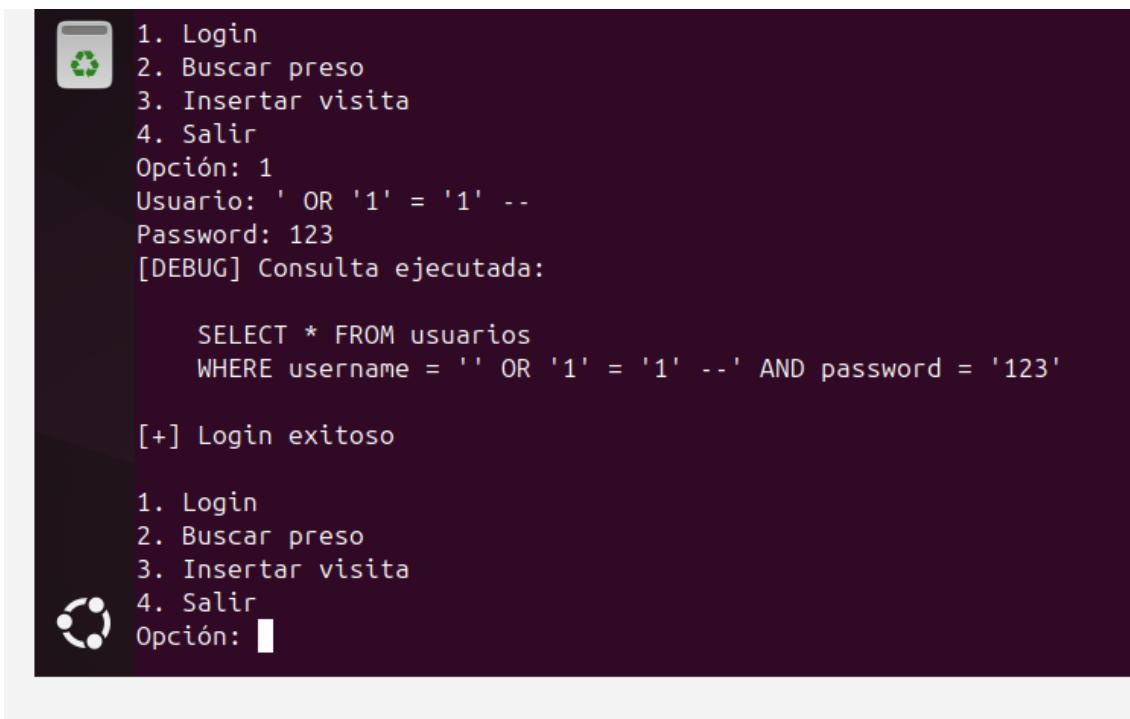
1. Login
2. Buscar preso
3. Insertar visita
4. Salir
Opción: 1
```

El login es fallido porque la consulta SQL compara directamente los valores que introducimos con los campos username y password de la base de datos y no existe ningún registro que coincida exactamente con esos datos. Al no obtener resultados, la aplicación interpreta que nuestras credenciales son incorrectas y nos deniega el acceso.

```
# Utilizamos esta cadena como inyección SQL para forzar que la condición sea
siempre verdadera y comentar el resto de la consulta
' OR '1'='1' --
```

```
SELECT * FROM usuarios
WHERE username = " OR '1'='1' -- ' AND password = 'algo'
```

[+] Login exitoso



```
1. Login
2. Buscar preso
3. Insertar visita
4. Salir
Opción: 1
Usuario: ' OR '1' = '1' --
Password: 123
[DEBUG] Consulta ejecutada:

    SELECT * FROM usuarios
    WHERE username = '' OR '1' = '1' --' AND password = '123'

[+] Login exitoso

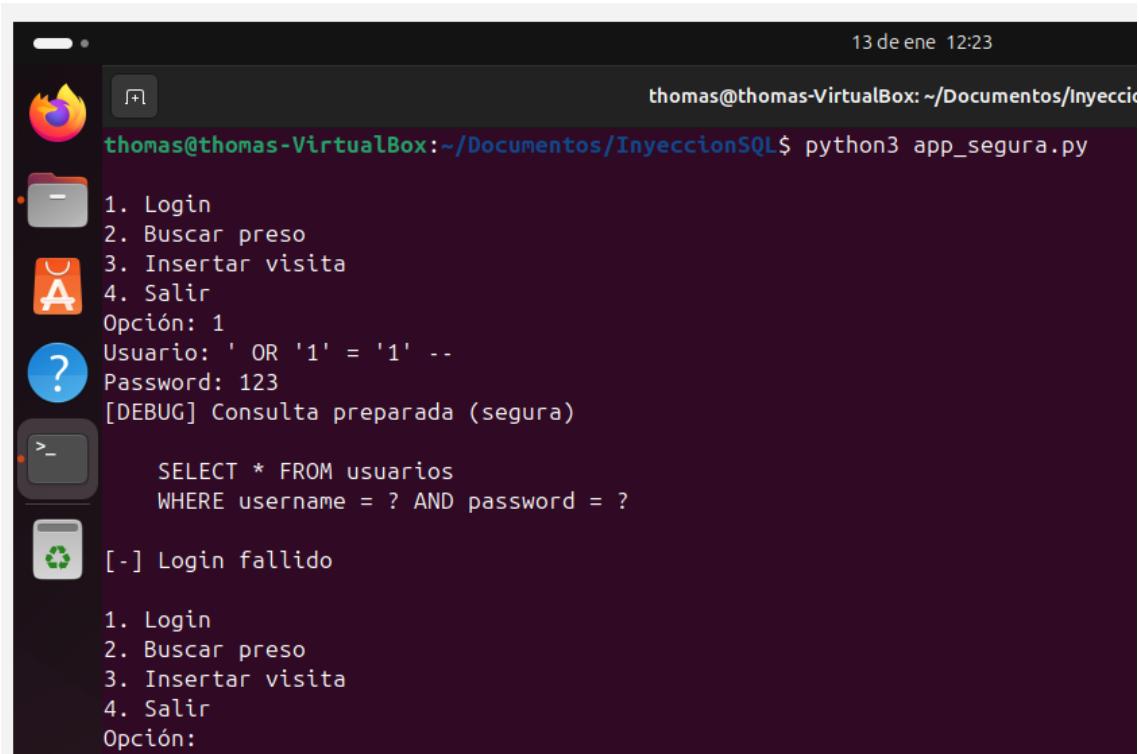
1. Login
2. Buscar preso
3. Insertar visita
4. Salir
Opción: 1
```

El login es exitoso porque al introducir la inyección SQL cerramos la cadena original del campo username y añadimos una condición que siempre se cumple ('1'='1'), haciendo que la cláusula WHERE sea verdadera para todos los registros de la tabla. Además, al usar -- comentamos el resto de la consulta, anulando la verificación de la contraseña. De este modo, la base de datos devuelve resultados aunque no conozcamos credenciales válidas, y la aplicación interpreta erróneamente que hemos iniciado sesión correctamente.

Es importante hacerlo desde Blue porque así simulamos un ataque real desde una máquina externa, actuando como un atacante que no tiene acceso directo al servidor ni a la base de datos. Al ejecutar la inyección SQL desde Blue comprobamos que la vulnerabilidad es explotable de forma remota, tal y como ocurriría en un entorno real, y no solo mediante pruebas internas. De esta manera validamos el impacto real del fallo de seguridad, entendemos mejor el riesgo para la aplicación y reforzamos el enfoque práctico de ciberseguridad al pensar como un atacante para poder defender correctamente el sistema.

```
# Ejecutamos la aplicación vulnerable escrita en Python 3
python3 app_segura.py
```

```
' OR '1'='1'-
[-] Login fallido
```

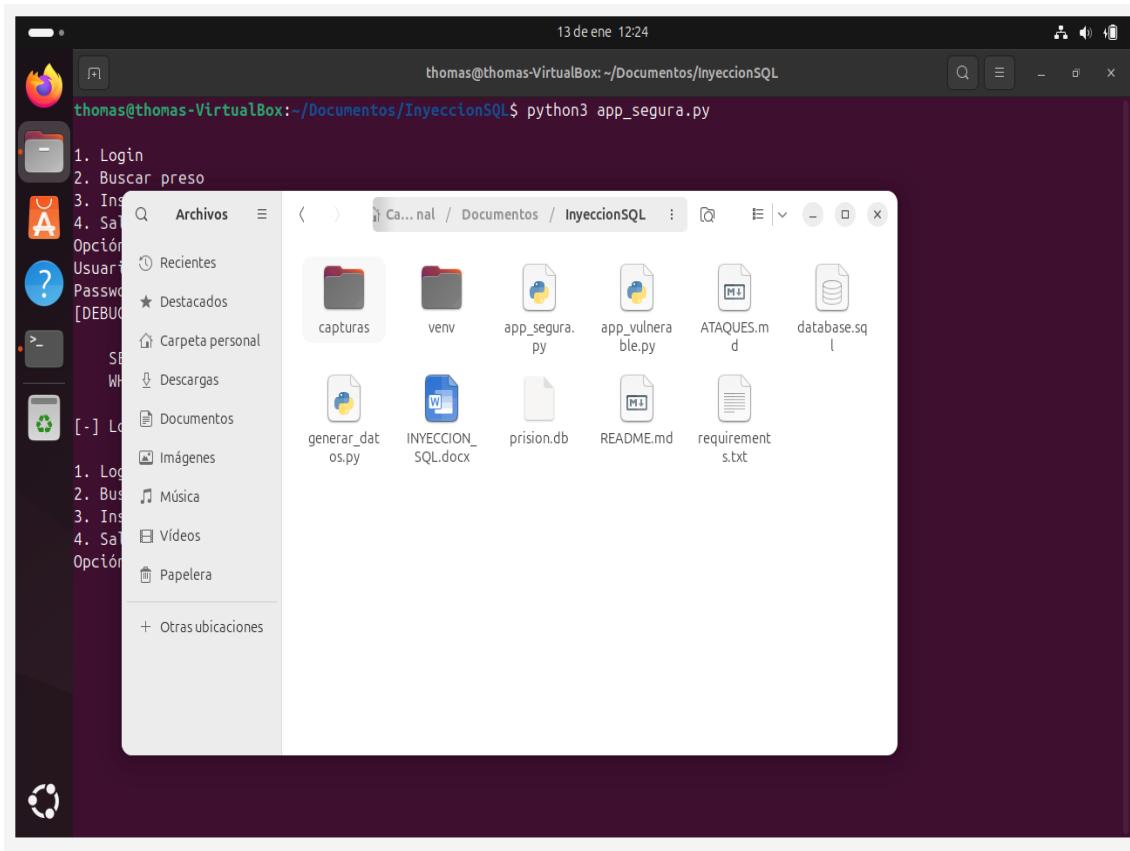


The screenshot shows a terminal window on a Linux desktop environment. The title bar indicates it's running on a VirtualBox machine. The command entered is `python3 app_segura.py`. The application displays a menu with options 1 through 4. The user selects option 1 (Login) and enters the credentials "Usuario: ' OR '1' = '1' --" and "Password: 123". The application logs the debug message "[DEBUG] Consulta preparada (segura)". It then executes the SQL query `SELECT * FROM usuarios WHERE username = ? AND password = ?`. Finally, it outputs the error "[ - ] Login fallido".

El login es fallido porque la aplicación segura utiliza consultas parametrizadas, lo que impide que la entrada del usuario se interprete como parte de la sentencia SQL. Cuando introducimos la cadena de inyección, el sistema la trata únicamente como un valor literal, no como código ejecutable, por lo que no altera la lógica de la consulta ni fuerza condiciones verdaderas. De esta forma, la validación de usuario y contraseña se realiza correctamente y la base de datos no devuelve resultados no autorizados.

Además, la aplicación aplica buenas prácticas de seguridad, como la validación de entradas y la separación entre datos y código, lo que elimina la posibilidad de manipular la consulta SQL. Aunque intentemos introducir operadores lógicos o comentarios, estos no tienen efecto, ya que el motor de base de datos recibe una consulta segura y estructurada, garantizando que solo se permita el acceso con credenciales válidas.

Es como intentar abrir una puerta segura usando una llave falsa escrita en un papel: aunque la forma parezca correcta, la cerradura solo reconoce llaves auténticas y no se deja engañar por trucos o apariencias, por lo que el acceso es rechazado.



Antes de pasar al trabajo del Red Team, cerramos nuestra fase como Blue Team con la tranquilidad de haber construido una base sólida y coherente. Hemos preparado una aplicación funcional, una base de datos completa y un entorno que refleja situaciones reales, lo que nos permite afrontar la siguiente etapa con una visión clara de cómo debe comportarse un sistema bien estructurado antes de ser puesto a prueba.

En esta fase azul hemos creado la estructura completa del proyecto: la base de datos con todas sus tablas y relaciones, el script que genera datos realistas, la aplicación vulnerable y su versión segura, así como la documentación interna que explica cada componente. Todo este trabajo nos permite comprender el sistema desde dentro, anticipar posibles fallos y establecer un punto de comparación entre lo que es una aplicación insegura y lo que debería ser una aplicación correctamente protegida.

En los archivos del proyecto encontraremos elementos esenciales para comprender y ejecutar todo lo que hemos construido. El README.md ofrecerá una guía clara sobre cómo iniciar la aplicación, cómo funciona cada parte y qué pasos seguir para reproducir el entorno. El requirements.txt contendrá todas las dependencias necesarias para ejecutar el proyecto sin errores, asegurando que cualquiera pueda instalarlo fácilmente. Finalmente, el archivo database.sql incluirá la estructura completa de la base de datos, con todas las tablas, claves primarias y foráneas, permitiendo reconstruir el entorno desde cero cuando sea necesario.

## FASE 2: ATAQUES ÉTICOS Y DOCUMENTACIÓN (RED TEAM)

**OBJETIVO:** Analizar y explotar vulnerabilidades en la app de otro equipo, documentar los resultados y validar la resistencia de la versión segura.

### HERRAMIENTAS CLAVE:

- **Kali Linux (VBox)** → entorno ofensivo
- **DVWA / Hack The Box / OverTheWire** → entrenamiento previo y práctica legal
- **OWASP** → payloads y vectores de ataque
- **Python 3** → scripts auxiliares si son necesarios
- **Markdown** → documentación en ATAQUES.md

### ENTREGABLES EN LA ESTRUCTURA DEL PROYECTO:

Linux Desktop (Documentos)  
— ATAQUES.md  
— capturas/  
    – exploit\_login\_bypass.png  
    – exploit\_data\_extraction.png

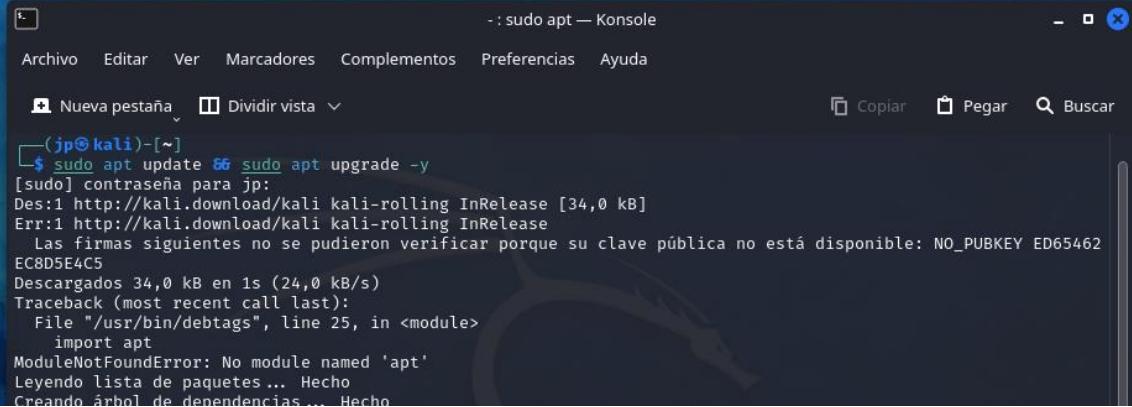
### RED TEAM

En la fase del Red Team asumimos el papel de atacantes éticos y analizamos la aplicación vulnerable creada por el Blue Team para descubrir fallos reales de seguridad. Exploramos cada funcionalidad buscando puntos débiles: entradas sin validar, consultas SQL inseguras, formularios manipulables, rutas expuestas o comportamientos inesperados. Nuestro objetivo es pensar como un atacante, probar técnicas de explotación y demostrar cómo un sistema aparentemente funcional puede romperse con acciones muy simples si no está protegido correctamente.

Una vez identificamos las vulnerabilidades, las explotamos de forma controlada para obtener información sensible, modificar datos, acceder a funciones restringidas o comprometer el sistema. Documentamos cada ataque con claridad: qué vulnerabilidad se encontró, cómo se explotó, qué impacto tuvo y qué debería corregirse.

Esta fase es esencial porque ofrece una visión realista de los riesgos y permite al Blue Team reforzar la aplicación basándose en pruebas concretas, no en suposiciones.

```
# Actualiza la lista de paquetes disponibles desde los repositorios  
sudo apt update && sudo apt upgrade -y
```



```
(jp㉿kali)-[~]  
$ sudo apt update && sudo apt upgrade -y  
[sudo] contraseña para jp:  
Des:1 http://kali.download/kali kali-rolling InRelease [34,0 kB]  
Err:1 http://kali.download/kali kali-rolling InRelease  
  Las firmas siguientes no se pudieron verificar porque su clave pública no está disponible: NO_PUBKEY ED65462  
EC8D5E4C5  
Descargados 34,0 kB en 1s (24,0 kB/s)  
Traceback (most recent call last):  
  File "/usr/bin/debtags", line 25, in <module>  
    import apt  
ModuleNotFoundError: No module named 'apt'  
Leyendo lista de paquetes... Hecho  
Creando árbol de dependencias... Hecho
```

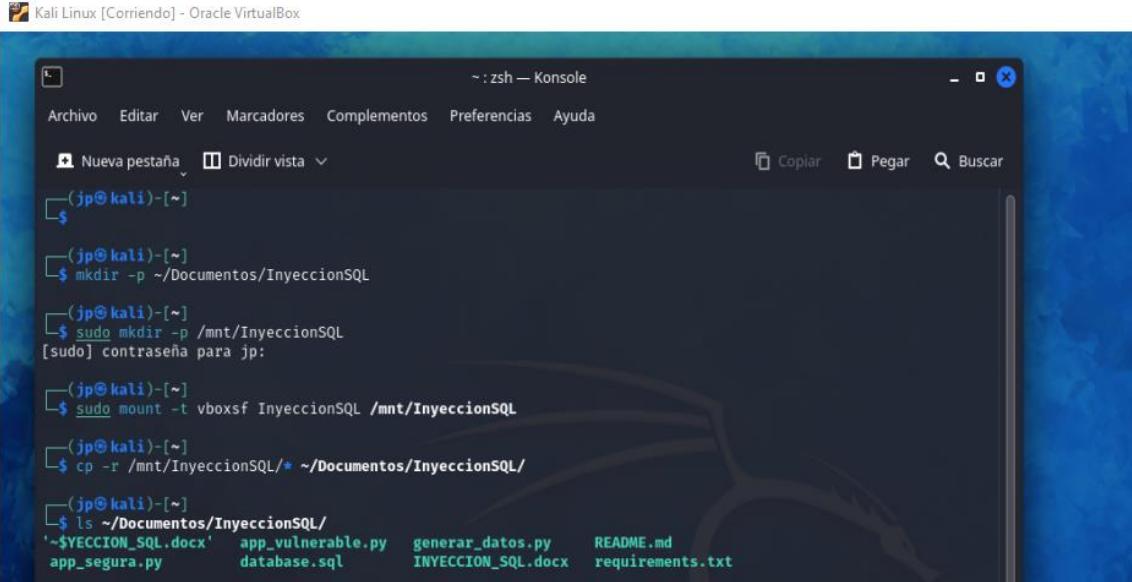
```
# Creamos el directorio InyeccionSQL dentro de Documentos si no existe  
mkdir -p ~/Documentos/InyeccionSQL
```

```
# Creamos el punto de montaje para la carpeta compartida  
sudo mkdir -p /mnt/InyeccionSQL
```

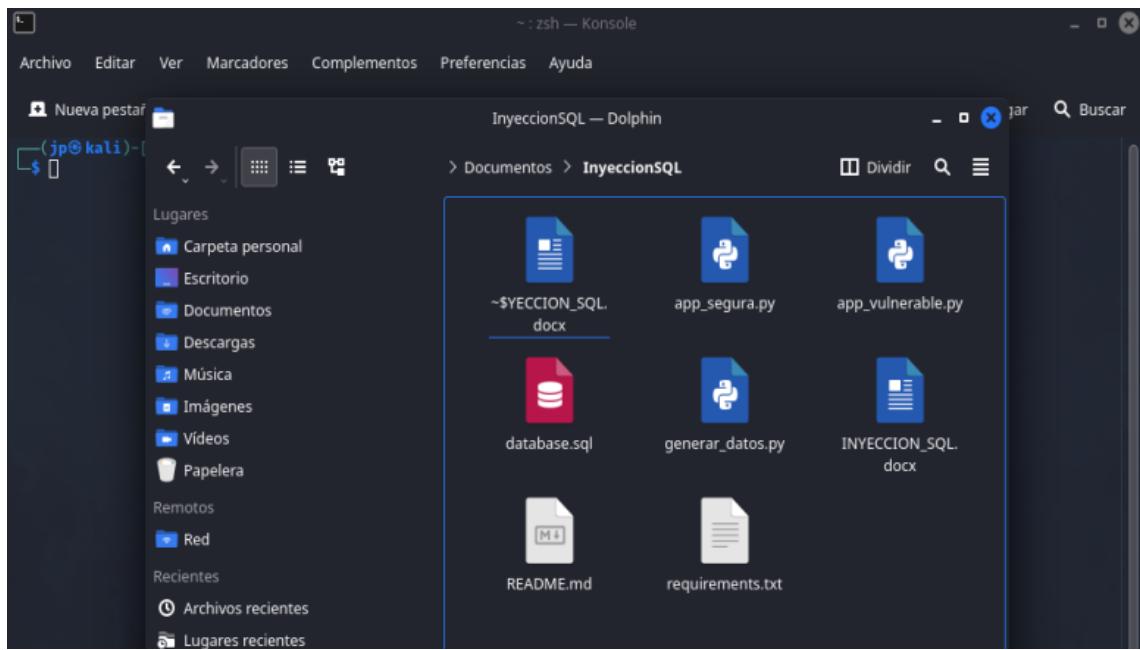
```
# Montamos la carpeta compartida de VirtualBox llamada "InyeccionSQL"  
sudo mount -t vboxsf InyeccionSQL /mnt/InyeccionSQL
```

```
# Copiamos todo el contenido de la carpeta compartida al directorio local  
cp -r /mnt/InyeccionSQL/* ~/Documentos/InyeccionSQL/
```

```
# Listamos el contenido del directorio InyeccionSQL para verificar la copia  
ls ~/Documentos/InyeccionSQL
```



```
Kali Linux [Corriendo] - Oracle VirtualBox  
~ : zsh — Konsole  
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda  
+ Nueva pestaña Dividir vista ~ : zsh — Konsole  
Copiar Pegar Buscar  
(jp㉿kali)-[~]  
$  
(jp㉿kali)-[~]  
$ mkdir -p ~/Documentos/InyeccionSQL  
(jp㉿kali)-[~]  
$ sudo mkdir -p /mnt/InyeccionSQL  
[sudo] contraseña para jp:  
(jp㉿kali)-[~]  
$ sudo mount -t vboxsf InyeccionSQL /mnt/InyeccionSQL  
(jp㉿kali)-[~]  
$ cp -r /mnt/InyeccionSQL/* ~/Documentos/InyeccionSQL/  
(jp㉿kali)-[~]  
$ ls ~/Documentos/InyeccionSQL/  
'~$VIECCION_SQL.docx' app_vulnerable.py generar_datos.py README.md  
app_segura.py database.sql INYECCION_SQL.docx requirements.txt
```



```
# Nos desplazamos al directorio InyeccionSQL dentro de Documentos  
cd ~/Documentos/InyeccionSQL
```

```
# Visualizamos el contenido del archivo app_vulnerable.py usando less  
less app_vulnerable.py
```

```
# Abrimos el archivo app_vulnerable.py con el editor nano para editararlo  
nano app_vulnerable.py (opcional)
```

A screenshot of a terminal window titled 'InyeccionSQL : less — Konsole'. The window title bar also includes 'Kali Linux [Corriendo] - Oracle VirtualBox'. The terminal content displays the code of the 'app\_vulnerable.py' script. The code uses SQLite3 to connect to a database named 'prision.db'. It defines a 'login()' function that takes 'username' and 'password' as input, connects to the database, and executes a query. The query is vulnerable to SQL injection because it concatenates user input directly into the query string. The code then prints the executed query and the results. The terminal interface includes standard Linux navigation keys like 'Copiar' (Copy) and 'Peg' (Paste).

## **ATAQUE**

La inyección sql es una de las vulnerabilidades más críticas en aplicaciones que interactúan con bases de datos, ya que permite a un atacante interferir directamente en las consultas sql ejecutadas por el sistema. este tipo de fallo se produce cuando los datos introducidos por el usuario no se validan ni se gestionan de forma segura, lo que abre la puerta a la modificación de la lógica interna de las consultas y al acceso no autorizado a la información.

En aplicaciones con funciones de autenticación, búsqueda o inserción de datos, una mala implementación de las consultas sql puede comprometer por completo la confidencialidad, integridad y disponibilidad de la base de datos.

a continuación, analizamos distintos escenarios de ataque basados en inyección sql, mostrando cómo un atacante puede aprovechar estas debilidades para eludir controles de acceso, extraer información sensible o manipular datos críticos del sistema.

### **Bypass De Autenticación (Login)**

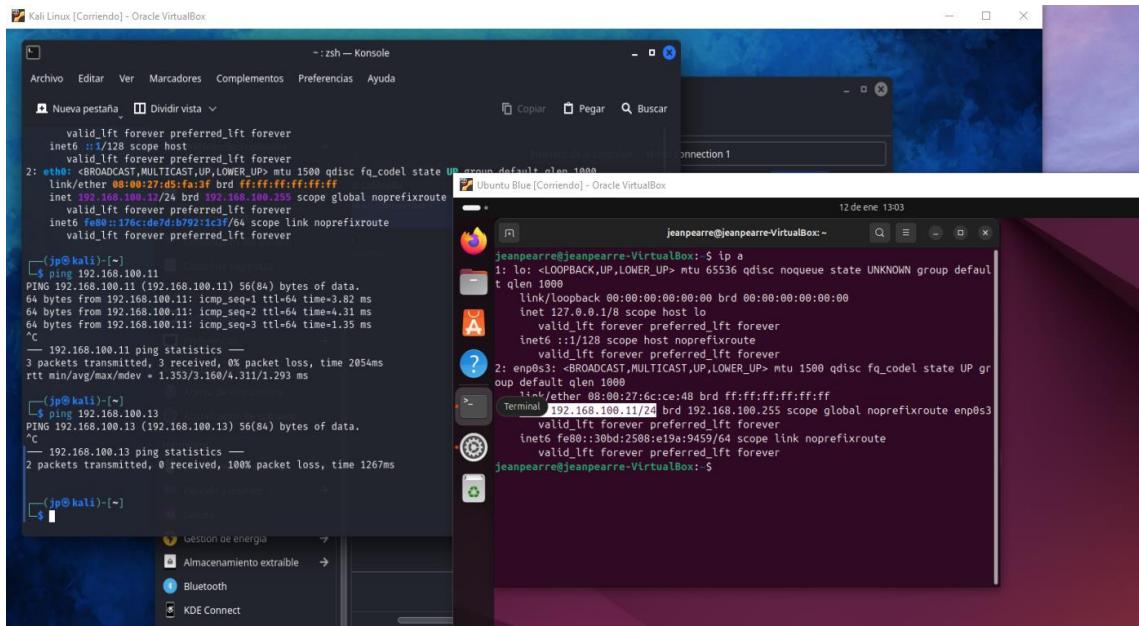
La función login() es vulnerable porque las variables **usuario** y **password**, controladas completamente por el usuario, se insertan directamente en la consulta SQL sin validación ni uso de consultas parametrizadas, lo que permite a un atacante modificar la lógica del WHERE y ejecutar inyección SQL, posibilitando el bypass del login, el acceso sin credenciales válidas y el compromiso total del sistema de autenticación.

### **Extracción De Datos (Sql Injection En Búsqueda)**

La función buscar\_preso() es vulnerable porque el parámetro **termino**, introducido directamente por el usuario, se inserta sin sanitización ni consultas parametrizadas y se reutiliza dentro de una condición OR, lo que facilita la creación de expresiones siempre verdaderas, permitiendo ataques de inyección SQL que pueden derivar en la extracción masiva de datos, la enumeración completa de la tabla **presos** y el acceso a información sensible almacenada.

### **Ataque: Manipulación De Datos (Sql Injection En Insert)**

La función insertar\_visita() es vulnerable porque los valores **id\_preso**, **visitante** y **fecha**, controlados totalmente por el usuario, se insertan directamente en una consulta INSERT sin validación de tipos, formato ni contenido ni uso de consultas parametrizadas, lo que permite inyección SQL capaz de provocar inserción de datos falsos, alteración de registros y corrupción de la integridad de la base de datos.

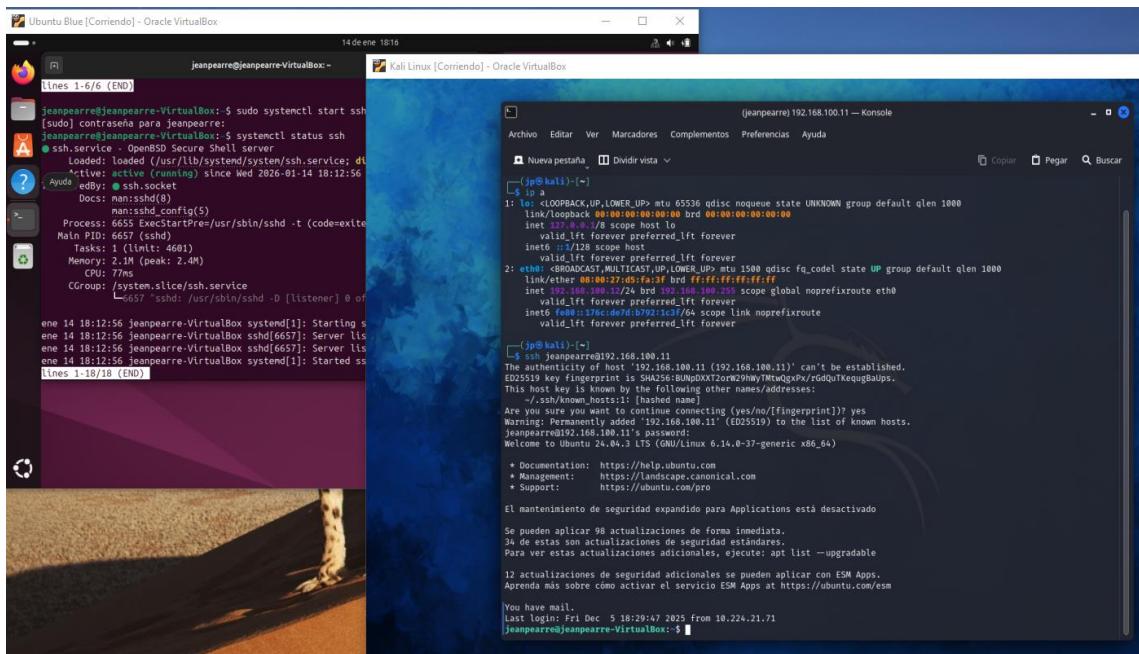


## Blue Team Habilita SSH

```
# Instalamos el servicio OpenSSH Server
sudo apt install openssh-server
# Habilitamos el servicio SSH para que se inicie automáticamente
sudo systemctl enable ssh
# Iniciamos el servicio SSH para que empiece a aceptar conexiones
sudo systemctl start ssh
```

## Red Team se conecta por SSH

```
ssh usuario_blue@IP_BLUE
```



## Red Team Ejecuta La App Vulnerable

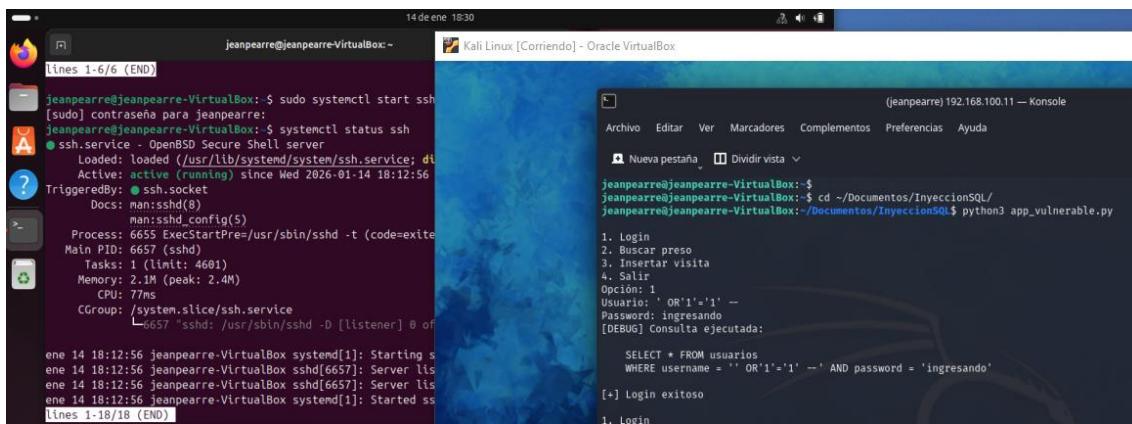
```
cd ~/Documentos/InyeccionSQL  
python3 app_vulnerable.py
```

Opción: 1

Usuario: ' OR '1'='1' --

Password: (ingresando)

[+] Login exitoso



En este escenario, nos conectamos desde el Red Team al sistema del Blue Team mediante SSH, comprobando que el servicio de acceso remoto está activo y accesible. Una vez dentro del sistema, ejecutamos la aplicación vulnerable y seleccionamos la opción de inicio de sesión, introduciendo una cadena maliciosa en el campo usuario que modifica la consulta SQL original. Al inyectar ' OR '1'='1' --, forzamos que la condición del WHERE sea siempre verdadera y comentamos el resto de la consulta, anulando la validación de la contraseña.

Como consecuencia, la base de datos devuelve registros válidos sin que introduzcamos credenciales legítimas, y la aplicación interpreta este resultado como un inicio de sesión exitoso. Este bypass de autenticación demuestra una vulnerabilidad crítica de inyección SQL, ya que nos permite acceder de forma no autorizada al sistema y compromete por completo el mecanismo de autenticación y la seguridad de los datos gestionados.

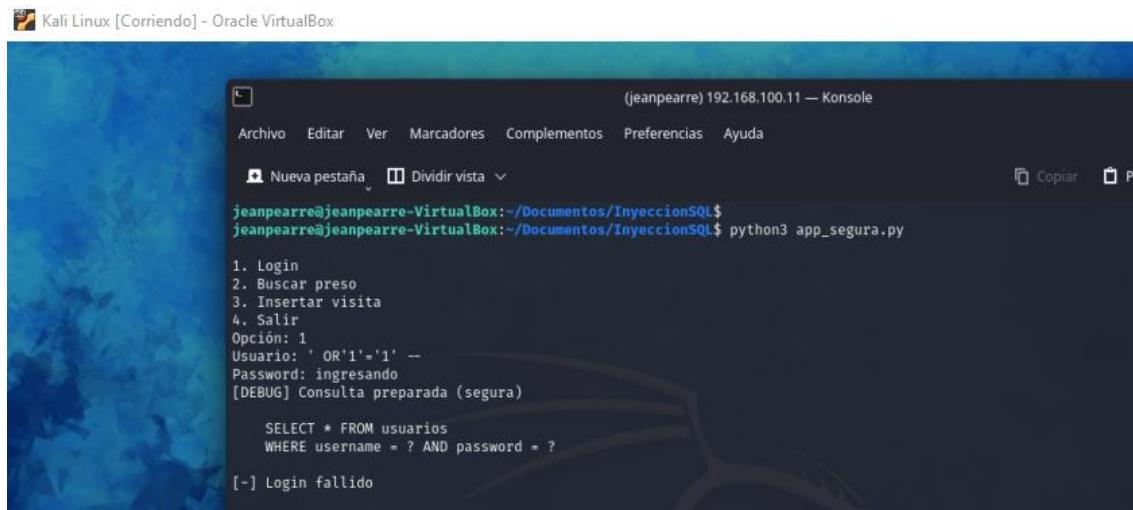
Nuestro código es vulnerable porque en todas las funciones construimos las consultas SQL mediante concatenación directa de datos introducidos por el usuario, sin validación ni uso de consultas parametrizadas, lo que permite inyección SQL. Tanto en `login()`, como en `buscar_preso()` e `insertar_visita()`, los valores introducidos se insertan directamente en las sentencias SELECT e INSERT, permitiéndonos modificar la lógica del WHERE, forzar condiciones siempre verdaderas, extraer información sensible o manipular datos de la base de datos. Esta mala práctica compromete la confidencialidad, integridad y seguridad del sistema completo, ya que cualquier usuario malintencionado puede alterar el comportamiento esperado de la aplicación.

```
# Ejecutamos la aplicación vulnerable escrita en Python 3
python3 app_segura.py
```

```
' OR '1'='1' -
Password: (ingresando)
```

```
SELECT * FROM usuarios
WHERE username = " OR '1'='1' AND password = 'hola'
```

[+] Login fallido



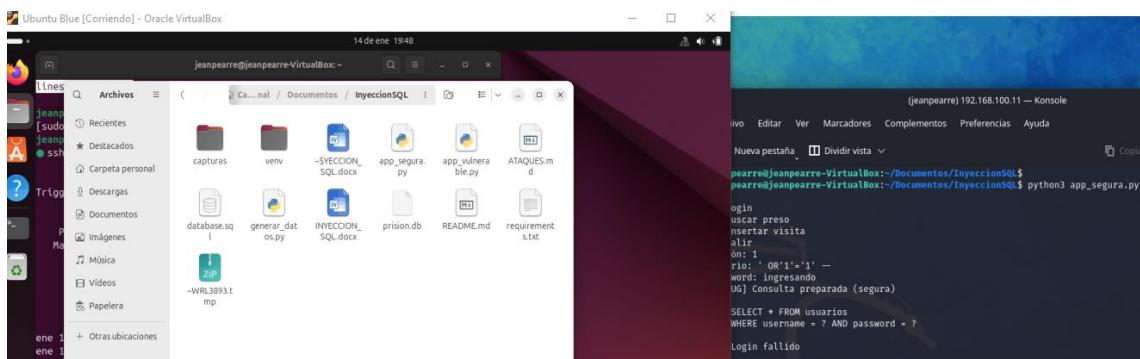
The screenshot shows a terminal window titled '(jeanpearre) 192.168.100.11 — Konsole'. The user has run the command 'python3 app\_segura.py'. The application presents a menu with options 1 through 4. The user selects option 1, which triggers a SQL injection attack. The application sends the query 'SELECT \* FROM usuarios WHERE username = ? AND password = ?' to the database. The terminal shows the resulting error message: '[+] Login fallido' (Login failed).

En este tipo de ataque intentamos explotar vulnerabilidades de **inyección SQL** introduciendo entradas maliciosas en los campos de la aplicación, como '`' OR '1'='1'`', con el objetivo de alterar la lógica de las consultas SQL y forzar condiciones siempre verdaderas. De esta manera, buscamos eludir los mecanismos de autenticación o acceder a información sin disponer de credenciales válidas, comprobando si la aplicación concatena directamente los datos del usuario en las consultas.

Durante la ejecución del ataque, verificamos si la aplicación interpreta nuestras entradas como parte del código SQL en lugar de tratarlas como datos. Cuando la aplicación es vulnerable, conseguimos bypass de login, extracción de datos o manipulación de registros; en cambio, si el intento falla y el login es rechazado, confirmamos que la inyección no ha surtido efecto y que la lógica de la consulta no ha sido alterada.

Ahora la aplicación es segura porque utilizamos consultas parametrizadas, lo que hace que los valores introducidos por el usuario se traten estrictamente como datos y no como código SQL. Al usar marcadores de posición (?) y pasar los parámetros por separado en `cursor.execute()`, evitamos que las entradas maliciosas modifiquen la estructura de la consulta, bloqueando la inyección SQL y protegiendo la autenticación, la consulta de datos y la integridad de la base de datos.

## FASE 3: BLUE & RED – ENTREGA Y ANALOGÍA



El proyecto completo BLUE–RED sobre Inyección SQL nos ha permitido recorrer el ciclo completo de seguridad de una aplicación.

En la fase Blue Team diseñamos la base de datos, generamos datos realistas, implementamos la aplicación vulnerable y también su versión segura, entendiendo cómo deben construirse las consultas, cómo proteger la entrada del usuario y cómo mantener la integridad del sistema. Esta parte nos dio la visión del desarrollador y del defensor: cómo debería funcionar todo cuando se hace bien. En la fase Red Team adoptamos la mentalidad opuesta: la del atacante que busca fallos, inconsistencias y puntos débiles. Analizamos la función de login, las consultas sin parametrizar, los formularios sin validación y las rutas expuestas, demostrando cómo una simple concatenación de texto puede abrir la puerta a una inyección SQL capaz de saltarse la autenticación o manipular datos. Esta etapa nos permitió ver el impacto real de los errores del Blue Team y entender por qué las buenas prácticas no son opcionales, sino esenciales para la seguridad de cualquier sistema.

Todo lo que hemos visto sería imaginar que construimos una casa: el Blue Team se encarga de levantar las paredes, instalar las puertas y colocar las cerraduras, asegurándose de que todo esté bien montado y funcione como debe. El Red Team, en cambio, actúa como un ladrón ético que prueba si las ventanas cierran bien, si alguna puerta quedó sin seguro o si es posible forzar una cerradura defectuosa. Solo cuando ambos equipos han hecho su trabajo podemos decir que la casa es realmente segura.

Thomas Rodriguez N.  
Jeanpearre Leon S.