

# **CptS 428/528 Software Security and Reverse Engineering**

## **Fall 2023**

### **Team: Undergraduate ABFMJ-SecureDove**

### **Project Deliverable 3**

**Team Members:** Muhammad Ali Dzulfiqar, Flavio Alvarez Penate, Phearak Both Bunna, Jaysen Anderson, Mitchell Kolb

#### **Objectives:**

- Verify your software product against security goals and metrics
- Develop attacks (e.g., vulnerability exploits) to *\*break\** your software's security, showing the presence of security defects in your product
  - Note that typically if you cannot break your software, it is more likely that you have not identified sufficient ways to break it than your software is too secure to be exploitable/attackable!
- Summarize any findings you obtained during the *\*break\** process, including what means you used and how you succeeded in breaking your software's security.
- Update your security requirements and amend your security plan in light of the above findings (i.e., strengthen your security requirements to ensure it is more secure than before)

#### **Current Security Requirements:**

##### **Security goals:**

- **Confidentiality**
  - When messages are deleted, they are only deleted for the party that wants to delete them (not all parties)
  - Account names are unique to avoid messages going to unwanted users
  - End to end encryption
  - Multi factor authentication for login
  - Users added to a group chat after creation shouldn't have access to previous messages.

- Only the account owner should be able to view messages sent to or from the account
- **Integrity**
  - Message text should not be modifiable by a third party or anyone other than the user that sent them.
  - For an edited message it should be marked as “Edited” in the UI for the application.
  - Other message aspects, such as time sent, sender, etc, should not be editable by any source (not even the user).
  - Users shouldn’t be able to be simultaneously logged into multiple accounts.
- **Availability**
  - The application should be able to resist denial of service attacks and be always available to the user.
  - The application should not provide the user with any frustrations with regards to logging in and making use of its services (i.e. security shouldn’t come at the cost of the user’s ease of use).

#### **Security metrics:**

- **Confidentiality**
  - The percentage of how many cases where the messages that got sent are viewed by another person who is not the sender or receiver should be 0%.
- **Integrity**
  - The percentage of how many cases where the messages have their body modified due to malicious attack, software or hardware failures after being sent by the sender should be 0%.
- **Availability**
  - The percentage of messages that are lost due to system failures should be 0%

## Known Security Vulnerabilities in dependencies

```
PS C:\Users\Both\Desktop\CPTS 428\Project\CptS428-ABFMJ-SecureDove\securedove\frontend> npm audit
# npm audit report

@babel/traverse <7.23.2
Severity: critical
Babel vulnerable to arbitrary code execution when compiling specifically crafted malicious code - https://github.com/advisories/GHSA-67hx-6x53-jw92
fix available via `npm audit fix`
node_modules/@babel/traverse

nth-check <2.0.1
Severity: high
Inefficient Regular Expression Complexity in nth-check - https://github.com/advisories/GHSA-rp65-9cf3-cjxr
fix available via `npm audit fix --force`
Will install react-scripts@3.0.1, which is a breaking change
node_modules/nth-check
  css-select <=3.1.0
  Depends on vulnerable versions of nth-check
  node_modules/css-select
    svgo 1.0.0 - 1.3.2
    Depends on vulnerable versions of css-select
    node_modules/svgo
      @svgr/plugin-svgo <=5.5.0
      Depends on vulnerable versions of svgo
      node_modules/@svgr/plugin-svgo
        @svgr/webpack 4.0.0 - 5.5.0
        Depends on vulnerable versions of @svgr/plugin-svgo
        node_modules/@svgr/webpack
          react-scripts >=2.1.4
          Depends on vulnerable versions of @svgr/webpack
          Depends on vulnerable versions of resolve-url-loader
          node_modules/react-scripts

postcss <8.4.31
Severity: moderate
PostCSS line return parsing error - https://github.com/advisories/GHSA-7fh5-64p2-3v2j
fix available via `npm audit fix --force`
Will install react-scripts@3.0.1, which is a breaking change
node_modules/postcss
node_modules/resolve-url-loader/node_modules/postcss
  resolve-url-loader 0.0.1-experiment-postcss || 3.0.0-alpha.1 - 4.0.0
  Depends on vulnerable versions of postcss
  node_modules/resolve-url-loader

9 vulnerabilities (2 moderate, 6 high, 1 critical)
```

- **Findings:**

- Our team found 9 security vulnerabilities associated with our npm (*node package manager*) packages including the needed dependencies and libraries for our application. Some of the versions of the packages are deprecated and not maintained anymore.
- We were able to check this by using this command: “*npm audit*” which gives us the output as a detailed report of security vulnerabilities along with the severity level.

- **Solution:**

- We can fix this by running this command in the terminal “*npm audit fix --force*”

## Check for user credentials transmitted over the network (using Wireshark)

The image shows a Wireshark network traffic capture. The top pane displays a list of captured packets. The bottom pane shows the details of a selected packet (Frame 1287), which is an HTTP POST request. The payload is a JSON object containing user credentials.

No.	Time	Source	Destination	Protocol	Length	Info
1287	64.992751	127.0.0.1	127.0.0.1	HTTP/J...	753	POST /register HTTP/1.1 , JavaScript Object Notation (application/json)
1585	124.056032	127.0.0.1	127.0.0.1	HTTP/J...	727	POST /login HTTP/1.1 , JavaScript Object Notation (application/json)
1656	134.544535	127.0.0.1	127.0.0.1	HTTP/J...	724	POST /login HTTP/1.1 , JavaScript Object Notation (application/json)

Frame 1287: 753 bytes on wire (6024 bits), 753 bytes captured (6024 bits) on interface \Device\NPF\_{...} Null/Loopback  
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1  
Transmission Control Protocol, Src Port: 51798, Dst Port: 8080, Seq: 1, Ack: 1, Len: 709  
Hypertext Transfer Protocol  
JavaScript Object Notation: application/json  
Object  
Member: username  
[Path with value: /username:Phearak Both]  
[Member with value: username:Phearak Both]  
String value: Phearak Both  
Key: username  
[Path: /username]  
Member: email  
[Path with value: /email:phearakboth@gmail.com]  
[Member with value: email:phearakboth@gmail.com]  
String value: phearakboth@gmail.com  
Key: email  
[Path: /email]  
Member: password  
[Path with value: /password:test123]  
[Member with value: password:test123]  
String value: test123  
Key: password  
[Path: /password]

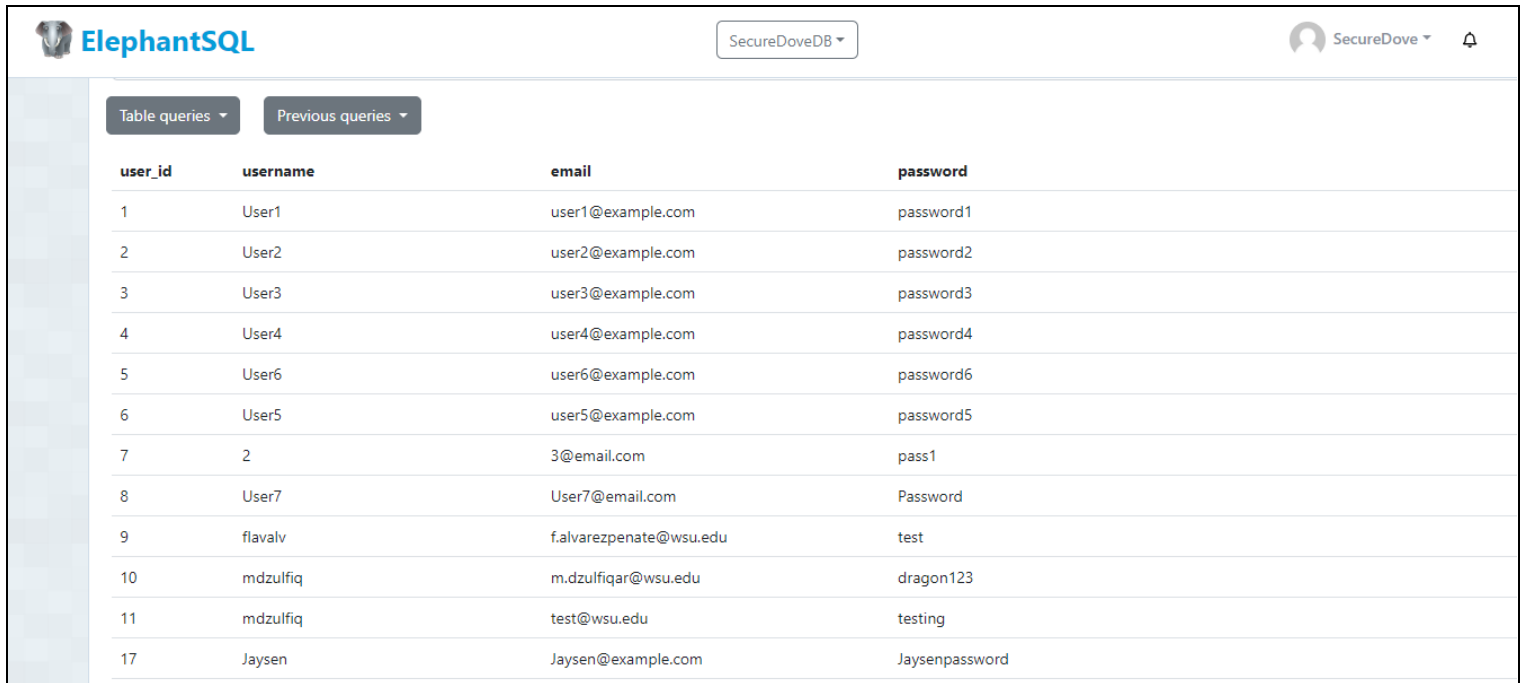
- Our team performed a network traffic analysis using *Wireshark* to check if the credentials such as *Email & Password* are shown over the network after the user submitted the login form.
- **Findings:**
  - The *Email* along with the *Password* that the user typed in the login form, are displayed in plain text over network traffic.
  - This is one of the critical security vulnerabilities that our messaging app is currently facing. If a perpetrator was to capture the network traffic on our instant messaging web app, they could gain access to our users' accounts with the credentials that are shown on the network traffic.
- **Solutions:**
  - One way that we can resolve this is to use **HTTPS** (HyperText Transfer Protocol Secure) instead of **HTTP**
  - Another way that we can go about this is to securely handle the passwords that the users entered on our login page by using strong hashing algorithms that are

available to us such as *crypto-js* (a JavaScript library used to perform cryptographic operations).

http.request.method == "POST"						
No.	Time	Source	Destination	Protocol	Length	Info
82	14.452196	127.0.0.1	127.0.0.1	HTTP/1.1	778	POST /login HTTP/1.1 , JavaScript Object Notation (application/json)
Frame 82: 778 bytes on wire (6224 bits), 778 bytes captured (6224 bits) on interface \Device\NPF_{...} Null/Loopback						
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1						
Transmission Control Protocol, Src Port: 58537, Dst Port: 8000, Seq: 1, Ack: 1, Len: 734						
Hypertext Transfer Protocol						
JavaScript Object Notation: application/json						
Object						
Member: email						
[Path with value: /email:lasthash@test.com]						
[Member with value: email:lasthash@test.com]						
String value: lasthash@test.com						
Key: email						
[Path: /email]						
Member: password						
[Path with value: /password:673d190b758967621da243f06c350ce68be4276174dc886560239fea923d4a5a]						
[Member with value: password:673d190b758967621da243f06c350ce68be4276174dc886560239fea923d4a5a]						
String value: 673d190b758967621da243f06c350ce68be4276174dc886560239fea923d4a5a						
Key: password						
[Path: /password]						
0040 0d 0a 48 6f 73 74 3a 20 6c 6f 63 61 6c 68 6f 73 ..Host: localhost						
0050 74 3a 38 30 30 30 0d 0a 43 6f 6e 6e 65 63 74 69 t:8000.. Connecti						
0060 6f 6e 3a 20 6b 65 65 70 2d 61 6c 69 76 65 0d 0a on: keep-alive..						
0070 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74 68 3a 20 Content-Length:						
0080 31 30 37 0d 0a 73 65 63 2d 63 68 2d 75 61 3a 20 107..sec -ch-ua:						
0090 22 43 68 72 6f 6d 69 75 6d 22 3b 76 3d 22 31 31 "Chromiu m";v="11						
00a0 38 22 2c 20 22 47 6f 6f 67 6c 65 20 43 68 72 6f 8", "Goo gle Chro						
00b0 6d 65 22 3b 76 3d 22 31 31 38 22 2c 20 22 4e 6f me";v="1 18", "No						
00c0 74 3d 41 3f 42 72 61 6e 64 22 3b 76 3d 22 39 39 t=A?Bran d";v="99						
00d0 22 0d 0a 41 63 63 65 70 74 3a 20 61 70 70 6c 69 ".Accep t: appli						
00e0 63 61 74 69 6f 6e 2f 6a 73 6f 6e 2c 20 74 65 78 cation/j son, tex						
00f0 74 2f 70 6c 61 69 6e 2c 20 2a 2f 2a 0d 0a 43 6f t/plain, /*..Co						
0100 6e 74 65 6e 74 2d 54 79 70 65 3a 20 61 70 70 6c ntent-Ty pe: appl						
0110 69 63 61 74 69 6f 6e 2f 6a 73 6f 6e 0d 0a 73 65 ication/ json..se						
0120 63 2d 63 68 2d 75 61 2d 6d 6f 62 69 6c 65 3a 20 c-ch-ua- mobile:						
0130 3f 30 0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 ?0..User -Agent:						
0140 4d 6f 7a 69 6c 6c 61 2f 35 2e 30 20 28 57 69 6e Mozilla/ 5.0 (Win						
0150 64 6f 77 73 20 4e 54 20 31 30 2e 30 3b 20 57 69 dows NT 10.0; Wi						
0160 6e 36 3a 3b 20 78 36 34 29 20 41 70 70 6c 65 57 n64; x64 ) AppleW						
0170 65 62 4b 69 74 2f 35 33 37 2e 33 36 20 28 4b 48 ebKit/53 7.36 (KH						
0180 54 4d 4c 2c 20 6c 69 6b 65 20 47 65 63 6b 6f 29 TML, lik e Gecko)						
0190 20 43 68 72 6f 6d 65 2f 31 31 38 2e 30 2e 30 2e Chrome/ 118.0.0.						
01a0 30 20 53 61 66 61 72 69 2f 35 33 37 2e 33 36 0d 0 Safari /537.36.						
01b0 0a 73 65 63 2d 63 68 2d 75 61 2d 70 6c 61 74 66 .sec-ch- ua-platf						
01c0 6f 72 6d 3a 20 22 57 69 6e 64 6f 77 73 22 0d 0a orm: "Wi ndows"..						
01d0 4f 72 69 67 69 6e 3a 20 68 74 74 70 3a 2f 2f 6c Origin: http://l						

- Now we can see that the password is not shown in plain-text anymore over the network traffic. Instead, it's showing as a hash which is a good practice to securely handle passwords and confidential or sensitive data.
- We used the **SHA-256** algorithm that comes with *crypto-js* and this is a secure hashing algorithm since it's making sure that it is impossible for the hashed passwords to be **reversed or decoded**. We can call it a **1-way hashing**.
- What this does is that everytime the user enters their *Email* and *Password* into the login form on our site, the password will be hashed before sending over and compared with the hashed password stored in our database.

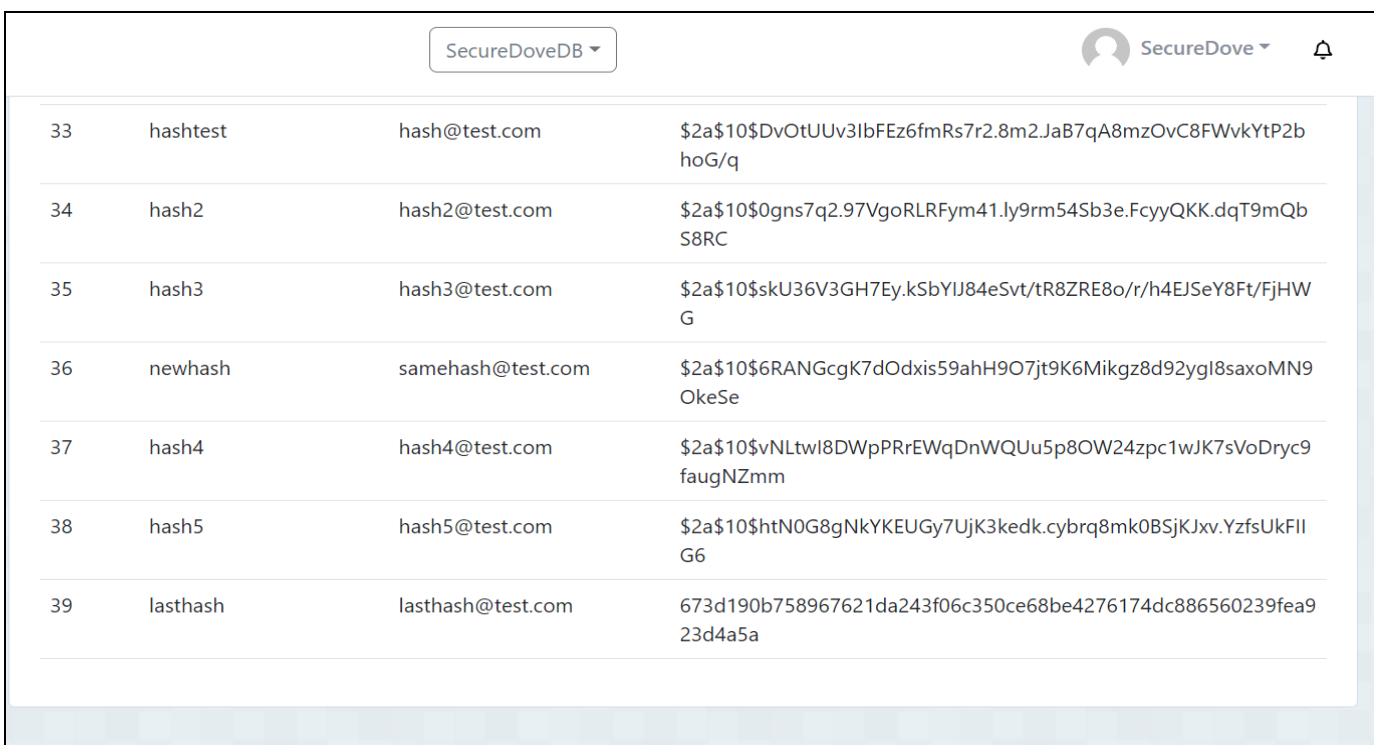
## Passwords are displayed in plain-text in the database



The screenshot shows the ElephantSQL interface for a database named 'SecureDoveDB'. It displays a table with four columns: 'user\_id', 'username', 'email', and 'password'. The 'password' column contains plain-text passwords for various users, including 'password1', 'password2', 'password3', 'password4', 'password6', 'password5', 'pass1', 'Password', 'test', 'dragon123', 'testing', and 'Jaysenpassword'.

user_id	username	email	password
1	User1	user1@example.com	password1
2	User2	user2@example.com	password2
3	User3	user3@example.com	password3
4	User4	user4@example.com	password4
5	User6	user6@example.com	password6
6	User5	user5@example.com	password5
7	2	3@email.com	pass1
8	User7	User7@email.com	Password
9	flavalv	f.alvarezpenate@wsu.edu	test
10	mdzulfiq	m.dzulfiqar@wsu.edu	dragon123
11	mdzulfiq	test@wsu.edu	testing
17	Jaysen	Jaysen@example.com	Jaysenpassword

- Our team utilizes **ElephantSQL** as our database to store sensitive and confidential users information such as the passwords and messages.
- As we can see from the screenshot above, all the passwords are displayed in plain-text which is not a good practice in handling sensitive data.
- **Solution:**
  - We decided to hash the passwords that the users entered in the registration form before storing it in our database (using **SHA-256** hashing algorithm). This will ensure that the passwords will not be displayed in plain-text anymore.



The screenshot shows the ElephantSQL interface for the same database 'SecureDoveDB'. It displays a table with four columns: 'id', 'username', 'email', and 'password'. The 'password' column now contains SHA-256 hashes of the previous passwords, such as '\$2a\$10\$DvOtUUV3IbFEz6fmRs7r2.8m2.JaB7qA8mzOvC8FWvkYtP2bhoG/q'.

id	username	email	password
33	hashtest	hash@test.com	\$2a\$10\$DvOtUUV3IbFEz6fmRs7r2.8m2.JaB7qA8mzOvC8FWvkYtP2bhoG/q
34	hash2	hash2@test.com	\$2a\$10\$0gns7q2.97VgoRLRFym41.ly9rm54Sb3e.FcyyQKK.dqT9mQbS8RC
35	hash3	hash3@test.com	\$2a\$10\$skU36V3GH7Ey.kSbYIJ84eSvt/tr8ZRE8o/r/h4EJSeY8Ft/FjHWG
36	newhash	samehash@test.com	\$2a\$10\$6RANGcgK7dOdxis59ahH9O7jt9K6Mikgz8d92ygl8saxoMN9OkeSe
37	hash4	hash4@test.com	\$2a\$10\$vNltwi8DWpPRrEWqDnWQUu5p8OW24zpc1wJK7sVoDryc9faugNZmm
38	hash5	hash5@test.com	\$2a\$10\$htN0G8gNkyKEUGy7UjK3kedk.cybrq8mk0BSjKJxv.YzfsUkFIIG6
39	lasthash	lasthash@test.com	673d190b758967621da243f06c350ce68be4276174dc886560239fea923d4a5a

## Brute force password attacks

- During the registration process, the only password that our application does not allow users to enter is an empty password. This means that there's no character length or special character requirements.
- Users can enter a password as simple as "pass" or even just "a", and our application allows it.
- These passwords are very easy to crack through various methods, and anyone could get access with enough attempts in a reasonable amount of time.
  - Additionally, an attacker could try an unlimited amount of password combinations, since our program does not have a limit.
- **Findings:**
  - Using the first method of randomly guessing a password of the same length, it took only 22.7 seconds in order to crack a four character password.
  - Using the second method that methodically generates every password possible and avoids any duplicates, it was significantly faster to "guess" the test password in comparison. Any four character password can be deciphered in under a second with this method.
- Our team has created a Python notebook that demonstrates how quick and easy some of these short and simple passwords can be cracked using two different methods.

### Method 1: Randomly guessing a password that matches in length

```
# create function that randomly creates a password from a character set provided and compares it to the entered password
# allows repeats, follows no methodology (i.e. it's all random)
def rand_crack_pass(char_set, password):
    guess = ""
    attempts = 0
    while (guess != password):
        guess = ''.join(random.choices(char_set, k=len(password)))
        attempts += 1
        if (guess == password):
            return attempts, guess
```

[41] ✓ 0.0s

```
attempts, guess = rand_crack_pass(avail_chars, "test")
print("Password: ", guess)
print("Attempts: ", attempts)
```

[42] ✓ 22.7s

... Password: test  
Attempts: 26087849

## Method 2: Generating every possible password and comparing

+ Code + Markdown

```
# generates every possible password of the specified length in order based on the provided character set
def crack_password(char_set, password):
    count = 0
    for item in itertools.product(char_set, repeat=len(password)):
        guess = ''.join(item)
        count += 1
        if (guess == password):
            return count, guess
```

[45] ✓ 0.0s

Testing the same password that we did with the random method above

```
# call function and print results
attempts, guess = crack_password(avail_chars, "test")
print("Password: ", guess)
print("Attempts: ", attempts)
```

[46] ✓ 0.4s

... Password: test  
Attempts: 4544744

**Outcome:** Significantly more effective at cracking a 4 character password

- **Solutions:**

- Every password entered by a user at the registration step should have a minimum character length of eight, contain at least one upper case and one lower case letter, contain at least one special character from “!@#\$%^&\*()” and contain at least one number 0-9.
- This will increase password length, along with the character set that passwords have to be guessed from. Increasing both of these will significantly increase the possible number of passwords that can be created.
- Finally, users that try to log in with an incorrect password more than five times should be timed out from attempting any more logins for a certain period of time.
- However, we want to ensure that we avoid any measures that would result in frustration for the user (such as making the password have to be 32 characters minimum, which would tremendously improve security but make the registration/login process frustrating).



## Database API Key Being Visible

- **Findings:**

- For SecureDove we use a third party postgresql hosting service called ElephantSQL. To use this service in our FastAPI backend we have to connect to the service using an API key. This key is very confidential and allows all users who have it, direct admin access to view, add, edit, and delete any and all data in our database using SQL code.
- Therefore it is important to make this API key string private when possible. The picture below shows the section of code in our securedove/backend/main.py file where we currently have the api key visible to all.

```
53 try:
54     #Should really put line 55 into a secure file that doesn't go on github but until everyone can run this code i'll leave it in the file as is. Security wise this is bad procedure
55     conn = psycopg2.connect('postgres://wjjloedt:JXfCuJ7Di3CtZ-enIemY93_m8VxNXpZ@berry.db.elephantsql.com/wjjloedt')
56     print('Connection Success!')
57     connectionsucceeded = True
58 except:
59     print("Unable to connect to the database")
60
61 # Open a cursor to execute SQL queries
62 cur = conn.cursor()
63
```

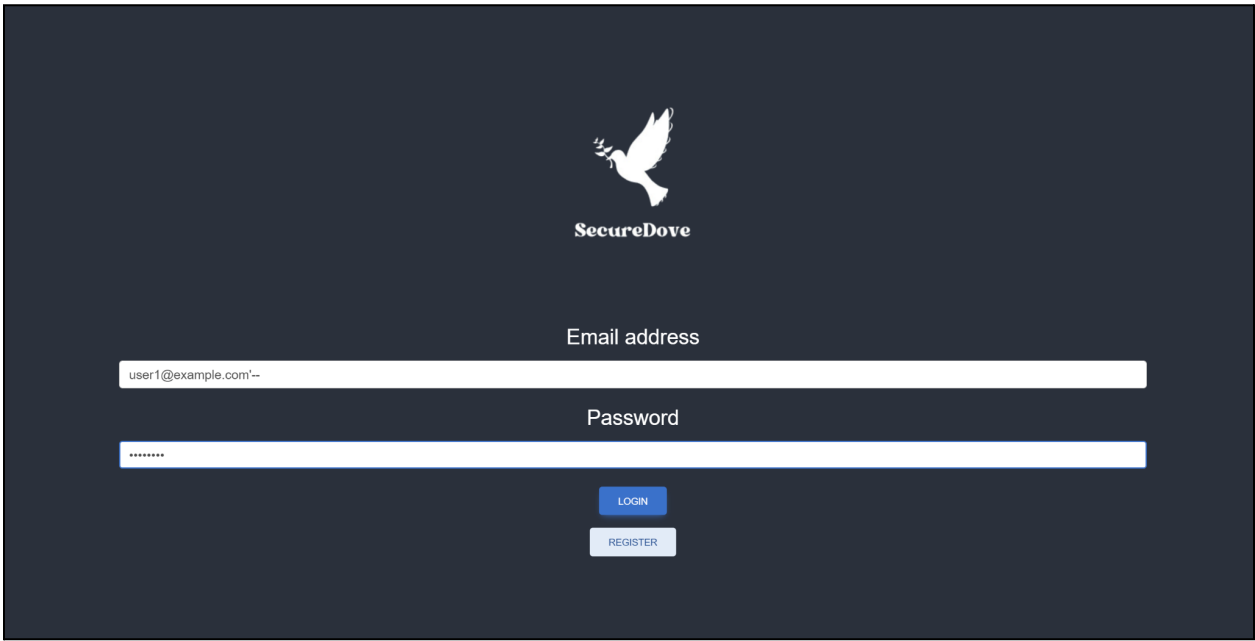
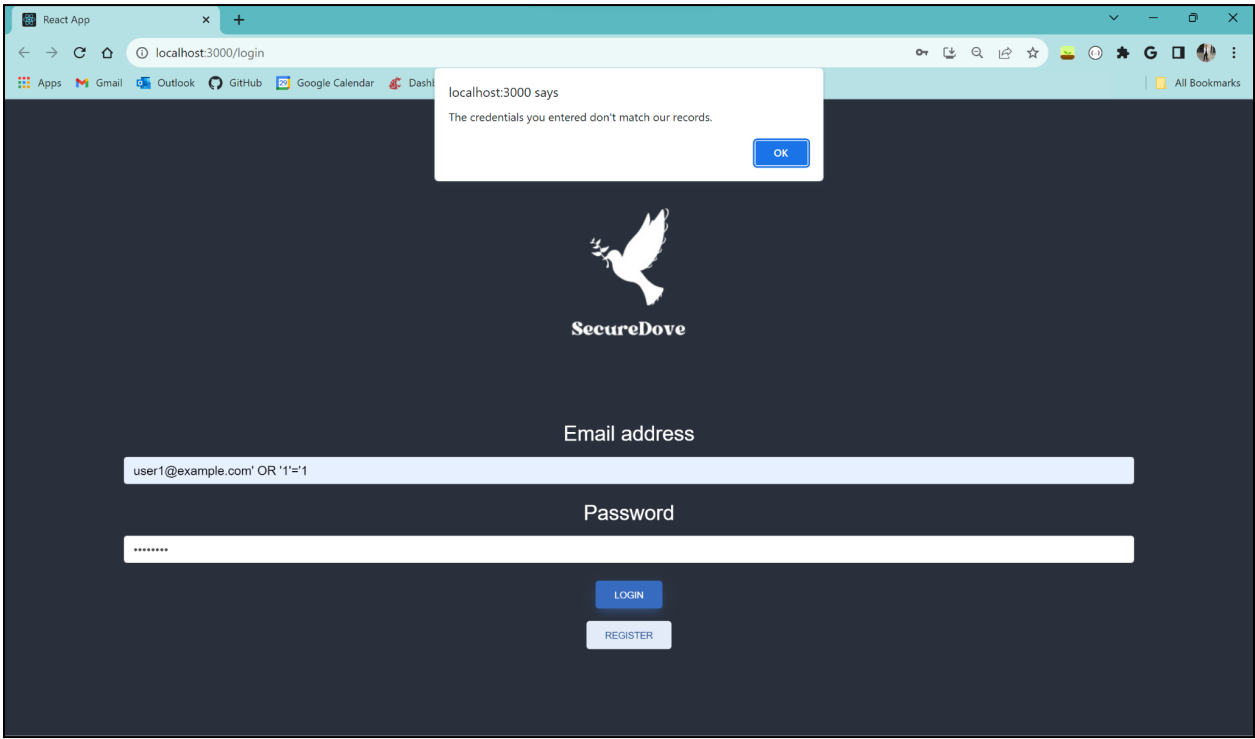
- **Solutions:**

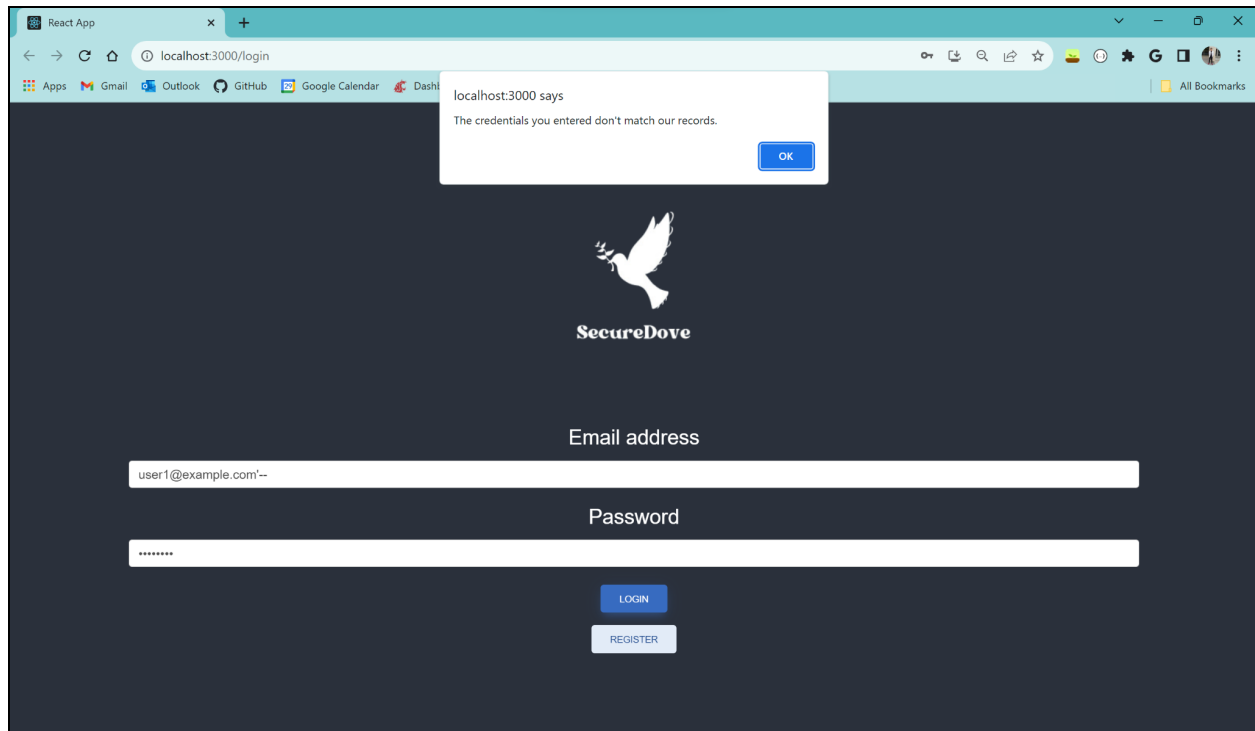
- A proposed solution to this problem is to separate the api key into its own file called “credentials.py” which contains a variable called “api\_key” that has the value of the key.
- We would then import the credentials.py into the main.py file and then access the value by using the line of code below:

```
conn = psycopg2.connect('{credentials.api_key}')
```

- We would then add the credentials.py file to the .gitignore so the api key value stays secure only on the developers machines and we would send out the file to others as they need it if the team were to expand. This allows our project to stay on github while giving us the security that not anyone who views our codebase has the ability to access our database.

# SQL Injection OR Payload and Comment Payload Testing

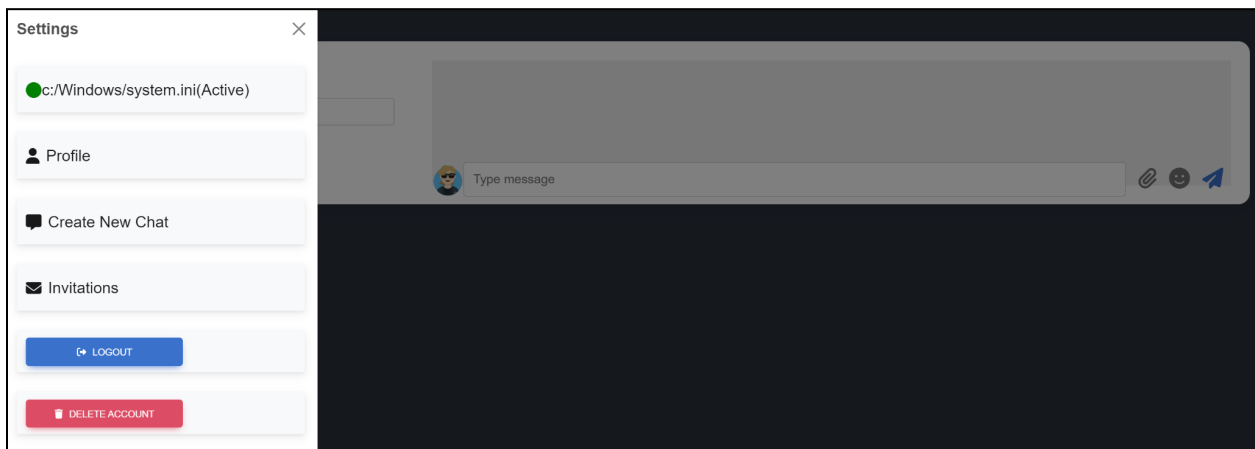
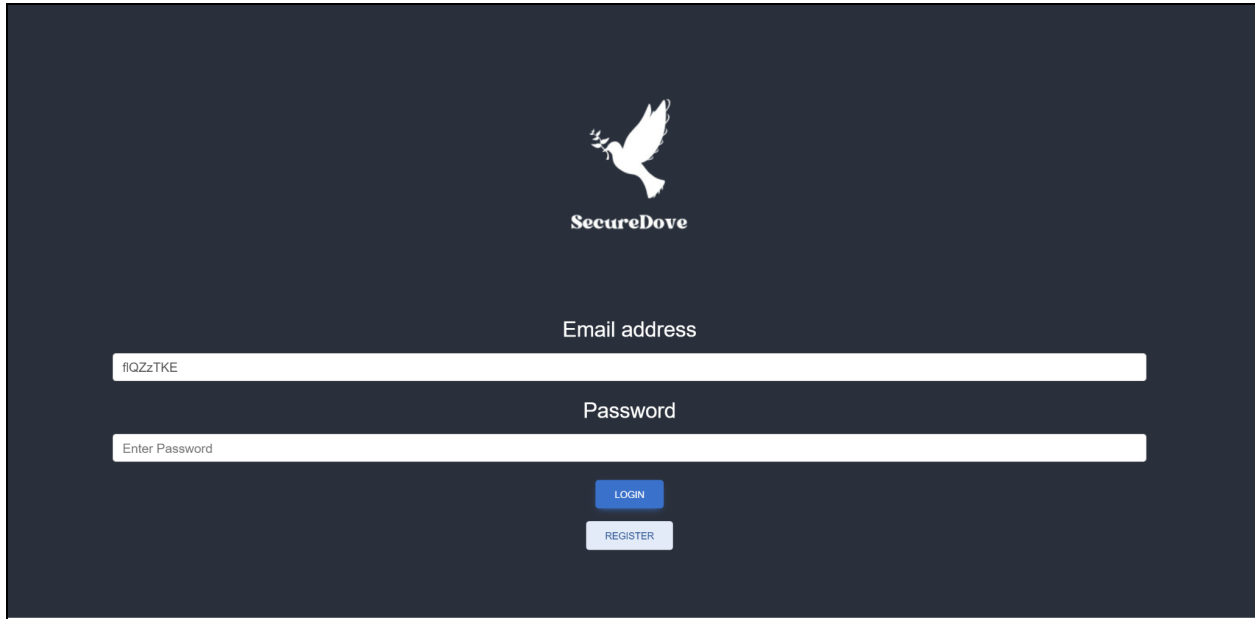




- These methods don't completely crack the logging in. As you can see we still get an error that says that the credentials don't match. However, these methods do get us past the email checking stage which is still a vulnerability.
- **Solution:**
  - Ensure that all SQL queries are parameterized.
  - Validate and sanitize all user input. This could be done by looking for keywords. For example, we could just decide that equal signs can't be in our passwords. That way we just search for equal signs and if we ever find one, we instantly reject it.

## SQL Injection Using SQLMap

user_id	username	email	password
1	User1	user1@example.com	password1
2	User2	user2@example.com	password2
3	User3	user3@example.com	password3
4	User4	user4@example.com	password4
5	User6	user6@example.com	password6
6	User5	user5@example.com	password5
7	2	3@email.com	pass1
8	User7	User7@email.com	Password
9	flavalv	f.alvarezpenate@wsu.edu	test
10	mdzulfiq	m.dzulfiqar@wsu.edu	dragon123
11	mdzulfiq	test@wsu.edu	testing
17	Jaysen	Jaysen@example.com	Jaysenpassword
23	c:/Windows/system.ini	fIQZzTKE	
25	Phearak Both	phearakboth@gmail.com	test123



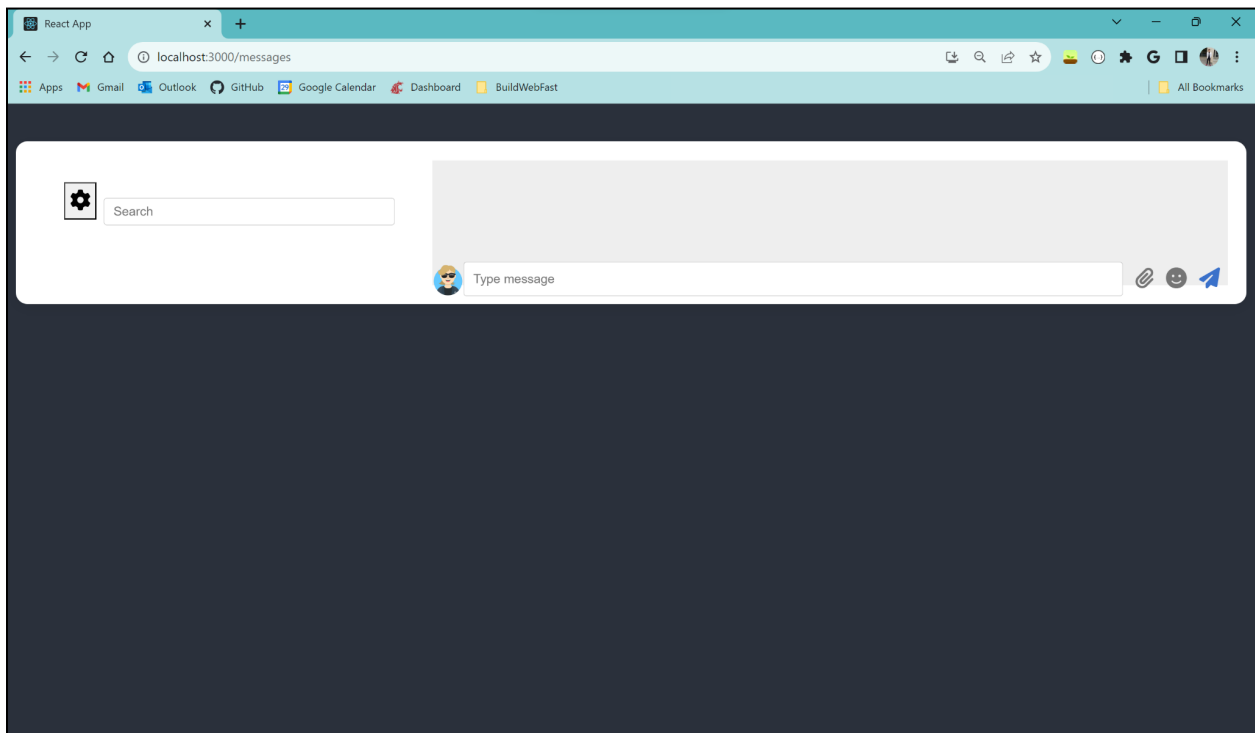
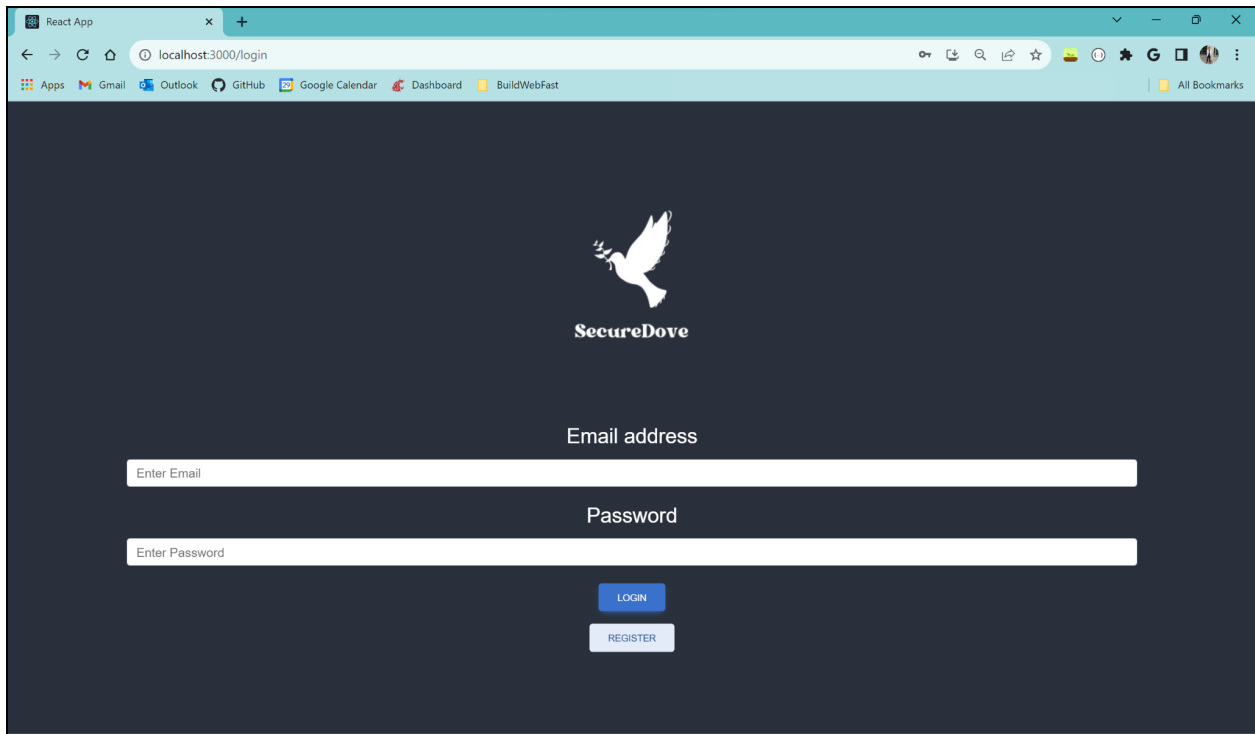
## Findings:

- Our team performed an SQL injection using **sqlmap** in order to test out whether we can add a new user to the database without needing a password.
- We were able to add a new user to the database without having a password even though the frontend of our application is designed to always require a password for both the registration and login forms.
- As shown in the screenshots above, we were able to login to the application with just **flQZzTKE** (not in an email format with the @ and no password).

**Solution:**

- One way we can go about solving this is to design not only the frontend of our application but also the backend of our application to check for the missing *Password* field and to make sure that the email is of email format (with @).
- This will ensure that no new accounts can be created without a password or not in an email format.

# URL Open Redirection Vulnerability



- **Findings:**
  - It is possible to bypass the login by modifying the URL from localhost:3000/login to localhost:3000/messages.

- The landing page only shows the messages of the recently logged user.
- **Solutions:**
  - One way to deal with URL Open Redirection Vulnerability is to remove the URL parameter to prevent direct redirection post-login.