

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

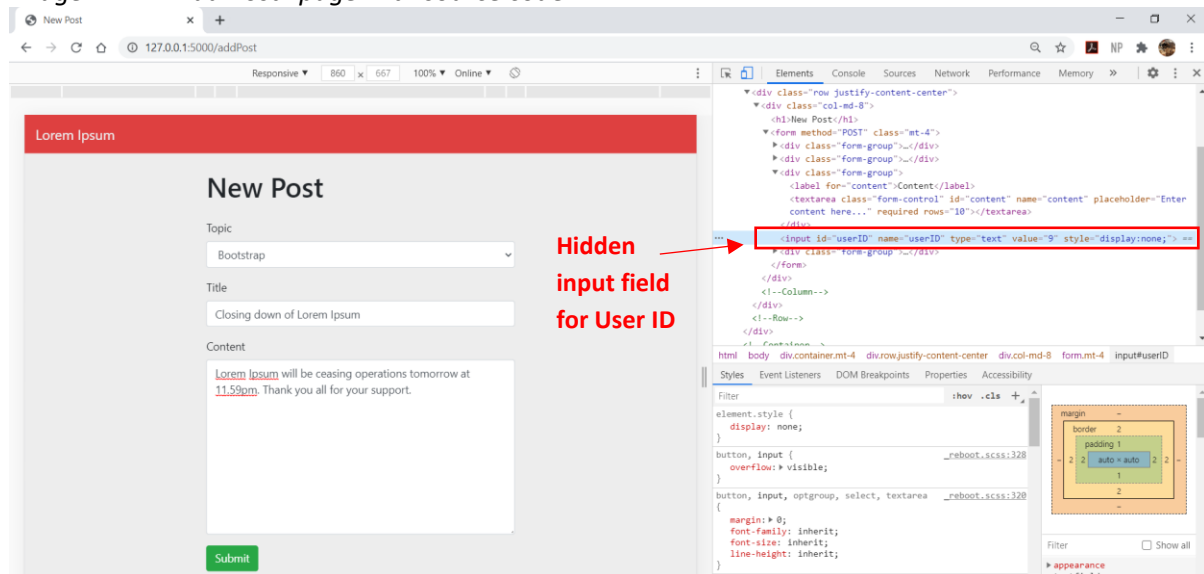
Implementation 1: Pretending to be another user

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).

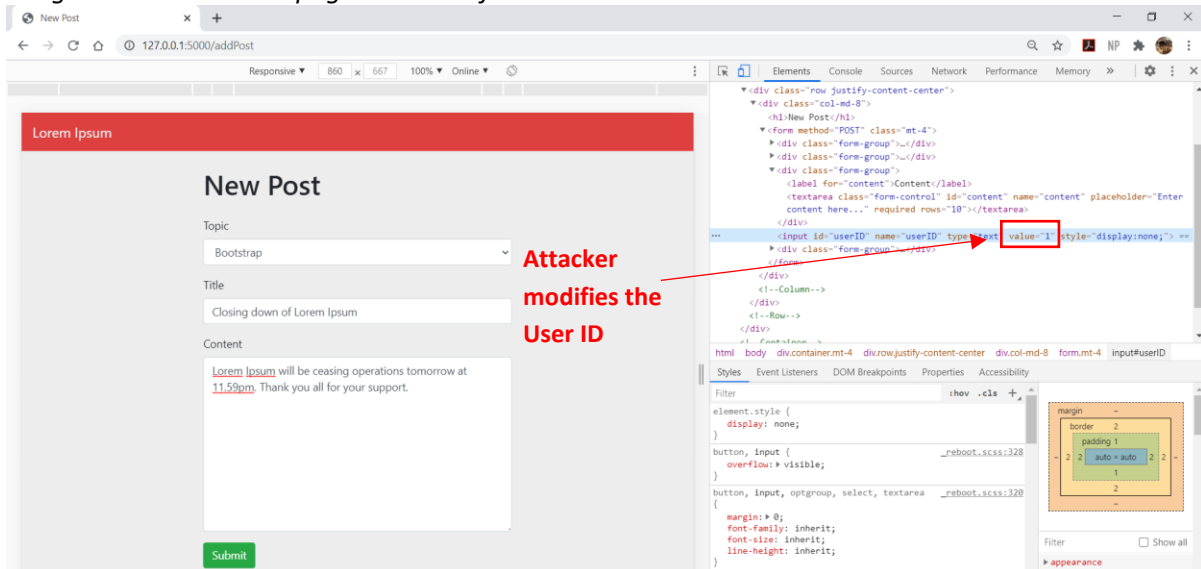
Step 1. Attacker inspects page source code to find the hidden field used to indicate the User ID.

Image 1.1.1 "Add Post" page with source code.



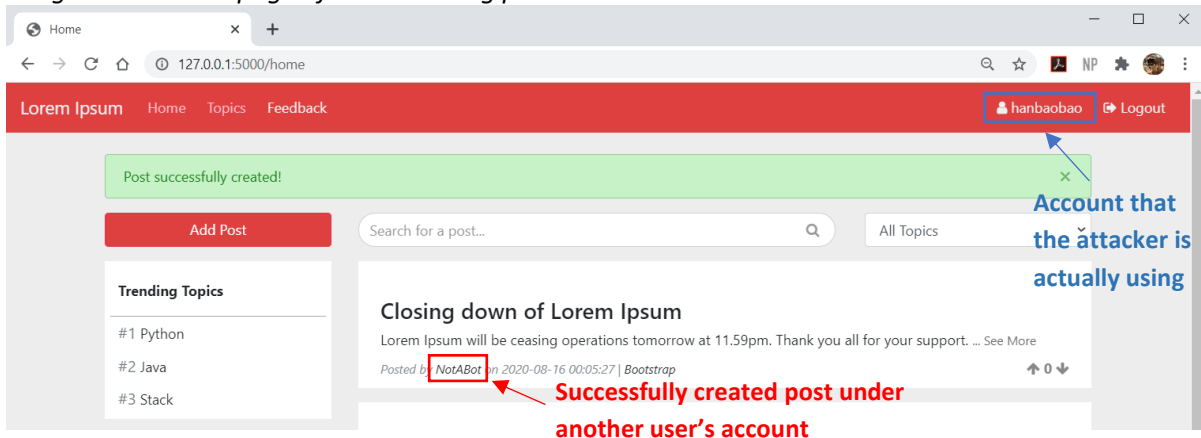
Step 2. Modify the value of hidden User ID field.

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



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.



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.

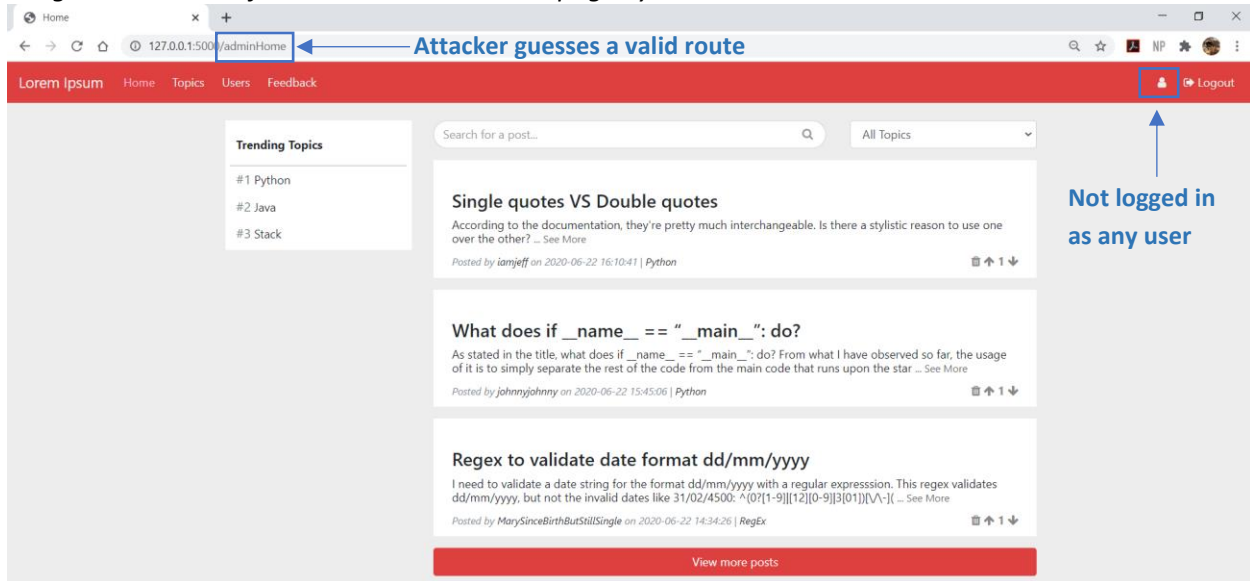
Image 1.1.4 WTForm Configuration of Form to create new post (secure version).

```
56 class PostForm(Form):
57     userID = HiddenField()
58     topic = SelectField('Topic')
59     title = StringField('Title', [validators.DataRequired()], render_kw={"placeholder": "e.g. Error Exception handling in Python"})
60     content = TextAreaField('Content', [validators.DataRequired()], render_kw={"rows": 10, "placeholder": "Enter content here..."})
```

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.



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).

```
91 def admin_required(function_to_wrap):
92     @wraps(function_to_wrap)
93     def wrap(*args, **kwargs):
94         if sessionInfo['login']:
95             if sessionInfo['isAdmin']==1:
96                 return function_to_wrap(*args, **kwargs)
97             else:
98                 abort(403)
99         else:
100             abort(401)
101     return wrap
```

Image 1.2.3 Snippet of some routes that use the “admin_required” decorator (secure version).

```
911 @app.route('/adminHome', methods=["GET", "POST"])
912 @admin_required
913 > def adminHome():=
932
933 @app.route('/adminViewPost/<int:postID>', methods=["GET", "POST"])
934 @admin_required
935 > def adminViewPost(postID):=
983
984 @app.route('/adminTopics')
985 @admin_required
986 > def adminTopics():=
993
994 @app.route('/adminIndivTopic/<topicID>', methods=["GET", "POST"])
995 @admin_required
996 > def adminIndivTopic(topicID):=
1013
1014 @app.route('/addTopic', methods=["GET", "POST"])
1015 @admin_required
1016 > def addTopic():=
```

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)

```
@app.route('/feedback', methods=["GET", "POST"])
def feedback():
    if not sessionInfo['login']:
        return redirect('/login')
    feedbackForm = Forms.FeedbackForm(request.form)

    if request.method == 'POST' and feedbackForm.validate():
        dateTime = datetime.strftime(datetime.now(), '%Y-%m-%d %H:%M:%S')
        sql = 'INSERT INTO feedback (UserID, Reason, Content, DateTimePosted) VALUES'
        sql += " ('" + str(feedbackForm.userID.data) + '"
        sql += " , '" + feedbackForm.reason.data + '"
        sql += " , '" + feedbackForm.comment.data + '"
        sql += " , '" + dateTime + '"")"
        tupleCursor.execute(sql)
        db.commit()
        flash('Feedback sent!', 'success')
        return redirect('/feedback')

    return render_template('feedback.html', currentPage='feedback', **sessionInfo, feedbackForm = feedbackForm)
```

Concatenation is used to form SQL query

Image 2.0.2 Static Analysis of vulnerable version using Bandit - Report

```
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction.
Severity: Medium Confidence: Low
Location: main.py:385
More Info: https://bandit.readthedocs.io/en/latest/plugins/b608_hardcoded_sql_expressions.html
384         # loginForm.password.errors.append('Wrong email or password.')
385         sql = "SELECT Username FROM user u WHERE u.Username= '" + loginForm.username.data + '"
386         tupleCursor.execute(sql)

>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction.
Severity: Medium Confidence: Low
Location: main.py:388
More Info: https://bandit.readthedocs.io/en/latest/plugins/b608_hardcoded_sql_expressions.html
387         username = tupleCursor.fetchone()
388         sql = "SELECT Password FROM user u WHERE u.Password='" + loginForm.password.data + '"
389         tupleCursor.execute(sql)

>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction.
Severity: Medium Confidence: Low
Location: main.py:480
More Info: https://bandit.readthedocs.io/en/latest/plugins/b608_hardcoded_sql_expressions.html
479         updateProfileForm = Forms.SignUpForm(request.form)
480         sql = "SELECT * FROM user WHERE user.Username='" + str(username) + '"
481         dictCursor.execute(sql)
```

Image 2.0.3 Dynamic Analysis of vulnerable version using ZAP - Report

High (Medium)	SQL Injection
Description	SQL injection may be possible.
URL	http://127.0.0.1:5000/searchPosts?searchQuery=ZAP+AND+1%3D1&topic=22
Method	GET
Parameter	searchQuery
Attack	ZAP AND 1=1
URL	http://127.0.0.1:5000/searchPosts?searchQuery=ZAP&topic=44%2F2
Method	POST
Parameter	topic
Attack	44/2
URL	http://127.0.0.1:5000/login
Method	POST
Parameter	password
Attack	ZAP' AND '1'='1' --
URL	http://127.0.0.1:5000/searchPosts?searchQuery=ZAP+AND+1%3D1+--+&topic=22
Method	POST
Parameter	searchQuery
Attack	ZAP AND 1=1 --
Instances	4

High (Medium)	SQL Injection - MySQL
Description	SQL injection may be possible.
URL	http://127.0.0.1:5000/login
Method	POST
Parameter	username
Attack	ZAP' UNION ALL select NULL --
Evidence	The used SELECT statements have a different number of columns
URL	http://127.0.0.1:5000/searchPosts?searchQuery=ZAP&topic=22
Method	GET
Parameter	topic
Attack	22' UNION ALL select NULL --
Evidence	The used SELECT statements have a different number of columns
Instances	2

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)	<code>a%' UNION SELECT '1', '2', 'test', 4, 5, '6','7','8' FROM information_schema.tables #</code>
List of Tables in database (Image 2.1.2)	<code>a%' UNION SELECT '1', '2', information_schema.tables.table_name, 4, 5, '6','7','8' FROM information_schema.tables #</code>
Attributes of 'user' table (Image 2.1.3)	<code>a%' UNION SELECT '1', '2', i.column_name, 4, 5, '6','7','8' FROM information_schema.columns WHERE TABLE_NAME='user' #</code>
Username, Password, Email and Privilege level of users (Image 2.1.4)	<code>a%' UNION SELECT '1', username, password, 4, 5, email,'7', isAdmin FROM user #</code>

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

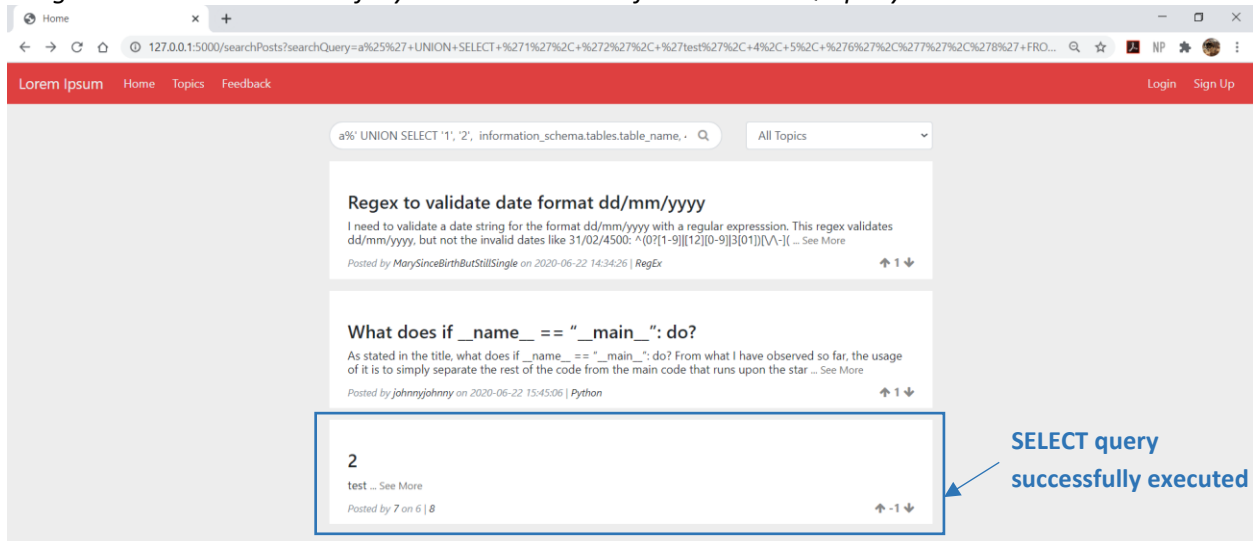


Image 2.1.2 Attacker retrieve list of tables names in database.

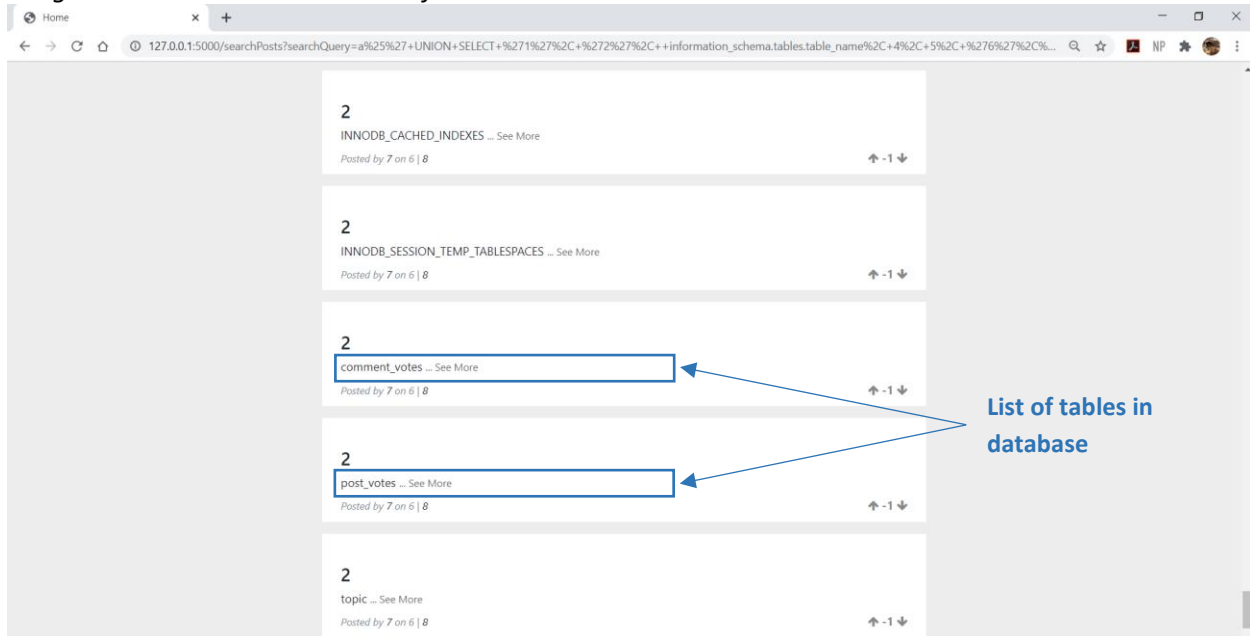


Image 2.1.3 Username, Password, Email and Privilege level of users.

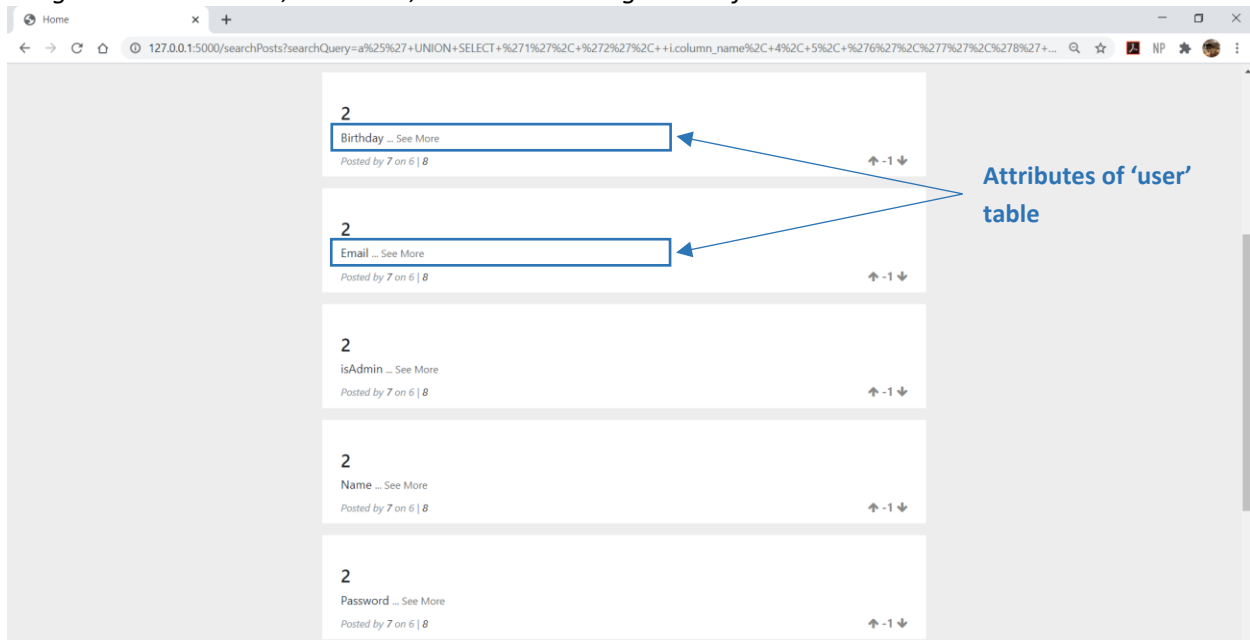
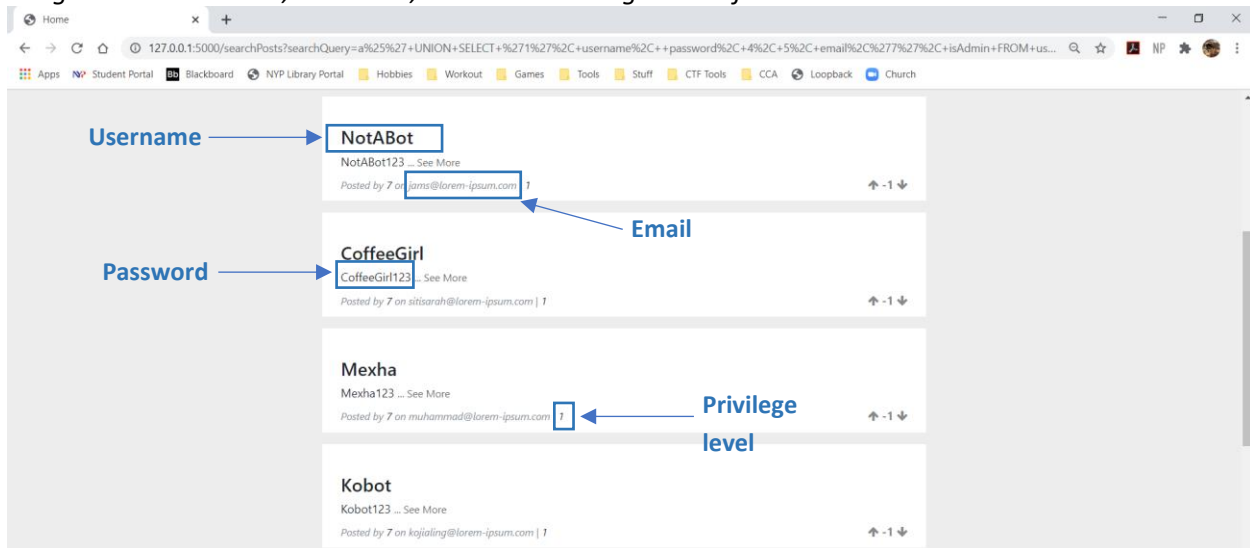


Image 2.1.4 Username, Password, Email and Privilege level of users.



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.

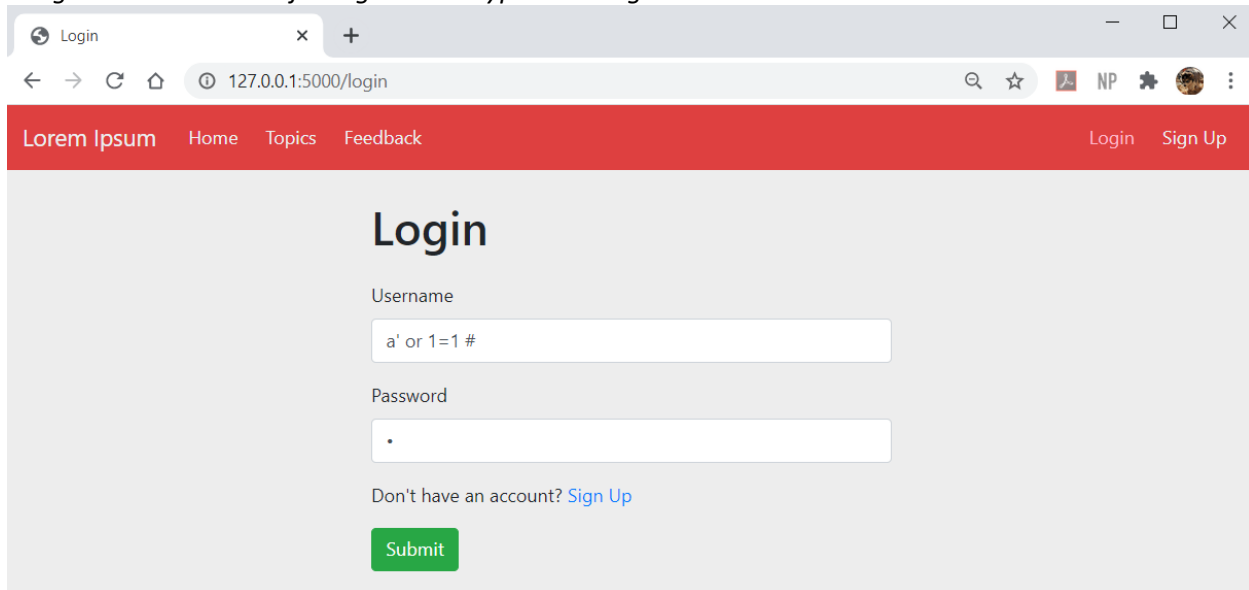


Image 2.2.2 Snippet of source code used to validate user upon login (vulnerable version).

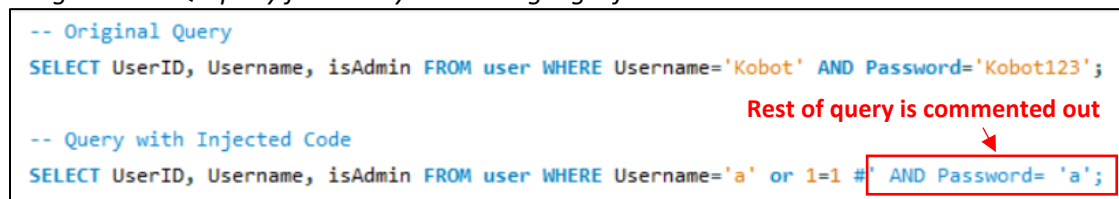
```
@app.route('/login', methods=["GET", "POST"])
def login():
    loginForm = Forms.LoginForm(request.form)
    if request.method == 'POST' and loginForm.validate():
        sql = "SELECT UserID, Username, isAdmin FROM user WHERE"
        sql += " Username='" + loginForm.username.data + "'"
        sql += " AND Password='" + loginForm.password.data + "'"
        dictCursor.execute(sql)
        findUser = dictCursor.fetchone()
```

Image 2.2.3 SQL query formed by submitting login form.

```
-- Original Query
SELECT UserID, Username, isAdmin FROM user WHERE Username='Kobot' AND Password='Kobot123';

-- Query with Injected Code
SELECT UserID, Username, isAdmin FROM user WHERE Username='a' or 1=1 # ' AND Password= 'a';
```

Rest of query is commented out

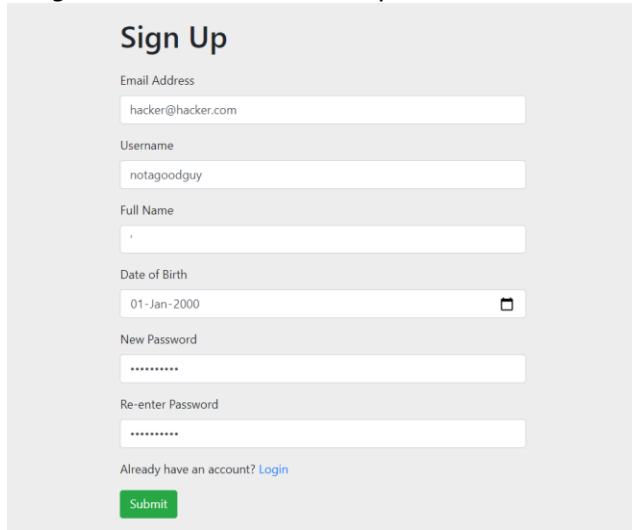


Implementation 3: Create a new user with admin privileges.

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 with 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.



The image shows a 'Sign Up' form with the following fields and values:

- Email Address: hacker@hacker.com
- Username: notagoodguy
- Full Name: '
- Date of Birth: 01-Jan-2000
- New Password: (masked with asterisks)
- Re-enter Password: (masked with asterisks)

At the bottom, there is a link 'Already have an account? Login' and a green 'Submit' button.

Image 2.3.2 Debugger reveals information about SQL query.

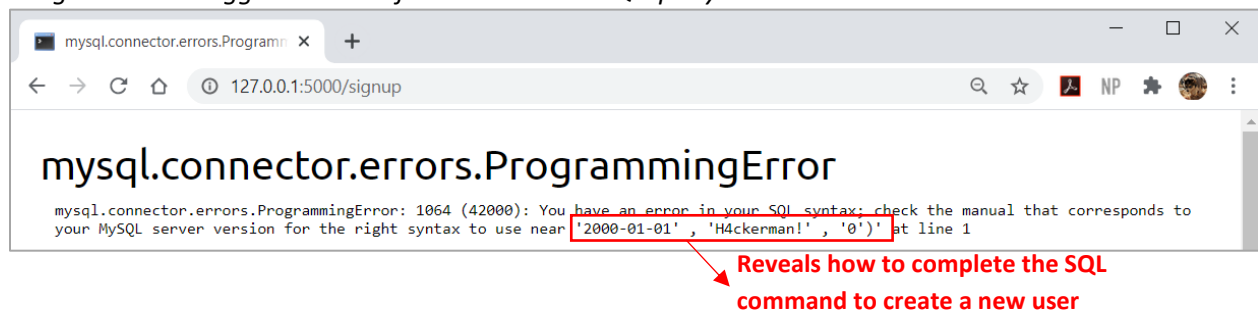


Image 2.3.3 Attacker forms new injection code based on information gained.

Sign Up

Email Address

Username

Full Name

Date of Birth

New Password

Re-enter Password

Already have an account? [Login](#)

Image 2.3.4 Attacker successfully creates account with admin privilege.

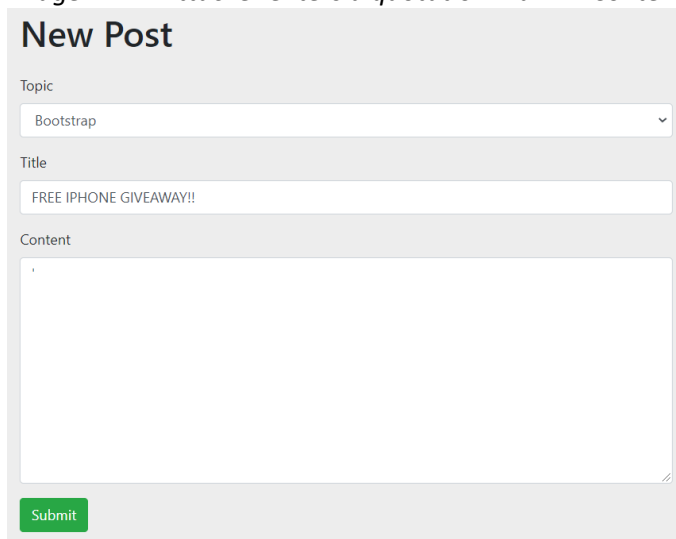
	UserID	Name	Email	Username	Password	Status	Birthday	isAdmin
	3	Muhammad	muhammad@lorem-ipsum.com	Mexha	Mexha123	HULL	2002-03-15	1
	4	Ko Jia Ling	kojialing@lorem-ipsum.com	Kobot	Kobot123	Kobot is too lazy to add a status	2003-01-01	1
	5	Mary Tan	marytan@gmail.com	MarySinceBirthButSt...	MaryTan123	HULL	2000-08-10	0
	6	Coco Mak	coconutmak@gmail.com	theauthenticcoconut	nuts@coco	HULL	2001-02-28	0
	7	Johnathan Tay Wei Jun	john2004@gmail.com	johnnyjohnny	hohohomerrychristmas	HULL	1997-10-03	0
	8	Amelia Jefferson	ameliajeff0206@yahoo.com	iamjeff	iaminevitable	HULL	1997-11-10	0
	9	Alexander Han	hansolo02@live.com	hanbaobao	ZAP	hanbaobao is too lazy to add a status	1998-01-30	0
	10	Hackerman	hacker@hacker.com	notagoodguy	H4ckerman!	HULL	2000-01-01	1

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.



New Post

Topic
Bootstrap

Title
FREE IPHONE GIVEAWAY!!

Content
'

Submit

Image 2.4.2 Debugger reveals information about SQL query.

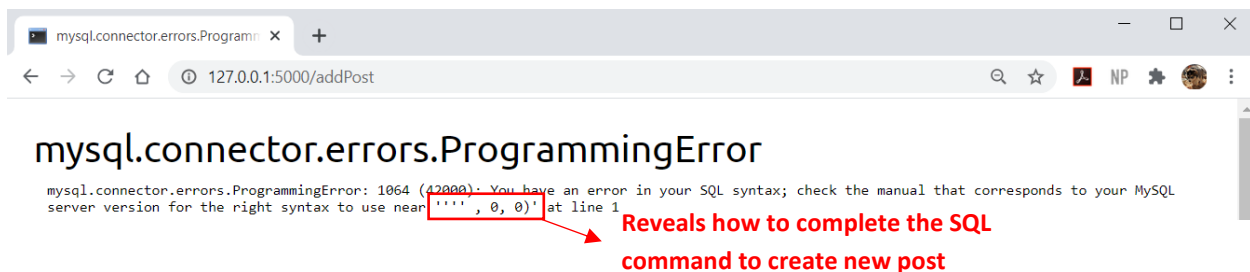


Image 2.4.3 Attacker forms new injection code based on information gained.

New Post

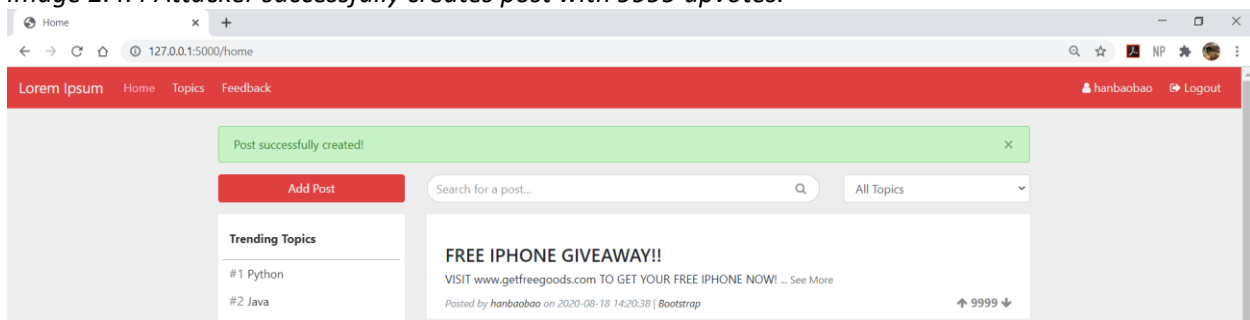
Topic
Bootstrap

Title
FREE IPHONE GIVEAWAY!!

Content
VISIT www.getfreegoods.com TO GET YOUR FREE IPHONE NOW!', 9999, 0) #

Submit

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).

```
if request.method == 'POST' and loginForm.validate():
    sql = "SELECT UserID, Username, Password, Active, LoginAttempts, Email FROM user WHERE Username=%s"
    val = (loginForm.username.data,)
    dictCursor.execute(sql, val)
    findUser = dictCursor.fetchone()
```


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 use for malicious reasons such as [forging another user's identity](#).

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)

UserID	Name	Email	Username	Password	Status	Birthday	isAdmin
1	Jams	jams@lorem-ipsum.com	NotABot	NotABot123	NULL	0000-00-00	1
2	Siti Sarah	sitisarah@lorem-ipsum.com	CoffeeGirl	CoffeeGirl123	NULL	2002-02-14	1
3	Muhammad	muhammad@lorem-ipsum.com	Mexha	Mexha123	NULL	2002-03-15	1
4	Ko Jia Ling	kojialing@lorem-ipsum.com	Kobot	Kobot123	Kobot is too lazy to add a status	2003-01-01	1
5	Mary Tan	marytan@gmail.com	MarySinceBirthButSt...	MaryTan123	NULL	2000-08-10	0
6	Coco Mak	coconutmak@gmail.com	theauthenticcoconut	nuts@coco	NULL	2001-02-28	0
7	Johnathan Tay Wei Jun	john2004@gmail.com	johnnyjohnny	hohohomerrychristmas	NULL	1997-10-03	0
8	Amelia Jefferson	ameliajeff0206@yahoo.com	iamjeff	iaminevitable	NULL	1997-11-10	0

Incremental

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

UserID	Email	Username	Password	Status	Birthday	Active	LoginAttempts
102939	jams@lorem-ipsum.com	NotABot	pls9w6UwgX45N0hDa...	NotABot is too lazy to add a status	0000-00-00	1	0
193006	coconutmak@gmail.com	theauthenticcoconut	J8NlB5JoJwNE2dERD...	theauthenticcoconut is too lazy to add a status	2001-02-28	1	0
274878	marytan@gmail.com	MarySinceBirthButStillSingle	L1Uz7HI3E.Box1lg6Nz...	MarySinceBirthButStillSingle is too lazy to add a ...	2000-08-10	1	0
283287	sitisarah@lorem-ipsum.com	CoffeeGirl	R8bbir1PaD7qSDlFIdH...	CoffeeGirl is too lazy to add a status	2002-02-14	1	0
437954	kojialing@lorem-ipsum.com	Kobot	dbalIhnhxHJubxDuww...	Kobot is too lazy to add a status	2003-01-01	1	0
621235	hansolo02@live.com	hanbaobao	eBymQcG3FonwnAqKH...	hanbaobao is too lazy to add a status	1998-01-30	1	0
734752	muhammad@lorem-ipsum.com	Mehxa	hV33ghtwfoIeVOajcGB...	Mehxa is too lazy to add a status	2002-03-15	1	0
823585	ameliajeff0206@yahoo.com	iamjeff	EZjbVWVlrsCOKnYQl...	iamjeff is too lazy to add a status	1997-11-10	1	0

Randomized

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

user - Table

Table Name: user

Schema: secureblogdb

Charset/Collation: utf8mb4 utf8mb4_0900_ai_ci

Engine: InnoDB

Comments:

▼ BEFORE INSERT

user_BEFORE_INSERT

AFTER INSERT

BEFORE UPDATE

AFTER UPDATE

BEFORE DELETE

AFTER DELETE

```
1 CREATE DEFINER='root'@'localhost' TRIGGER `user_BEFORE_INSERT` BEFORE INSERT ON `user` FOR EACH ROW BEGIN
2 SET NEW.UserId = LPAD(FLOOR(RAND() * 999999.99), 6, '0');
3 WHILE (@UserId IS NOT NULL) DO
4 SET NEW.UserId = LPAD(FLOOR(RAND() * 999999.99), 6, '0');
5 END WHILE;
6 END
```

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

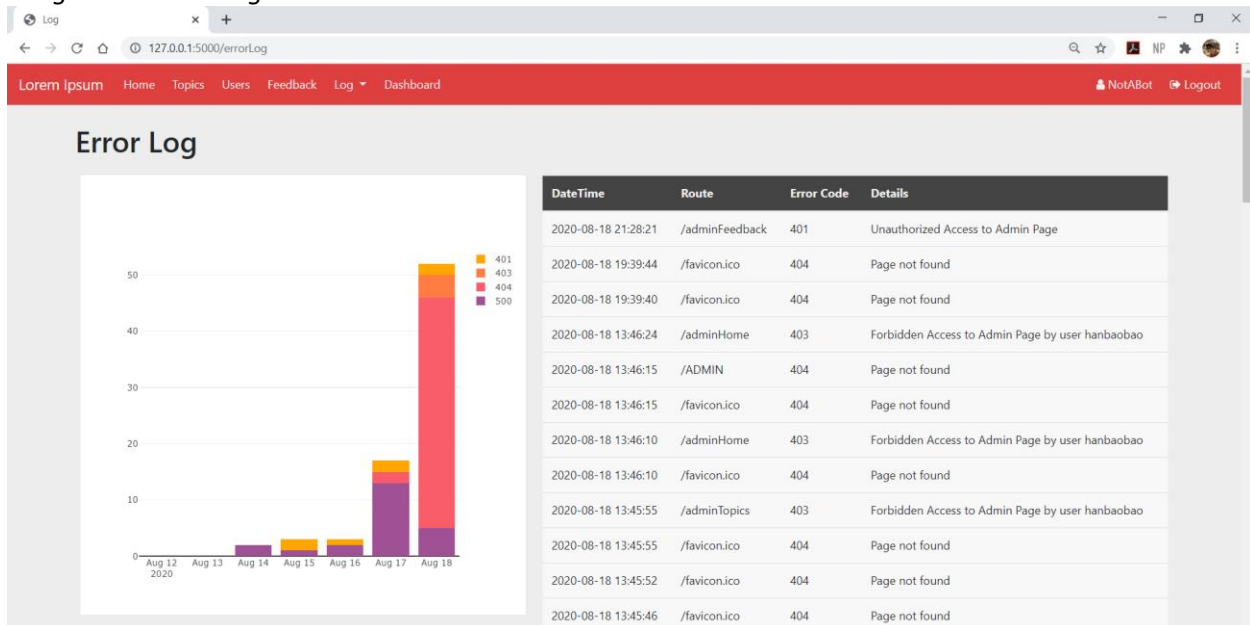


Image 3.2.2 Activity Log Dashboard

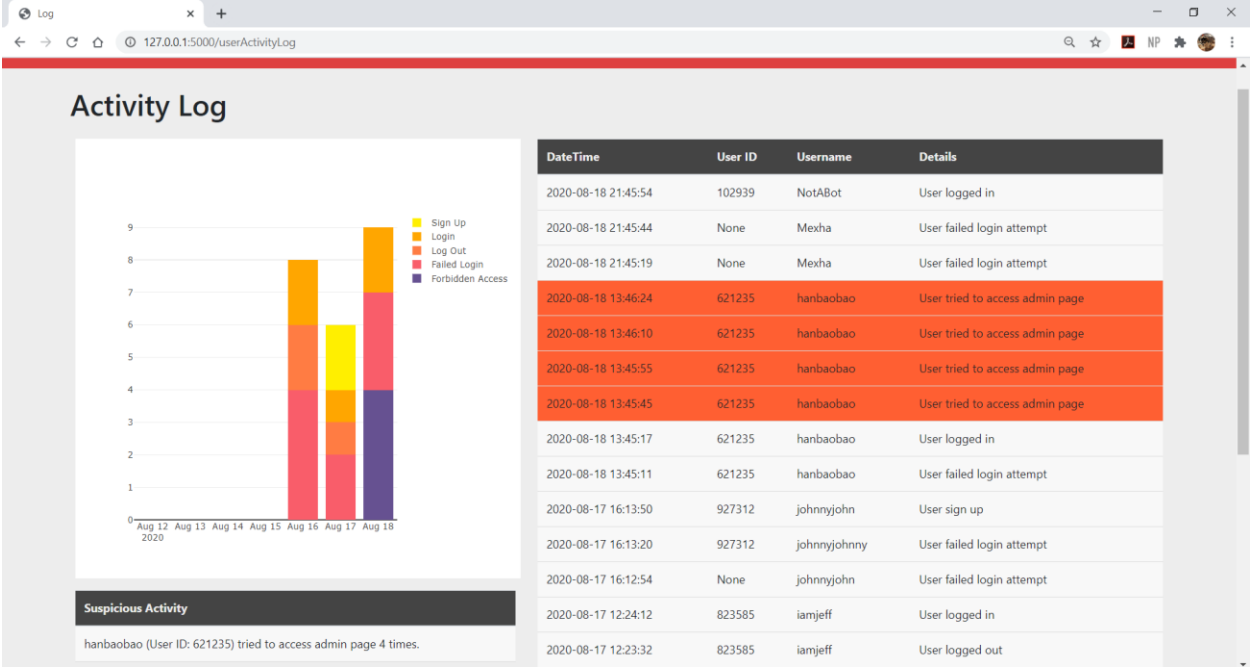


Image 3.2.3 Activity Codes, Details and Severity Ranking (1 being least severe).

	activityCode	details	severity
▶	1	User sign up	1
	2	User logged in	1
	3	User logged out	1
	4	User failed login attempt	1
	5	User account locked out	2
	6	User account reactivated	1
	7	User tried to access admin page	3

Solution 3: Monitoring

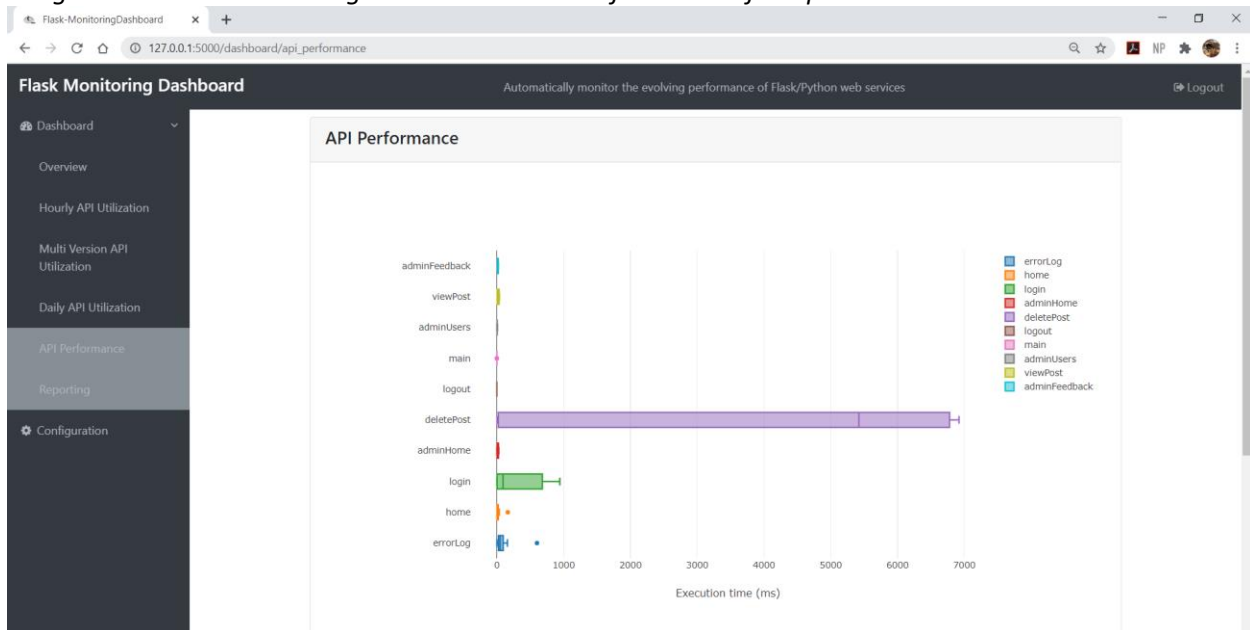
In the vulnerable version of Lorem Ipsum, there was no monitoring of requests and responses.

Hence I implemented the [Flask-Monitoring-Dashboard](#) 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 -