**Annex C: Individual Project Report**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student’s Name** | Ko Jia Ling | **Admin No** | 190681D |
| **Course** | Diploma in Cyber Security and Digital Forensics | **Acad / Sem** | 2020S1 |
| **Module Name** | IT2555 Applications Security Project | **Module Group** | 03 |
| **Module Supervisor** | Ms Verawaty | | |
| **Team Leader** | Siti Sarah Binte Sa'ad Bagarib | **Team No** | 1 |

# 1. Project Description

For this project, we have developed a vulnerable and secure version of an online forum named “Lorem Ipsum”.

Lorem Ipsum is an online forum that allows users to communicate with each other on a variety of subjects. They will be able to read all the threads on a particular subject or discussion, as well as contribute their own posts. Given the nature of the forum, where it is simple to make a post that will be read by other users, there is a need for administrators to vet through messages and take appropriate action to supervise the community.

The vulnerable version of Lorem Ipsum contains the implementations of the 6 out of the Top 10 OWASP Web Vulnerabilities that have been chosen by our team. The team members as well as our chosen vulnerabilities are as follows:

* Sarah Bagarib – Broken Authentication, Security Misconfiguration
* Muhammad Ennaayattulla – Sensitive Data Exposure, Cross Site Scripting
* Ko Jia Ling – Broken Access Control, Injection

The secure version of Lorem Ipsum is mitigated against the 6 chosen vulnerabilities by the respective member in charge. Additional mitigation against a few of the other vulnerabilities found during the analysis of the vulnerable version of Lorem Ipsum has been implemented as well.

# 2. Individual Task

The 2 OWASP Top 10 Web Vulnerabilities I have chosen are broken access control and injection. In addition, I have implemented a monitoring and logging system for the secure version of Lorem Ipsum. The following documents the implementations of my chosen vulnerabilities in the vulnerable version of Lorem Ipsum, as well as the solutions used to fix the vulnerability in the secure version of Lorem Ipsum.

## Vulnerability 1: Broken Access Control

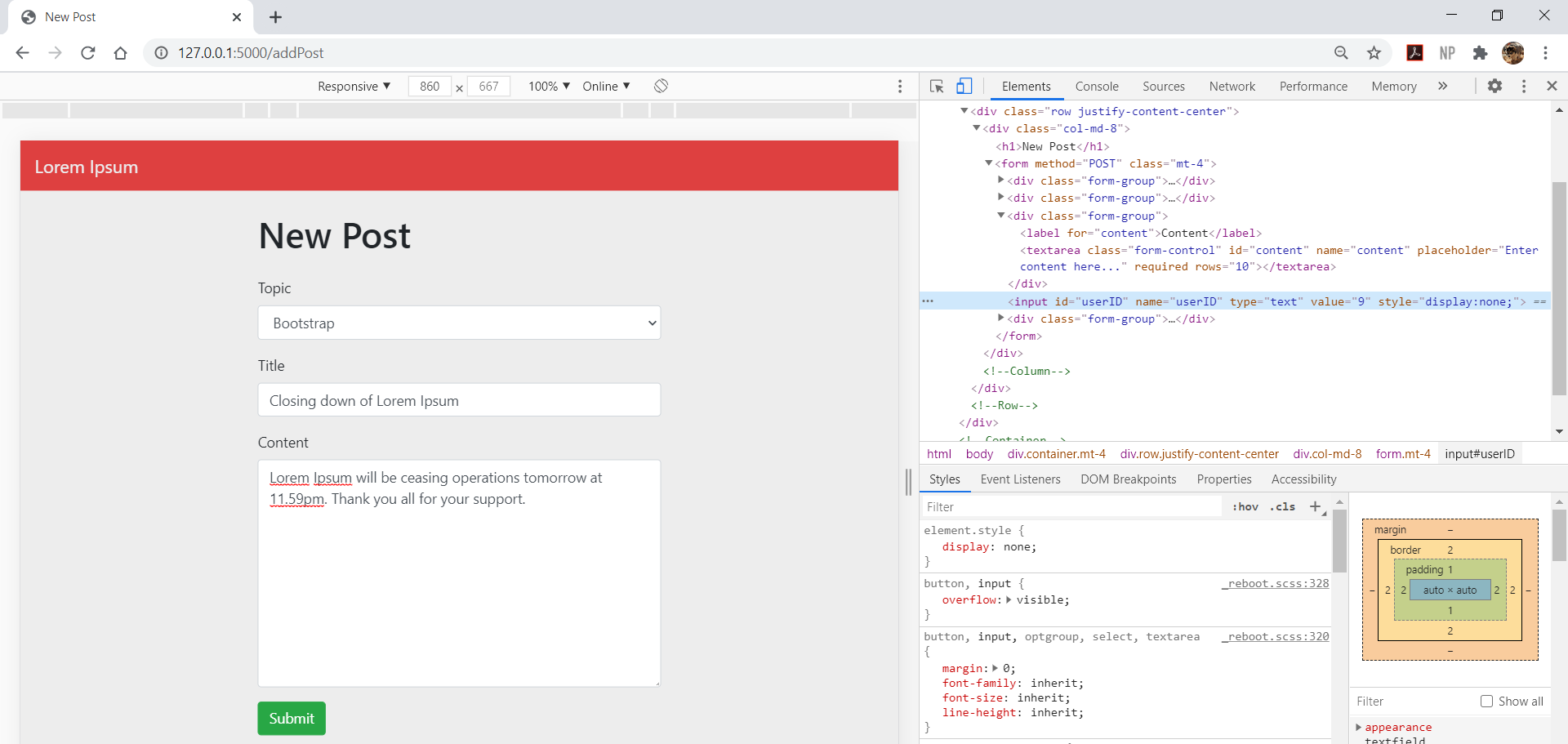
### Implementation 1: Pretending to be another user

On pages where a user’s identity is required (creating new post, submitting feedback, commenting or replying), a hidden form field is used to store the user’s associated User ID. Attackers can find the hidden field and change the value by inspecting the source code. This allows attackers to trick the system into believing that the post was created by a different user.

The following example demonstrates how an attacker (using account “hanbaobao”, User ID = 9) may forge a post under another user’s account (“NotABot”, User ID = 1).

Step 1. Attacker inspects page source code to find the hidden field used to indicate the User ID.

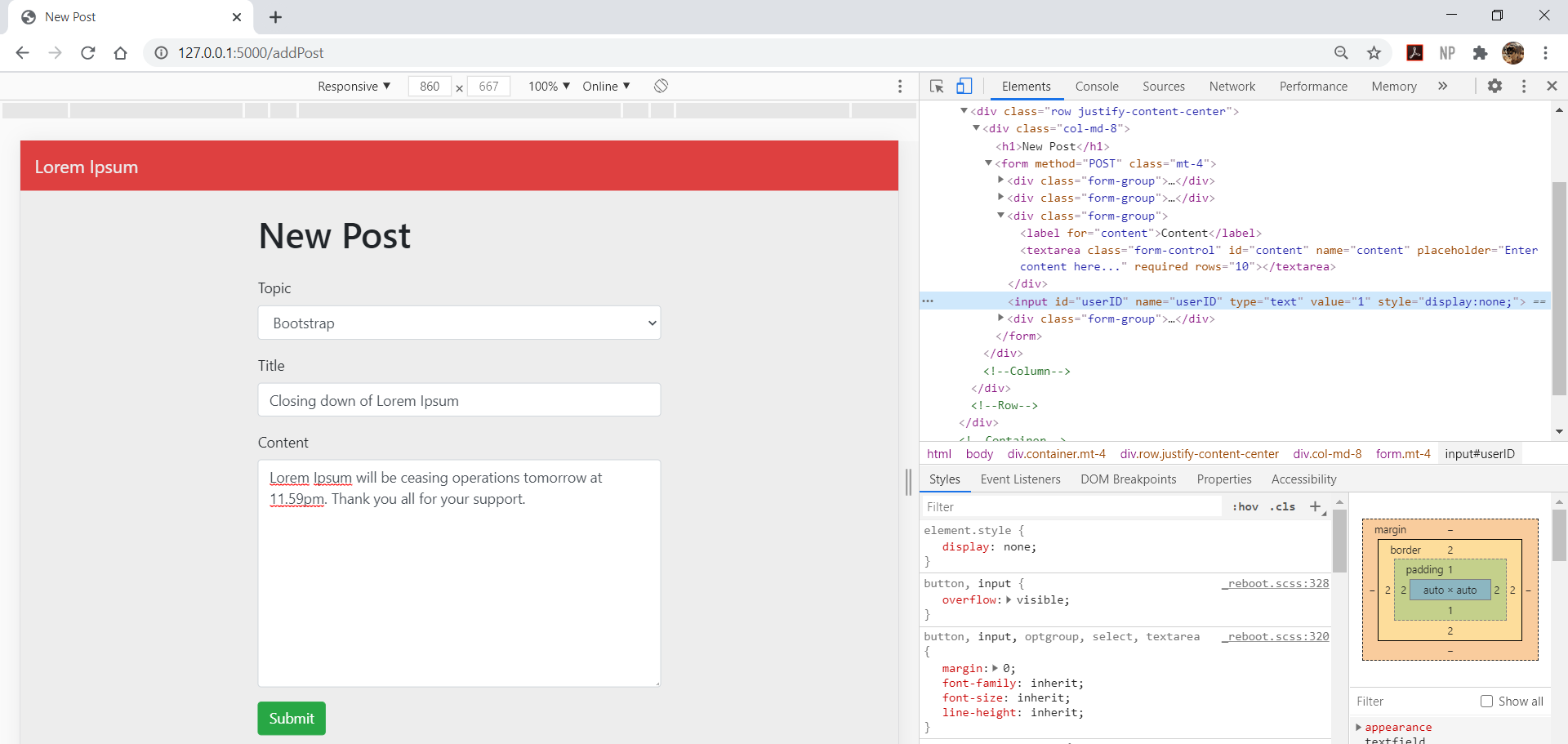
#### Image 1.1.1 “Add Post” page with source code.



**Hidden input field for User ID**

Step 2. Modify the value of hidden User ID field.

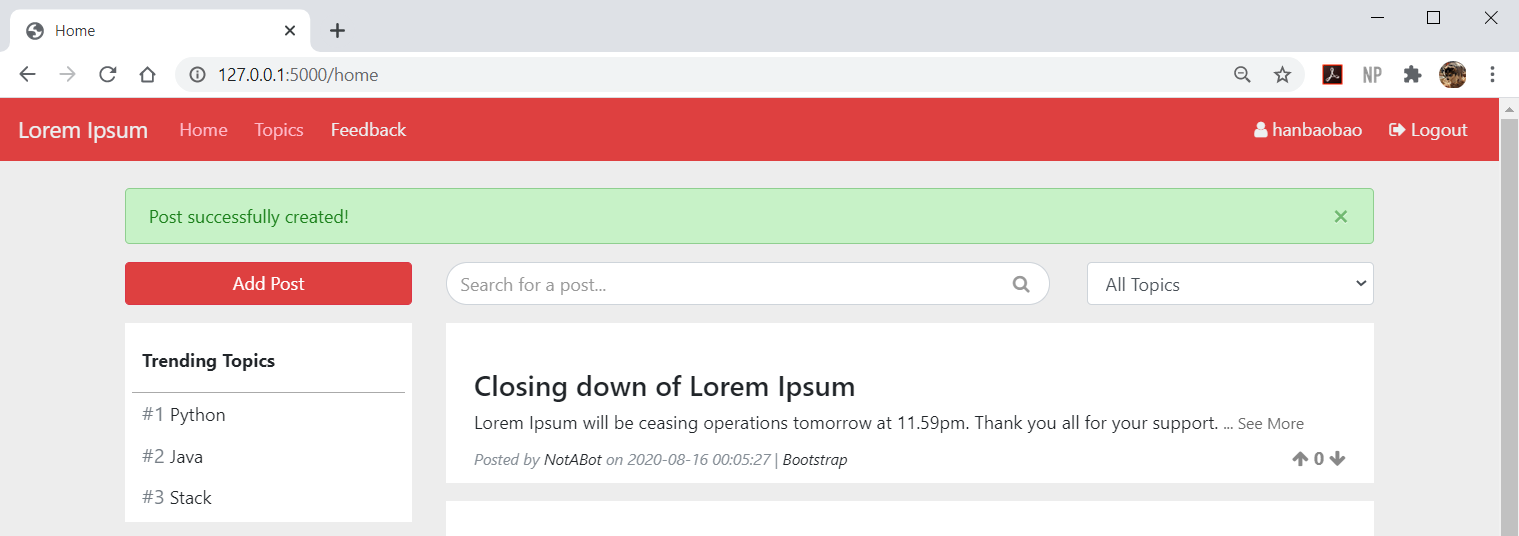
#### Image 1.1.2 “Add Post” page with modified source code.



**Attacker modifies the User ID value**

Step 3. Submit post. A new post is created and the system recognizes it as being created by “NotABot” (User ID = 1) instead of the actual user “hanbaobao” (User ID=9)

#### Image 1.1.3 Home page after submitting post.



**Successfully created post under another user’s account**

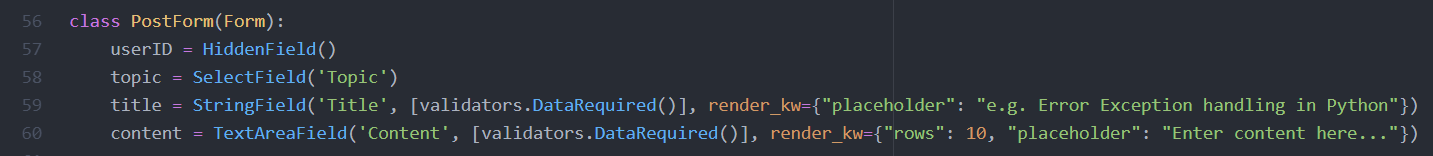
**Account that the attacker is actually using**

### Solution 1: Pretending to be another user

WTForm Hidden Field has a built-in security feature that ignores any changes made to the value of the hidden field.

Modifying the value of the hidden field in the source code, or intercepting the post request and modifying the value of the user ID parameter, will not affect the value of the hidden field that WTForm will pass to the server.

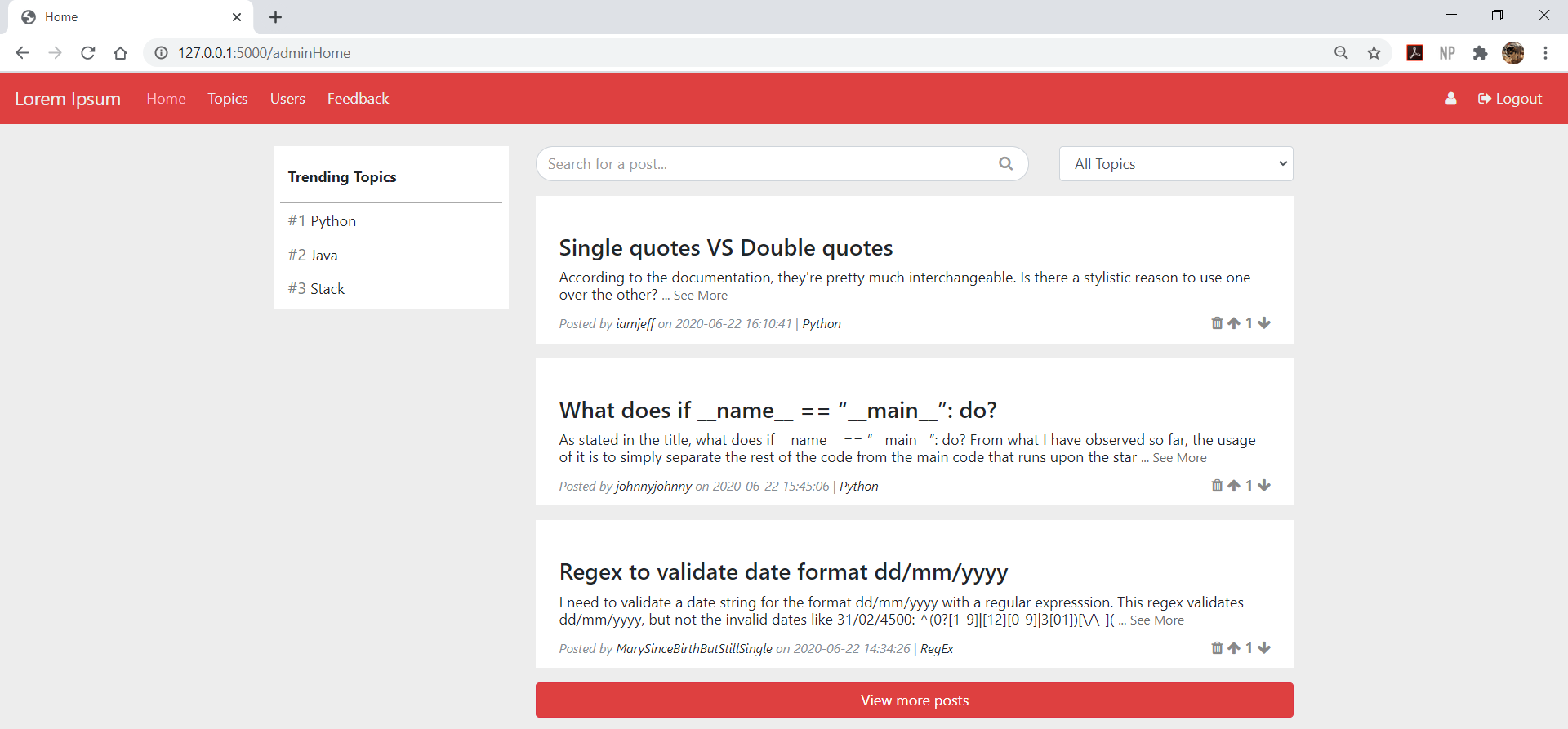
#### Image 1.1.4 WTForm Configuration of Form to create new post (secure version).



### Implementation 2: Unauthorized access to admin pages

Routes to admin pages are not protected, and hence are accessible to anyone via URL manipulation. By observing the URL naming convention of Lorem Ipsum, or by brute forcing, attackers may guess the route for admin pages. This allows them to gain access to the admin platform and perform actions above the privilege of a normal user such as deleting posts.

#### Image 1.2.1 Successful access to admin home page by unauthorized user.



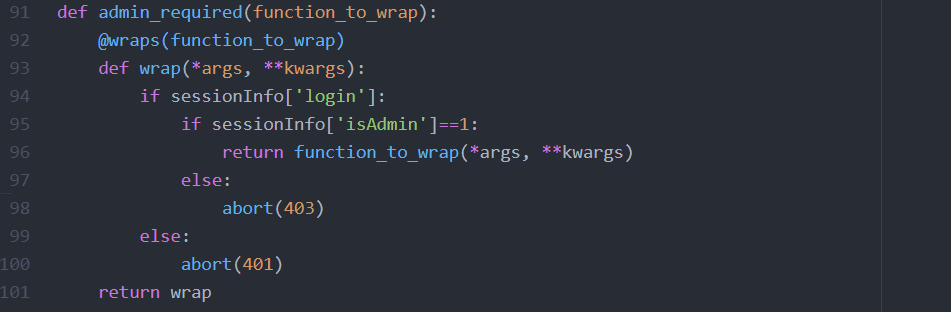
**Attacker guesses a valid route**

**Not logged in as any user**

### Solution 2: Unauthorized access to admin pages

To secure the admin pages from unauthorized access, I created an “admin\_required” decorator. The decorator checks if the user is logged in, and if so whether they are an admin. If the user is not logged in, or does not have the admin role, error 401 and 403 will be invoked respectively. The decorator is then used on the routes that are only accessible by admins.

#### Image 1.2.2 Code for “admin\_required” decorator (secure version).



#### Image 1.2.3 Snippet of some routes that use the “admin\_required” decorator (secure version).

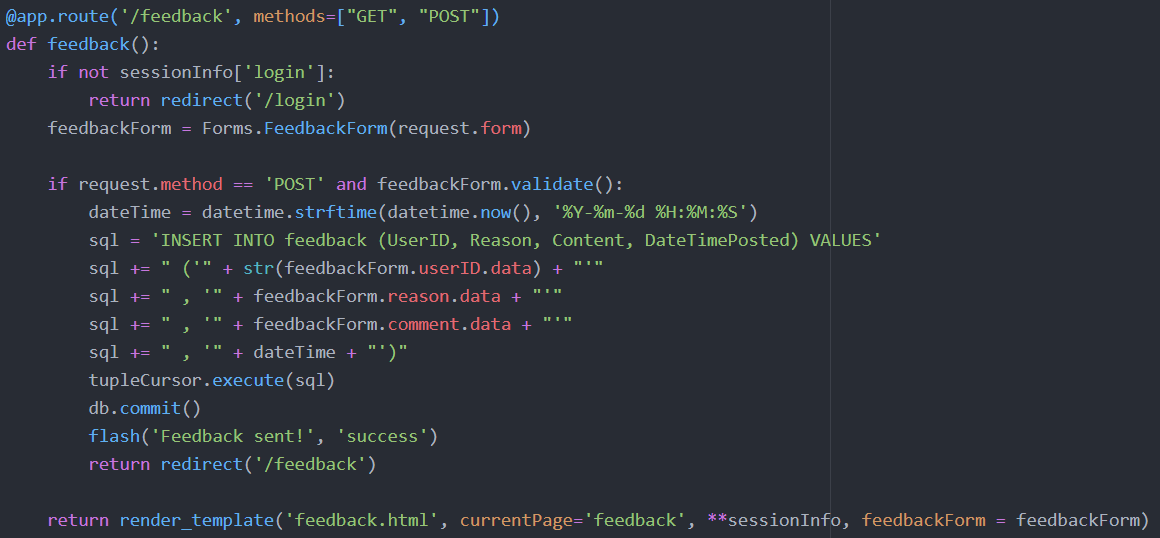


## Vulnerability 2: Injection

Injection is used by attackers to trick an application into including unintended commands in the data sent to an interpreter. For the vulnerable version of Lorem Ipsum, the raw input is concatenated to the SQL command. Therefore, allowing the SQL command inserted by the attacker to be executed.

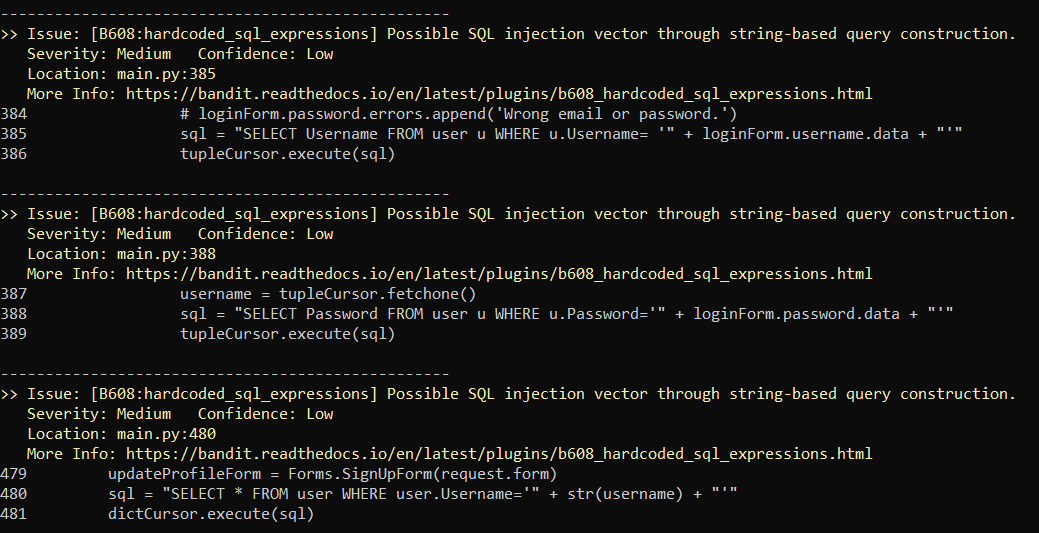
During the static and dynamic analysis of the vulnerable version of Lorem Ipsum (using Bandit and ZAP respectively), the tools were able to detect the risk of injection.

#### Image 2.0.1 Snippet of source code for feedback page (vulnerable version)

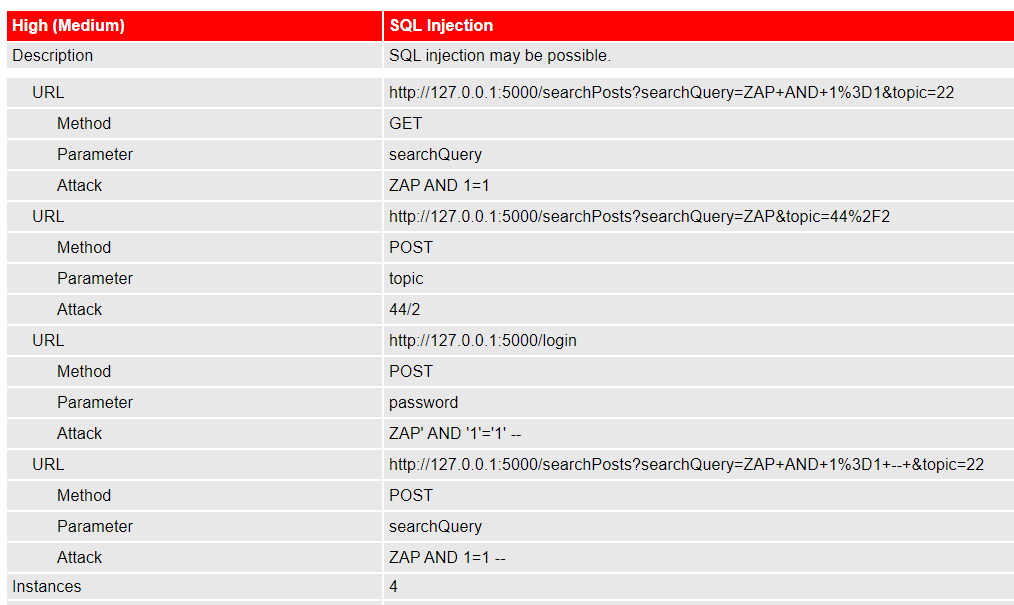


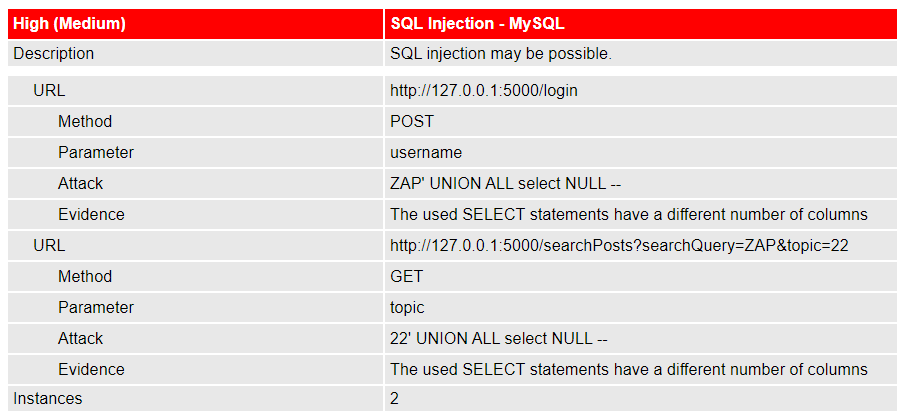
**Concatenation is used to form SQL query**

#### Image 2.0.2 Static Analysis of vulnerable version using Bandit - Report



#### Image 2.0.3 Dynamic Analysis of vulnerable version using ZAP - Report



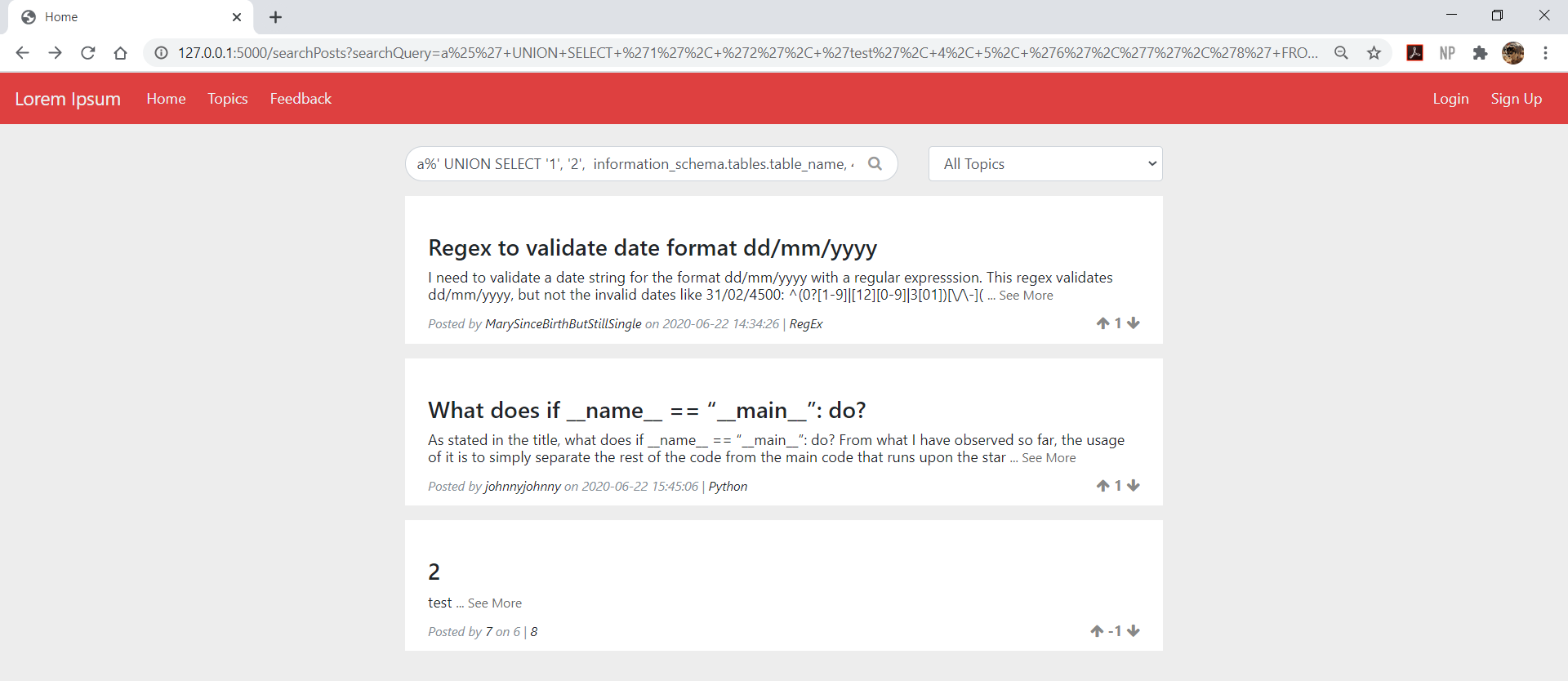


### Implementation 1: Union Select attack on Search Bar

The following SQL queries are some examples that will allow attackers to obtain database information using the Union Select attack method.

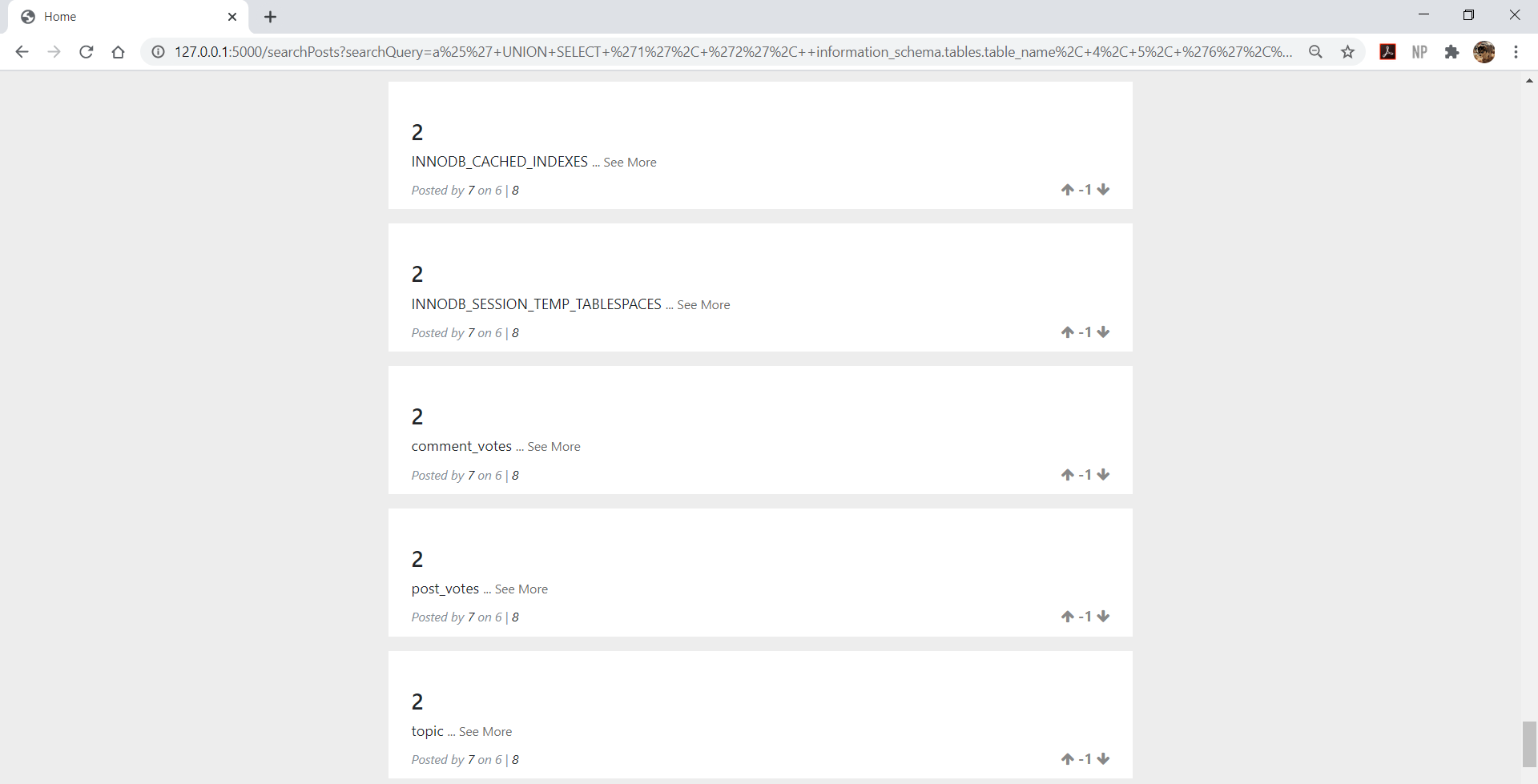
|  |  |
| --- | --- |
| **Information obtained** | **Query** |
| No. of columns to match query ([Image 2.1.1](#_Image_2.1.1_)) | a%' UNION SELECT '1', '2', 'test', 4, 5, '6','7','8' FROM information\_schema.tables # |
| List of Tables in database ([Image 2.1.2](#_Image_2.1.2_)) | a%' UNION SELECT '1', '2', information\_schema.tables.table\_name, 4, 5, '6','7','8' FROM information\_schema.tables # |
| Attributes of ‘user’ table ([Image 2.1.3](#_Image_2.1.3_)) | a%' UNION SELECT '1', '2', i.column\_name, 4, 5, '6','7','8' FROM information\_schema.columns WHERE TABLE\_NAME='user'# |
| Username, Password, Email and Privilege level of users ([Image 2.1.4](#_Image_2.1.4_)) | a%' UNION SELECT '1', username, password, 4, 5, email,'7', isAdmin FROM user# |

#### Image 2.1.1 Attacker successfully matches number of columns in SQL query.



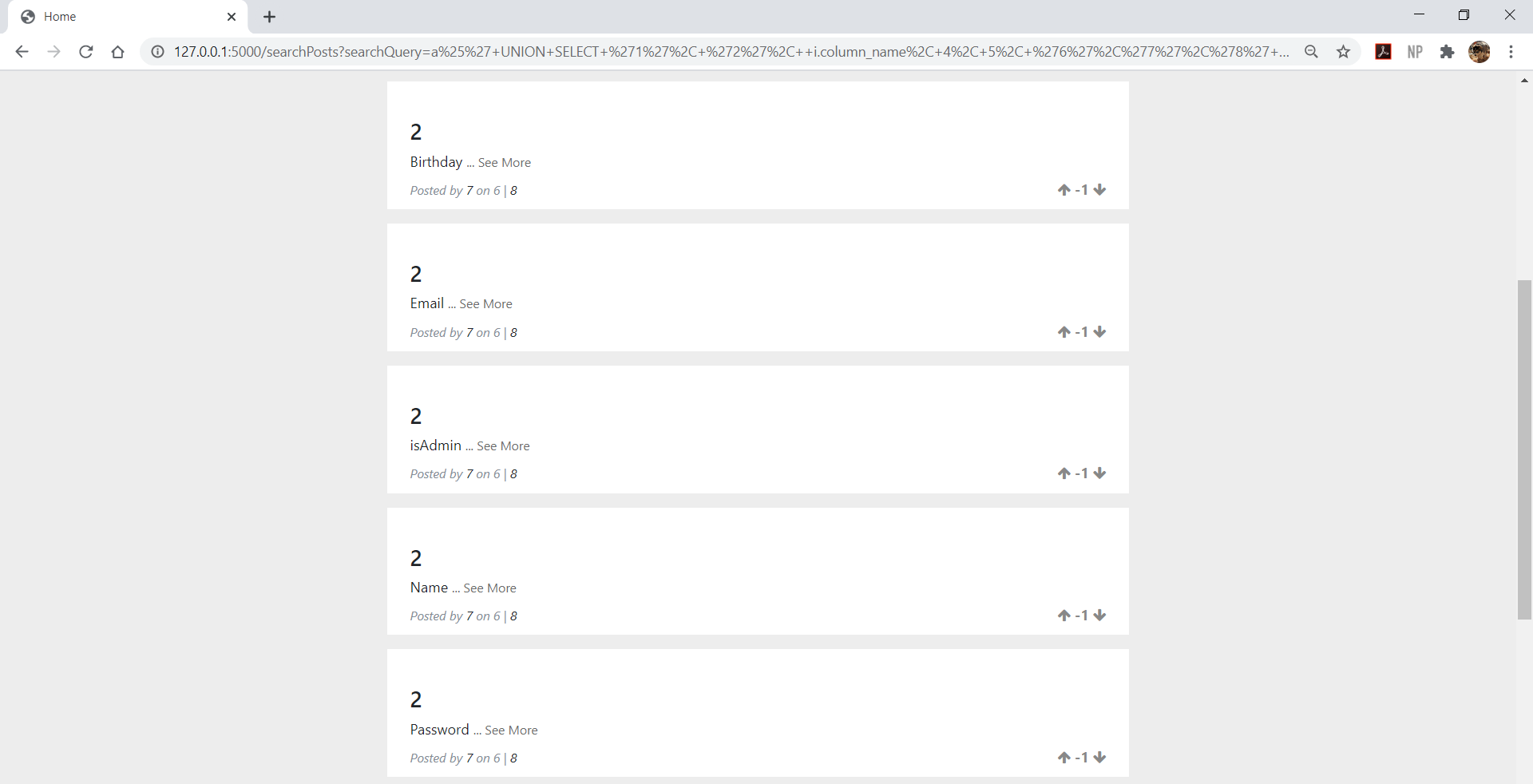
**SELECT query successfully executed**

#### Image 2.1.2 Attacker retrieve list of tables names in database.



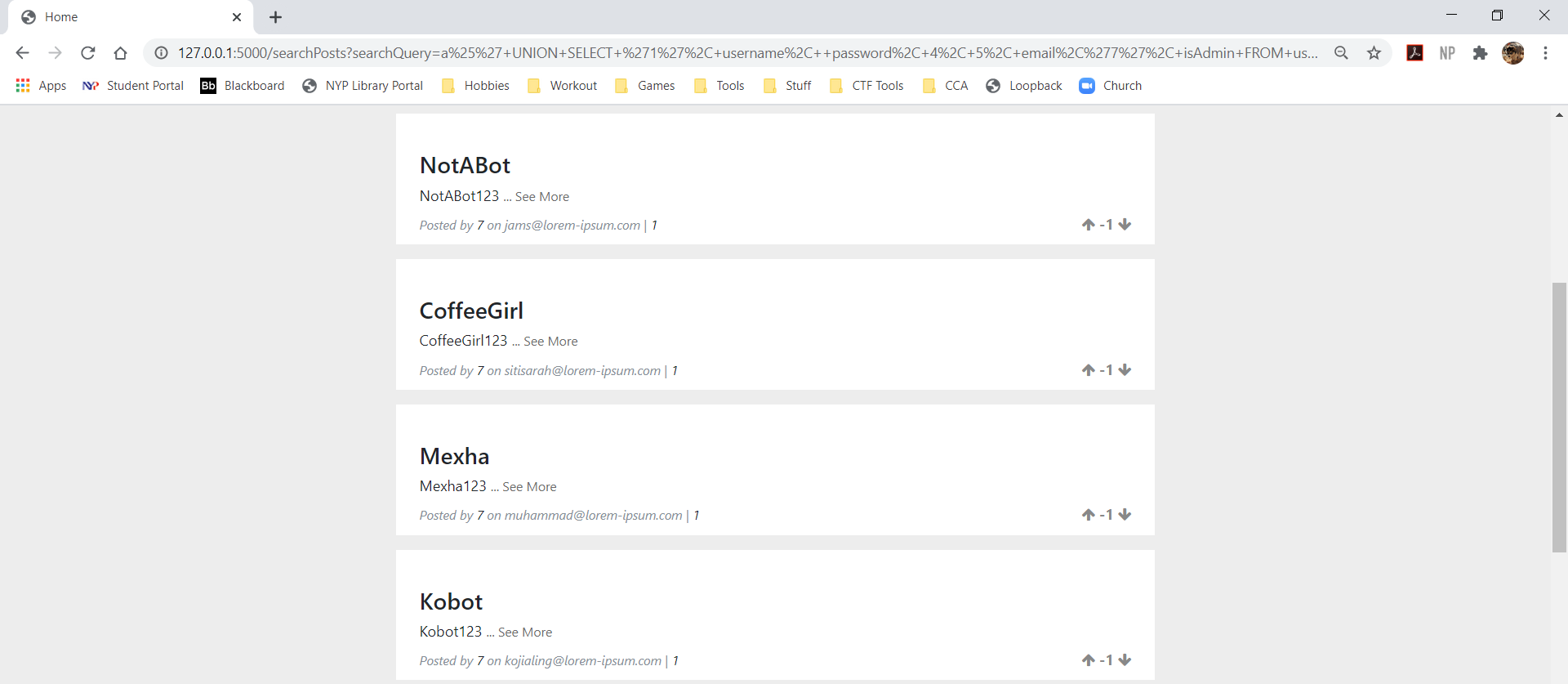
**List of tables in database**

#### Image 2.1.3 Username, Password, Email and Privilege level of users.



**Attributes of ‘user’ table**

#### Image 2.1.4 Username, Password, Email and Privilege level of users.



**Password**

**Privilege level**

**Email**

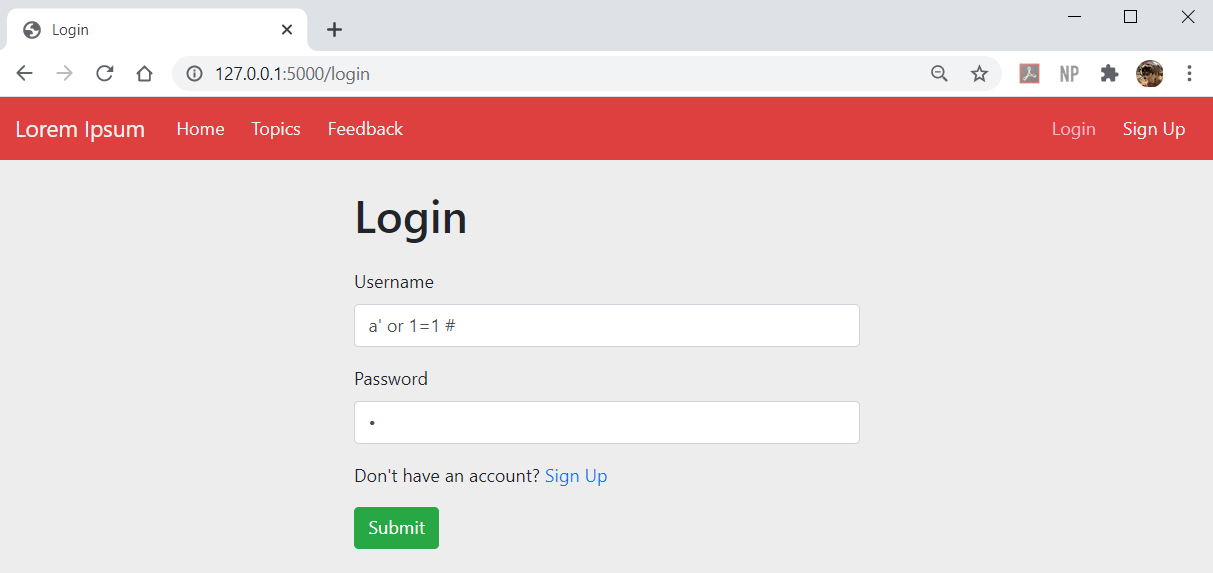
**Username**

### **Implementation 2:** Bypass login validation

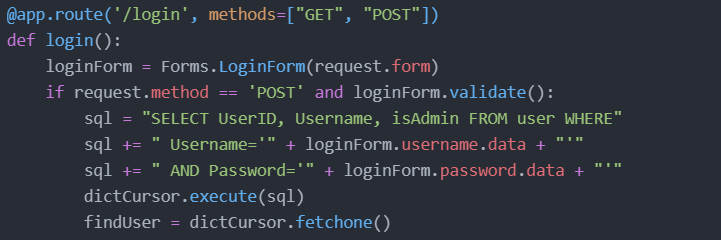
The attacker uses injection to bypass login validation and access user’s account. In the example below, the attacker injects “a’ or 1=1 #” into the username input, which effectively comments out the rest of the SQL query after the input. This causes the database to ignore the “AND password=<user password>” check that usually follows afterwards. The query with injected code will then retrieve a user account, and give the attacker access to it.

Alternatively, attackers can inject “<username>’ #” into the username input to gain access to a specific user’s account.

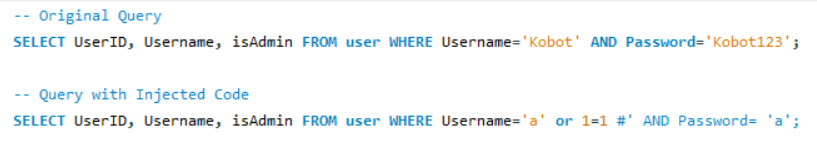
#### Image 2.2.1 Attacker injecting code to bypass the login check.

**

#### Image 2.2.2 Snippet of source code used to validate user upon login (vulnerable version).

**

#### Image 2.2.3 SQL query formed by submitting login form.

**

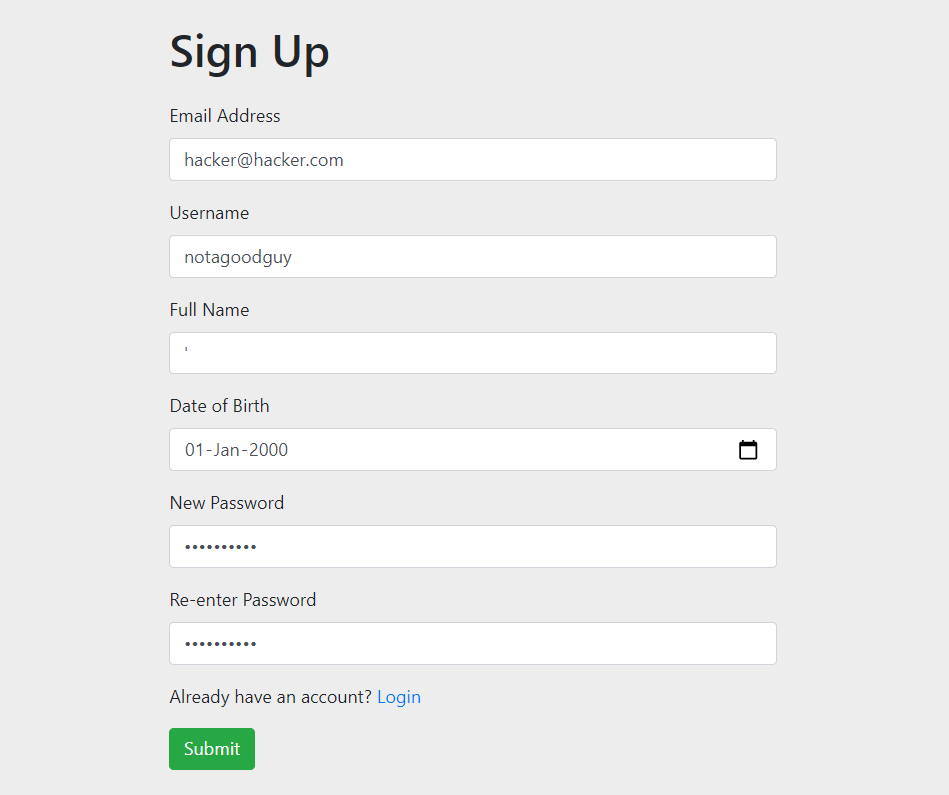
**Rest of query is commented out**

### **Implementation 3:** Create a new user with admin privileges.

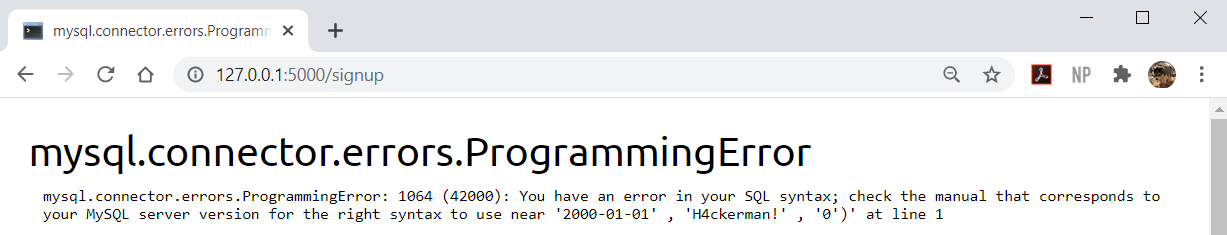
Attackers may test input fields for susceptibility to injection. Upon doing so, an error will occur and the debugger will appear. The debugger reveals information about the type of SQL server used, as well as how the rest of the SQL command is completed. This allows the attacker to guess what the remaining parameters are for and form a new injection query.

In this example, the attacker may guess that the final parameter is a True/False parameter and replace it will a value of ‘1’ instead of ‘0’. This creates a new user with admin privileges.

#### Image 2.3.1 Attacker enters a quotation mark in Full Name field to test for susceptibility to injection.

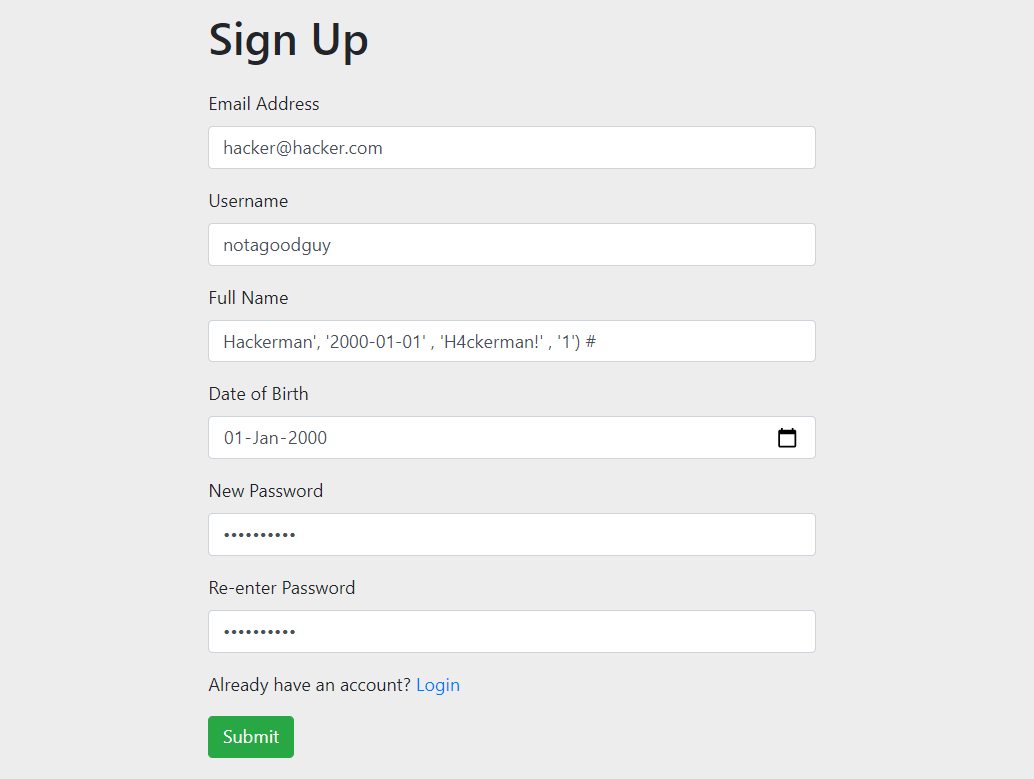


#### Image 2.3.2 Debugger reveals information about SQL query.

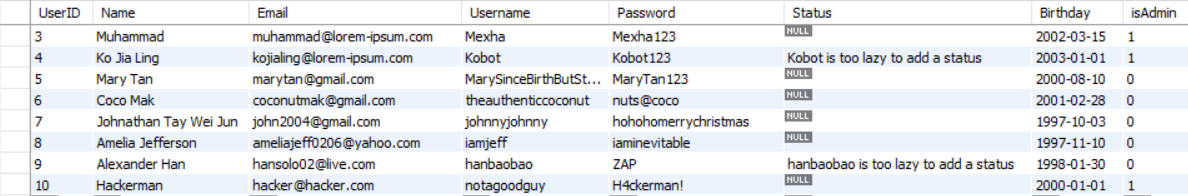


**Reveals how to complete the SQL command to create a new user**

#### Image 2.3.3 Attacker forms new injection code based on information gained.



#### Image 2.3.4 Attacker successfully creates account with admin privilege.

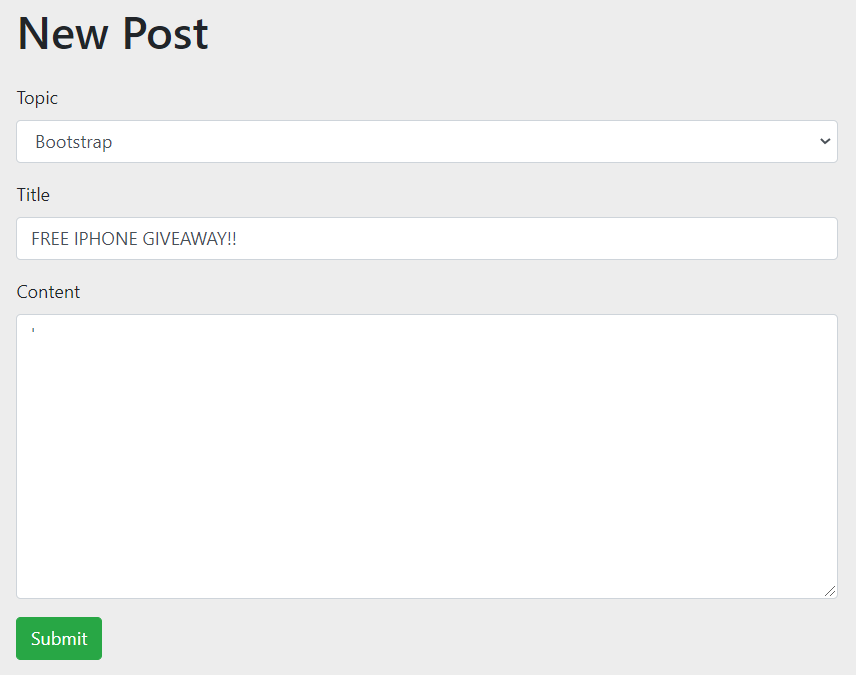


### **Implementation 4:** Create a new post with 9999 upvotes.

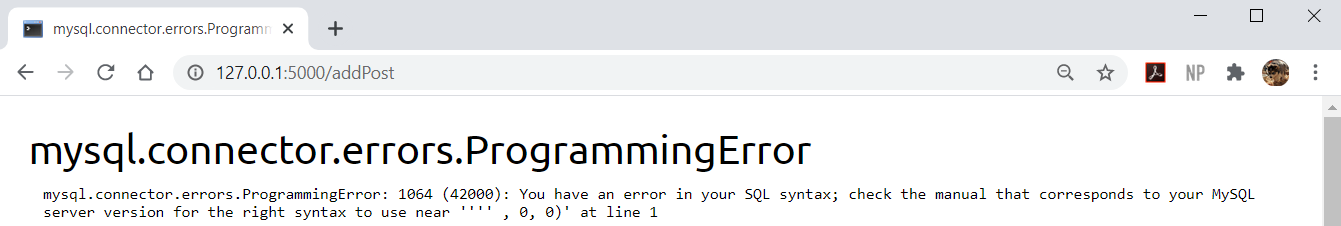
This is similar to implementation 3. Attackers may test input fields for susceptibility to injection. Upon doing so, an error will occur and the debugger will appear. The debugger reveals information about the type of SQL server used, as well as how the rest of the SQL command is completed. This allows the attacker to guess what the remaining parameters are for and form a new injection query.

In this case, the attacker may guess that the last 2 parameters are to configure the upvotes and downvotes respectively and replace the upvote value with ‘9999’. This creates a post with 9999 upvotes. This is useful for attackers that wish to create spam posts for advertising as posts with multiple upvotes will be displayed at the top of home page.

#### Image 2.4.1 Attacker enters a quotation mark in Content field to test for susceptibility to injection.

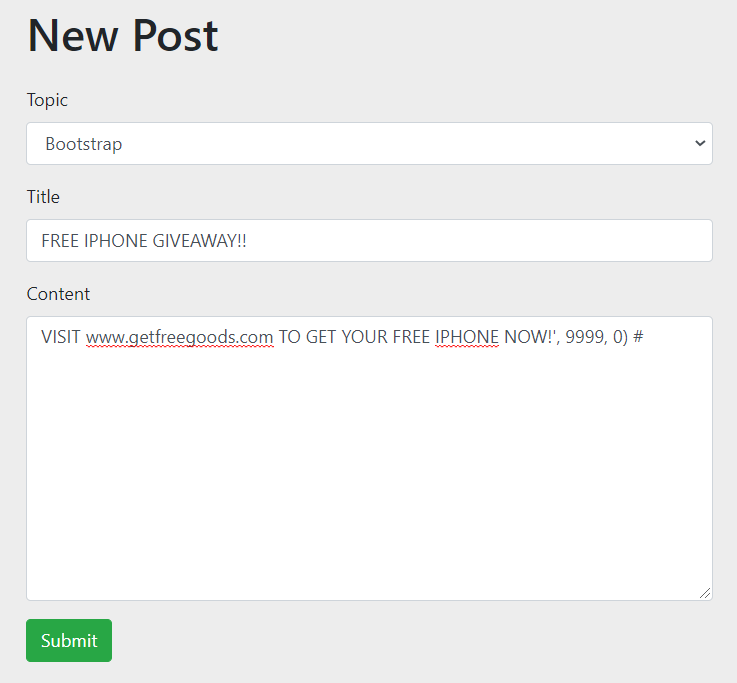


#### Image 2.4.2 Debugger reveals information about SQL query.

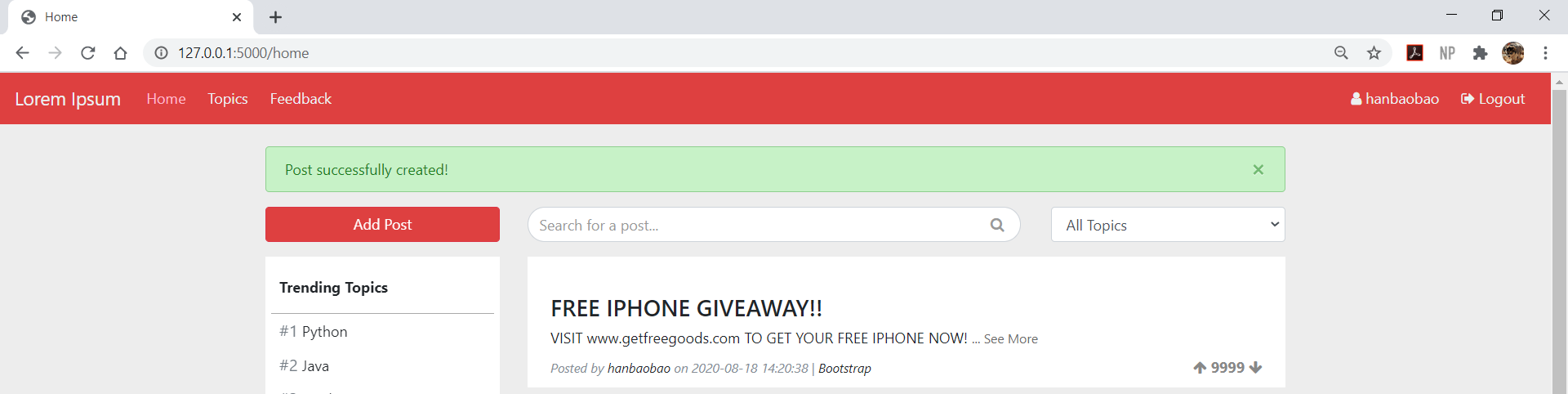


**Reveals how to complete the SQL command to create new post**

#### Image 2.4.3 Attacker forms new injection code based on information gained.



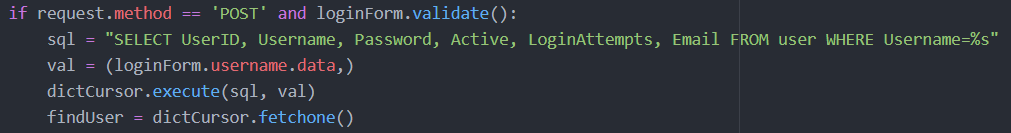
#### Image 2.4.4 Attacker successfully creates post with 9999 upvotes.



### Solution for all implementations: SQL Injection

In the secure version of Lorem Ipsum, parameterized queries are used instead of concatenating the user input to the SQL query. This means that the input is treated as a literal value instead of executable code.

#### Image 2.3.1 Snippet of source code used to validate user upon login (secure version).



## Other Vulnerabilities

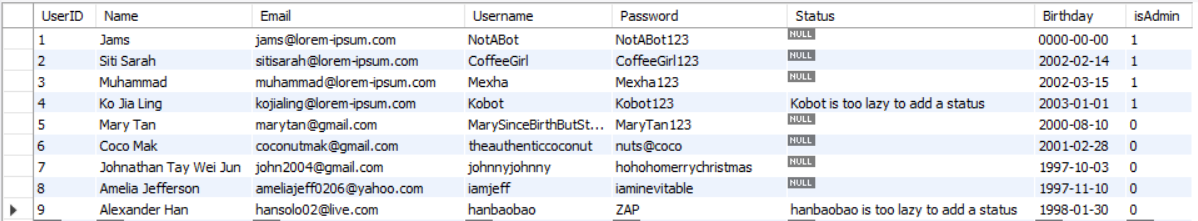
Apart from mitigating the vulnerabilities I have chosen, I implemented additional security measures to enforce Lorem Ipsum in other areas. However, as these were additional work and not part of my main chosen vulnerabilities, the extent to which these vulnerabilities are mitigated is limited. The idea was to simply make the application more secure in these aspects than the vulnerable version, not to fully prevent the vulnerability.

### Solution 1: Generating Random User IDs

In the vulnerable version of Lorem Ipsum, auto increment was used for the User ID field. This makes it easy for attackers to guess which User IDs are in use, which they can used for malicious reasons such as [forging another user’s identity](#_Implementation_1:_Pretending).

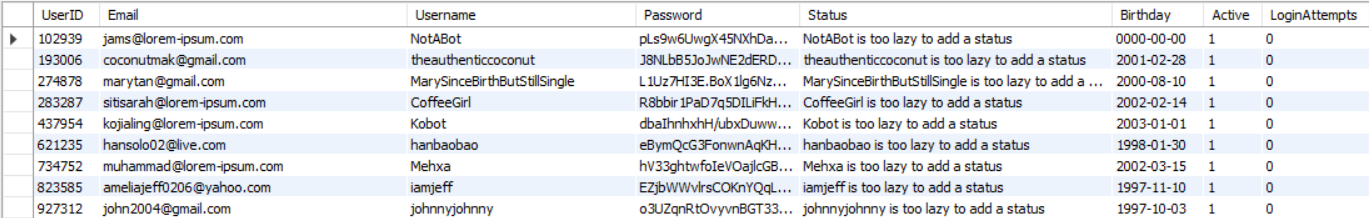
In the secure version of Lorem Ipsum, I added a trigger to the User table to generate a random 6-digit user ID. While it is still possible for attackers to brute force and successfully guess another user’s ID, the randomization will hinder their efforts in finding a valid user ID.

#### Image 3.1.1 User table of blogdb (Database for vulnerable of Lorem Ipsum)



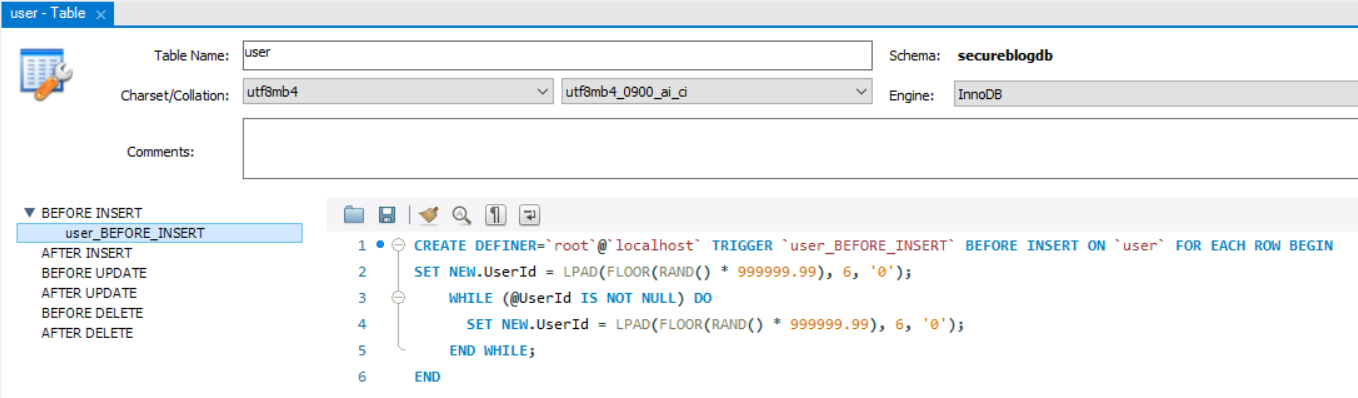
**Incremental**

#### Image 3.1.2 User table of secureblogdb (Database for secure of Lorem Ipsum)



**Randomized**

#### Image 3.1.3 Trigger for User table (secureblogdb) before inserting data.



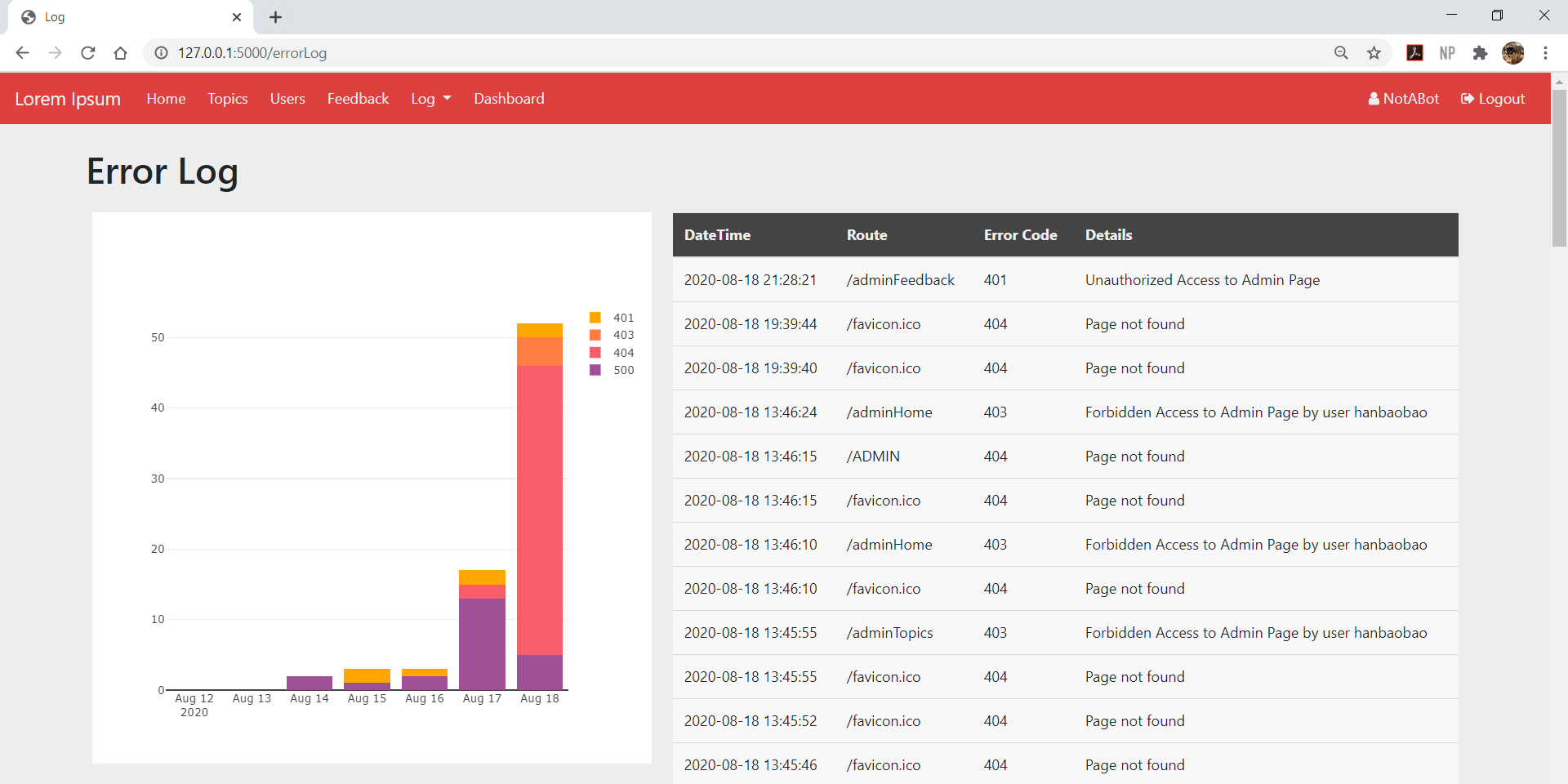
### Solution 2: Logging

In the vulnerable version of Lorem Ipsum, there was no logging of any errors or activities. Hence, I decided to create logs for errors and user activities.

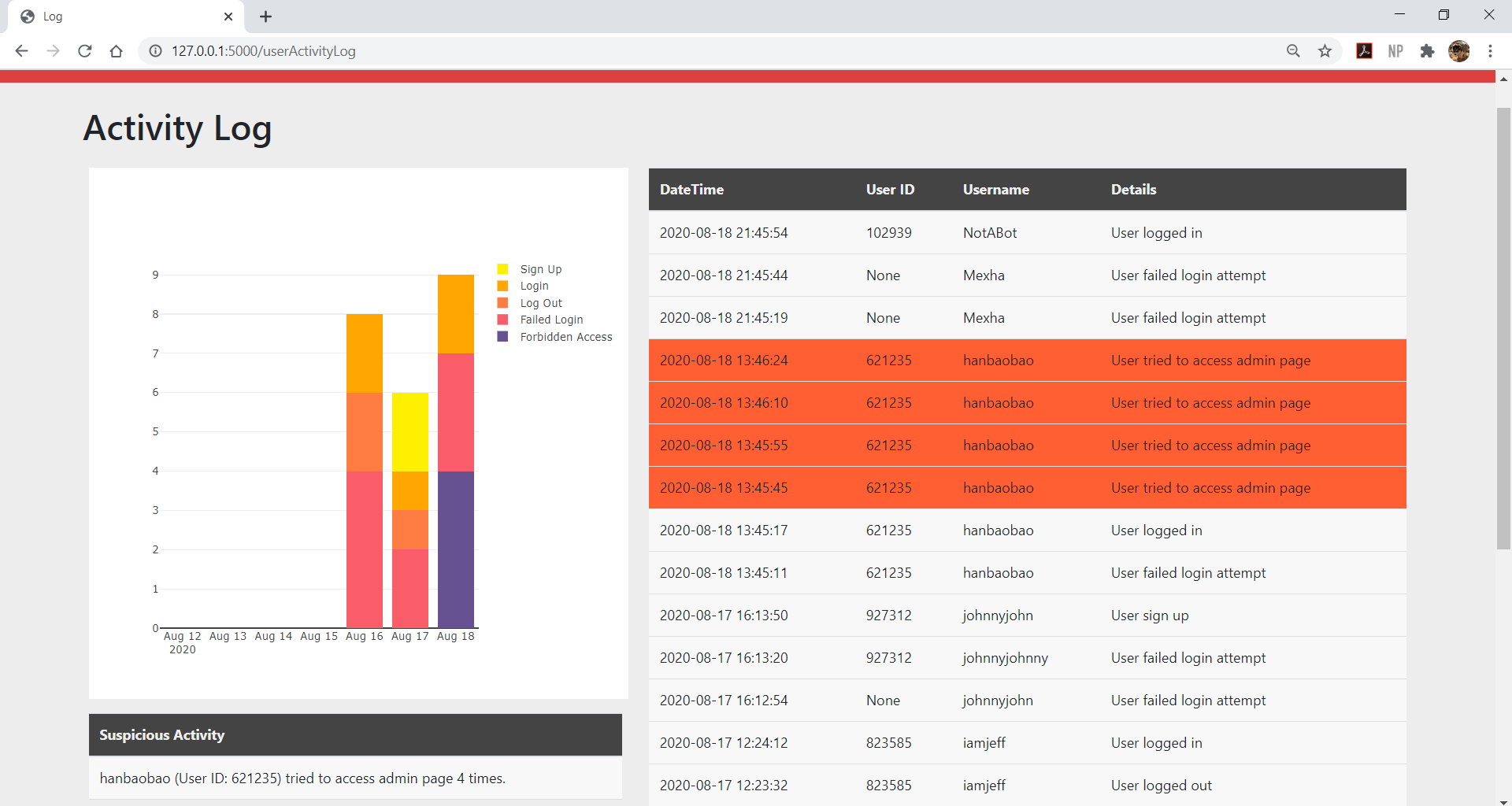
Both the error and activity log dashboard display a stacked bar chart of logged errors/activities in the last 7 days. The details of the errors/activities are displayed on a table next to the graph.

For the activity log, activities are assigned a severity level (1 being least severe). Activities with higher severity levels will be highlighted in the table. In addition, suspicious activity will be flagged and displayed under the “Suspicious Activity” table. For example, if an unauthorized user tried to access the admin page, this will be flagged as suspicious activity.

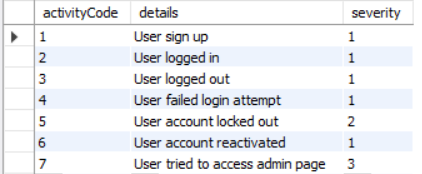
#### Image 3.2.1 Error Log Dashboard



#### Image 3.2.2 Activity Log Dashboard



#### Image 3.2.3 Activity Codes, Details and Severity Ranking (1 being least severe).



### Solution 3: Monitoring

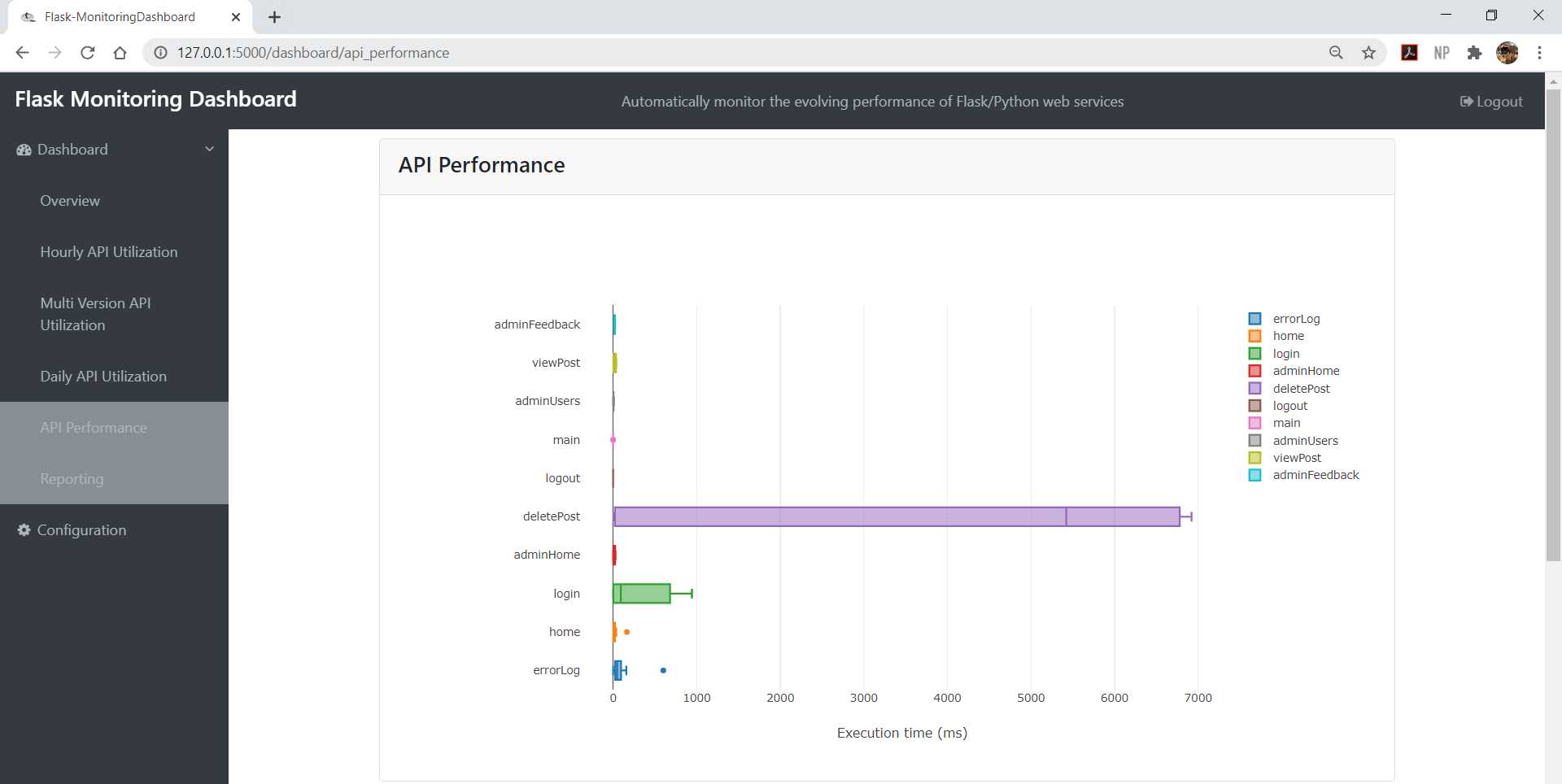
In the vulnerable version of Lorem Ipsum, there was no monitoring of requests and responses.

Hence I implemented the [Flask-Monitoring-Dashboard](https://flask-monitoringdashboard.readthedocs.io/en/v1.13.0/) library, which provides 4 main functionalities:

1. Monitor the performance and utilization.
2. Profile requests and endpoints.
3. Collect extra information about outliers.
4. Collect additional information about the Flask application.

This provides useful information that helps to determine which areas of the website needs optimization. For example, the monitoring dashboard shows that it takes about 6.8 seconds for a post to be deleted. Hence the admin may want to optimize the process of deleting posts.

#### Image 3.3.1 Flask Monitoring Dashboard – API Performance of endpoints



# 3. Other Individual Contributions

I’m actually rather proud of the fact that we managed to build a forum website from scratch in such a short period of time. It was a daunting challenge as we were aiming to have functionalities rivaling those that we made in our previous semester Application Development Project, with a shorter deadline and that’s not even considering the security portion that is the main focus of this project. We had to figure out how to use a database management system as well.

Nevertheless, we pulled through and managed to pull off what we planned. I was responsible for creating the base of the website. That includes the following functions:

* Home Page
* Search Bar – Filter posts
* Adding Posts
* Commenting
* Replying to Comments
* Upvote/Downvotes for posts and comments
* Sending Feedback
* Sign Up – Creating new account
* Log In

The most challenging part in the initial stage was learning how to use MySQL, as well as the various features that it offers. Another challenge was learning how to make use of plotly, especially since documentations on using plotly with Flask is quite limited and most of the examples are different from what I hoped to achieve.

*- End of Report -*