

Penetration Testing Report

*Prepared for: Bank of
Charles*

*Prepared by:
Manan Amitkumar Patel*

Contents:

Executive Summary..... 3

Technical Summary..... 3

 Scope 3

 Key Findings: 3

 Key Recommendations:..... 4

Findings and Recommendations 5

 Findings 5

 Plaintext Storage of Password: 5

 Cross-Site Request Forgery (CSRF) in Account Creation Form: 6

 Session Hijacking via Reused Session Cookies: 10

 Insecure Login Password Authentication and SQL Injection:13

 Insecure Cookie-Based Access Control: 16

 Cross-Site Scripting (XSS):.....17

 Insecure Authentication Bypass via Cookie Manipulation:20

 Information Disclosure via Debug Parameter:22

 Command Injection via Query Parameters:24

Appendices 26

 Appendix A – Risk Matrix26

Executive Summary

This penetration testing report provides a comprehensive analysis of the security vulnerabilities identified in the Charles Bank's newly rearchitected rewards platform. Conducted by Manan Amitkumar Patel and Rehan Umatia, the assessment involved a detailed code review and web application testing using Burp Suite. The evaluation revealed several critical security issues, including plaintext password storage, Cross-Site Request Forgery (CSRF), session hijacking, Cross Site Scripting (XSS), insecure authentication, SQL injection vulnerabilities and many more. The identified weaknesses pose significant risks to user data integrity and system security. Immediate remediation actions have been recommended and implemented to mitigate these risks and enhance the overall security posture of the application.

Technical Summary

Scope

Attribute	Value
IP / Hostname	Localhost:5000
Testing Type	Web Application Penetration Testing
User	Penetration Tester

Key Findings:

- **Plaintext Storage of Passwords:** User passwords are stored in plaintext within the database, posing a significant security risk as attackers can easily access all user passwords if the database is compromised.
- **Cross-Site Request Forgery (CSRF) in Account Creation Form:** The account creation form lacks CSRF protection, allowing attackers to perform unauthorized actions on behalf of authenticated users.
- **Session Hijacking via Reused Session Cookies:** Reused session cookies can be captured and reused to access user accounts, indicating improper session management.
- **Insecure Login Password Authentication and SQL Injection:** Weak password validation and SQL injection vulnerabilities allow unauthorized access to user accounts.
- **Insecure Cookie-Based Access Control:** Admin page access relies on client-side cookies, which can be manipulated to gain unauthorized access.
- **Cross-Site Scripting (XSS):** The application is vulnerable to XSS attacks, allowing attackers to steal cookies or perform other malicious actions.
- **Insecure Authentication Bypass via Cookie Manipulation:** Attackers can manipulate session cookies to bypass authentication and delete accounts.
- **Information Disclosure via Debug Parameter:** Endpoints contain a debug parameter that exposes sensitive user information.

- **Command Injection via Query Parameters:** The endpoint allows command injection through query parameters, leading to unauthorized access and system compromise.

Key Recommendations:

- **Hash and Salt Passwords:** Implement strong password hashing algorithms (such as bcrypt) along with salting techniques before storing passwords in the database.
- **Enable CSRF Protection:** Use Flask-WTF's built-in CSRF protection mechanism to secure all forms in the application.
- **Invalidate Session on Logout:** Ensure the session is properly invalidated when the user logs out by clearing the session cookie on the server side.
- **Strengthen Password Validation and Implement SQL Injection Prevention:** Use proper password validation and parameterized queries to prevent SQL injection attacks.
- **Implement Server-Side Access Control:** Use robust server-side mechanisms to verify admin privileges and prevent cookie manipulation.
- **Sanitize User Input and Encode Output:** Implement input sanitization to remove or escape malicious scripts before storing them in the database and ensure user-provided data is properly encoded before rendering in HTML.
- **Remove Insecure Cookie Handling and Implement Proper Account Status Checks:** Eliminate the use of cookies for authentication, ensure only active accounts can log in, and properly handle account deletion.
- **Remove Debug Functionality in Production:** Ensure debug-related code is removed or disabled in production environments.
- **Remove Command Execution Functionality:** Disable any functionality that allows executing system commands from user inputs.

Findings and Recommendations

Findings

Plaintext Storage of Password:

Risk: Critical	Likelihood: Very Likely	Consequence: Catastrophic
-----------------------	--------------------------------	----------------------------------

Description:

Plaintext Storage of Passwords refers to storing user passwords in their raw, unencrypted form within the database. This practice is a significant security risk because if the database is compromised, all user passwords are immediately accessible. Secure password storage requires hashing and salting passwords before storing them to protect against unauthorized access.

Impact:

Storing passwords in plaintext can lead to severe security breaches. If an attacker gains access to the database, they can obtain all user passwords, leading to unauthorized access to user accounts, potential data theft, and further exploitation of other systems where users might have reused the same passwords. This vulnerability compromises user trust and can result in significant reputational and financial damage to the organization.

Evidence:

The code below shows that user passwords are stored in plaintext:

```
@app.route("/create", methods=["GET", "POST"])
def create_account():
    form = CreateForm()
    if form.validate_on_submit():
        name = form.name.data
        password = form.password.data
        new_account = Account(name, password, 100)
        db.session.add(new_account)
        db.session.commit()
        new_transaction = Transaction(
            "Balance Add.", "Rewards account opened.", new_account.id, 100
        )
        db.session.add(new_transaction)
        db.session.commit()
        session["username"] = new_account.name

        return redirect(url_for("display.my_account"))

    return render_template("create_account.html", form=form)
```

Recommendation:

- Hash and Salt Passwords: Use a strong hashing algorithm to hash and salt passwords before storing them in the database. This prevents attackers from easily recovering passwords if the database is compromised.
- Update User Registration: Ensure that all points where user passwords are handled in registration form use the hashing mechanism.

Action taken:

- Implemented password hashing using generate_password_hash from Flask's werkzeug.security module.
- Updated the create_account function to hash passwords before storing them.

```
from werkzeug.security import generate_password_hash, check_password_hash

app = Blueprint('accounts', __name__, template_folder='templates')

@app.route("/create", methods=["GET", "POST"])
def create_account():
    form = CreateForm()
    if form.validate_on_submit():
        name = form.name.data
        password = form.password.data # This is the plain password from the client
        server_hashed_password = generate_password_hash(password) # Hash the password on the server side
        new_account = Account(name=name, password=server_hashed_password, balance=100) # Store hashed password
        db.session.add(new_account)
        db.session.commit()
        session["username"] = new_account.name
        return redirect(url_for("display.my_account"))
    return render_template("create_account.html", form=form)
```

References

Cross-Site Request Forgery (CSRF) in Account Creation Form:

Risk: High	Likelihood: Likely	Consequence: Major
-------------------	---------------------------	---------------------------

Description:

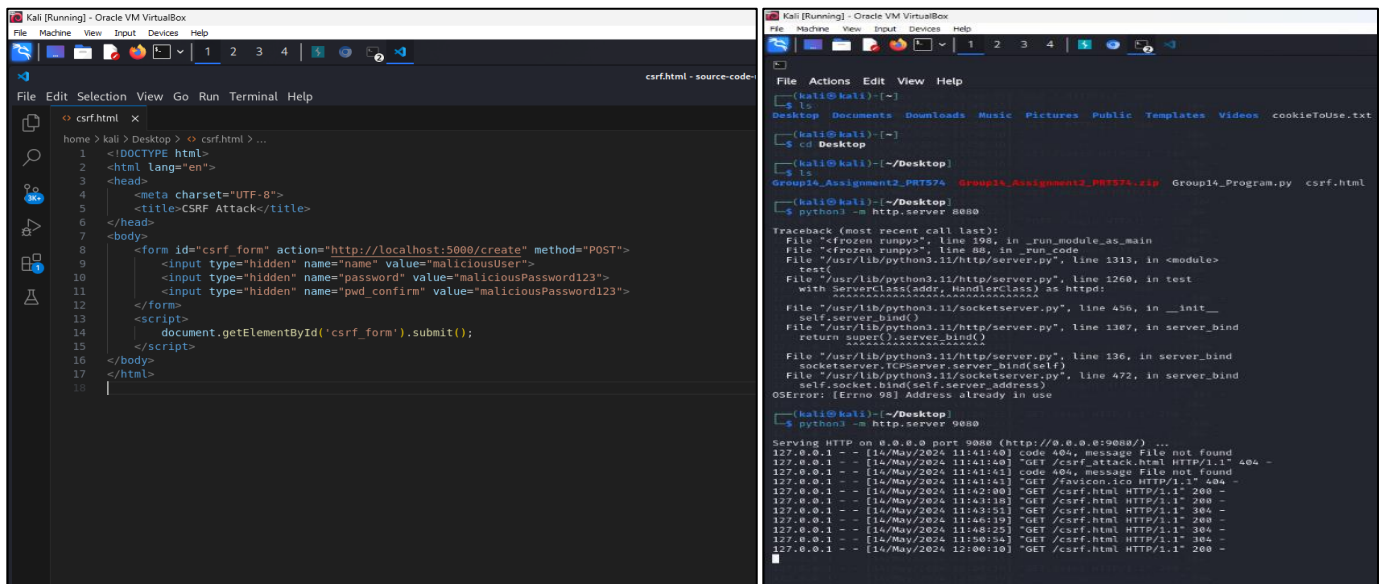
Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious website causes a user's web browser to perform an unwanted action on a trusted site where the user is authenticated. In this case, the account creation form at <http://localhost:5000/create> lacks CSRF protection, making it vulnerable to such attacks. An attacker can exploit this by tricking a user into submitting a form that creates a new account without their permission.

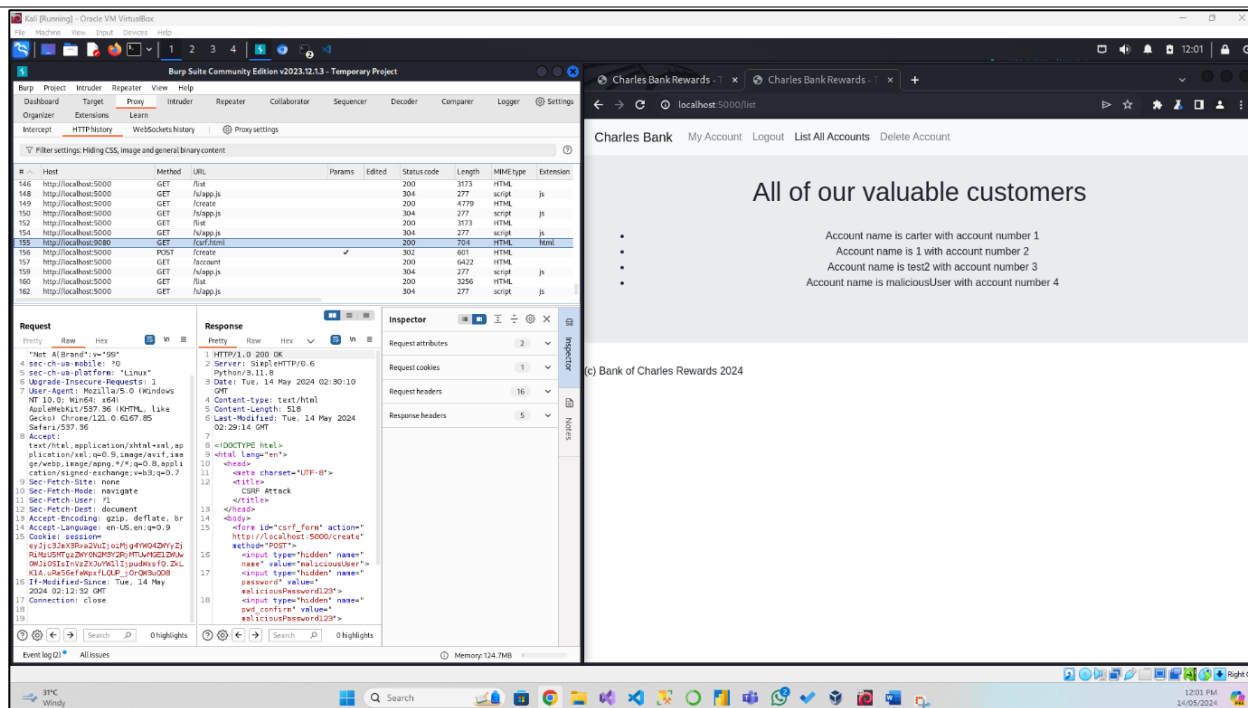
Impact:

- **Unauthorized Actions:** Attackers can perform actions on behalf of authenticated users without their knowledge, leading to unauthorized account creation.
- **Compromised Security:** CSRF can be used in combination with other vulnerabilities to escalate privileges or gain unauthorized access to the system.
- **Loss of User Trust:** Users might lose trust in the application if they discover that actions can be performed without their consent.

Evidence:

- Created a CSRF attack page (csrf_create_account.html) with a form that submits account creation data to `http://localhost:5000/create`.
- Hosted the page on a local web server and navigated to it.
- The form automatically submitted and successfully created a new account without any CSRF token validation.





Recommendation:

- Enable CSRF Protection Globally: Use Flask-WTF's built-in CSRF protection mechanism to secure all forms in the application.
- Modify Form Definitions: Remove the csrf = False or set csrf = True, in the form definitions to ensure CSRF tokens are included and validated.
- Update Application Configuration: Ensure the application is properly configured to support CSRF protection.

Actions Taken:

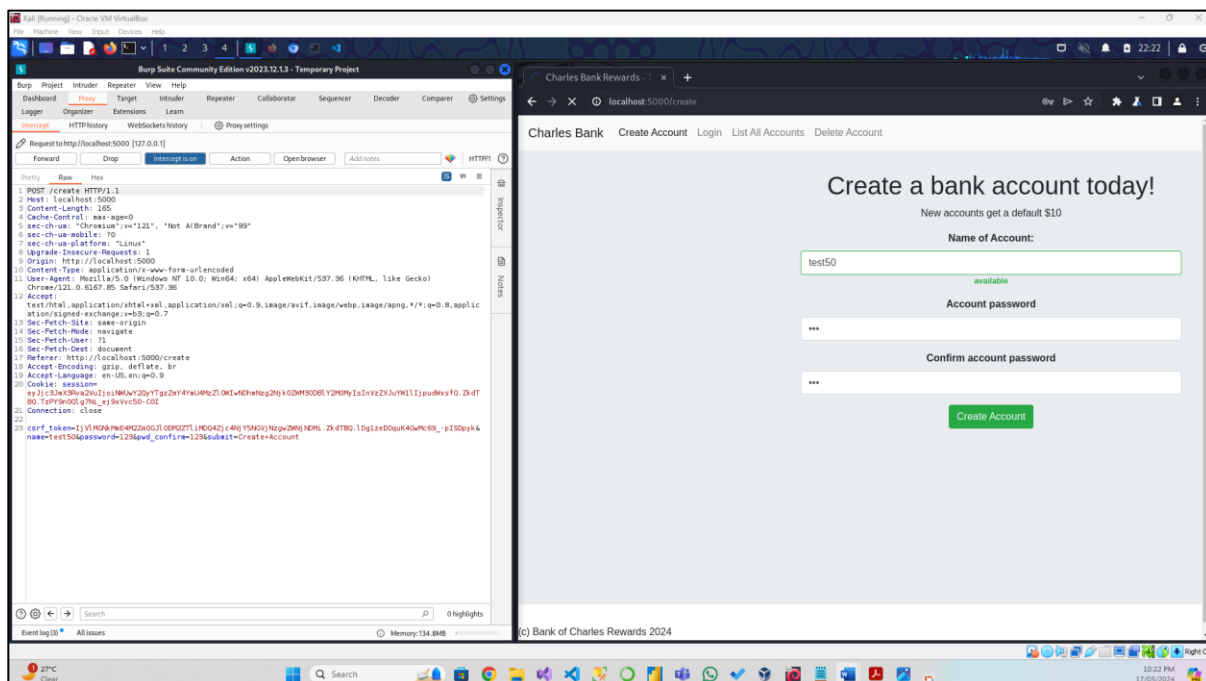
- Enable CSRF Protection in forms.py:
 - Removed csrf = False from all form classes or set csrf = True both cases should be fine.
 - Updated forms.py to include CSRF protection for all forms.
- Update app.py to Enable CSRF Protection:
 - Initialize CSRF protection by adding CSRFProtect from flask_wtf.


```

accounts.py M login.html M create_account.html M app.py M x
app.py > ...
1 import os
2 from flask import Flask
3 from db.base import db
4 from flask_migrate import Migrate
5 from flask_wtf.csrf import CSRFProtect
6
7 # Import views
8 from views.accounts import app as accounts_view
9 from views.base import app as base_view
10 from views.display import app as display_view
11 from views.admin import app as admin_view
12
13 app = Flask(__name__)
14 app.register_blueprint(accounts_view)
15 app.register_blueprint(base_view)
16 app.register_blueprint(display_view)
17 app.register_blueprint(admin_view)
18
19 # Use a strong secret key for session management and CSRF protection
20 SECRET_KEY = os.urandom(32)
21 app.config["SECRET_KEY"] = SECRET_KEY
22
23 app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.db"
24 app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = True
25 app.config["DEBUG"] = True
26
27 # Enable CSRF protection globally for our Flask app
28 csrf = CSRFProtect(app)
29
30 db.init_app(app)
31 Migrate(app, db)
32
33
34 if __name__ == "__main__":
35     app.run(debug=True, host="0.0.0.0", port=5000)
36

```

- `Os.random()` generates a string of random bytes suitable for cryptographic use, ensuring a strong, unpredictable secret key for session management and CSRF protection.
- As we are in debug mode, we are generating secret key every time, but during production time, there should be one confidential secret key.
- Update Templates to Include CSRF Tokens:
 - Ensure all forms include CSRF tokens by using `{{ form.hidden_tag() }}` in the templates.



From the above image we can see that CSRF token has been successfully implemented during POST request.

References

Session Hijacking via Reused Session Cookies:

Risk: High

Likelihood: Likely

**Consequence:
Major**

Description:

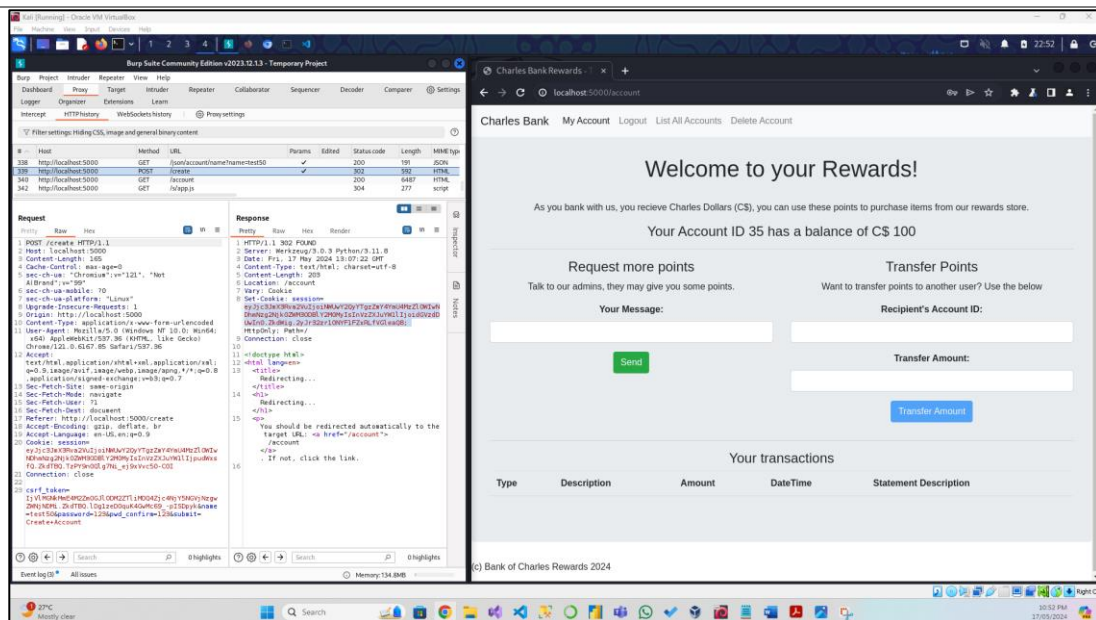
Session Hijacking happens when an attacker captures a user's session cookie and uses it to impersonate the user and access their account without permission. In this case, after creating an account or logging in, the user is redirected to the account page, which requires authentication. During this process, the user's session cookie can be captured. Even after the user logs out, if the attacker visits <http://localhost:5000/account> (a page that requires authentication), intercepts the request, and replaces the current session cookie with the captured one, they can access the account page without logging in again. This shows that the session management is not properly invalidating session cookies upon logout.

Impact:

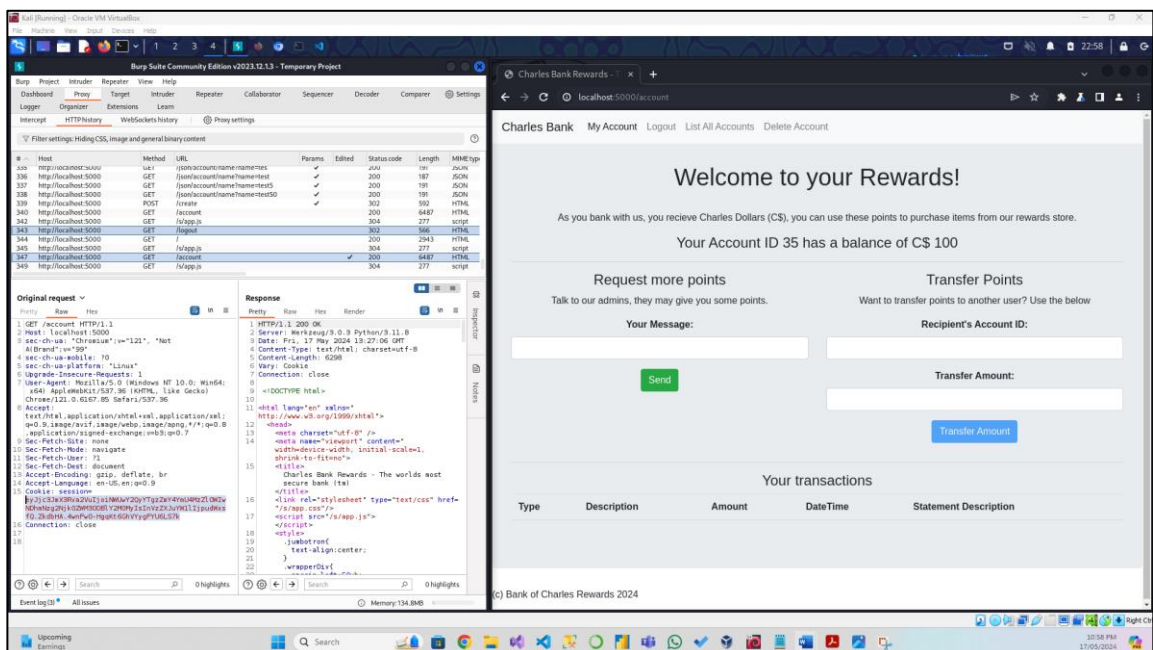
- **Unauthorized Access:** Attackers can access user accounts without permission, leading to potential data theft and unauthorized actions.
 - **Security Breach:** Compromised sessions can lead to further system exploitation and unauthorized access to sensitive information.
-

Evidence:

- **Steps to Reproduce:**
 - **Create an Account or Log In:** Navigate to the account creation or login page and submit the form to log in.
 - **Capture the Session Cookie:** Use Burp Suite to intercept the POST response and capture the session cookie.
-



- **Log Out:** Log out from the application.
- **Reuse the Captured Session Cookie:** Navigate to <http://localhost:5000/account>, intercept the request, replace the current session cookie with the captured one, and forward the request.



- **Access Account Page:** Observe that we can access the account page without needing to log in or authentication.

Recommendation:

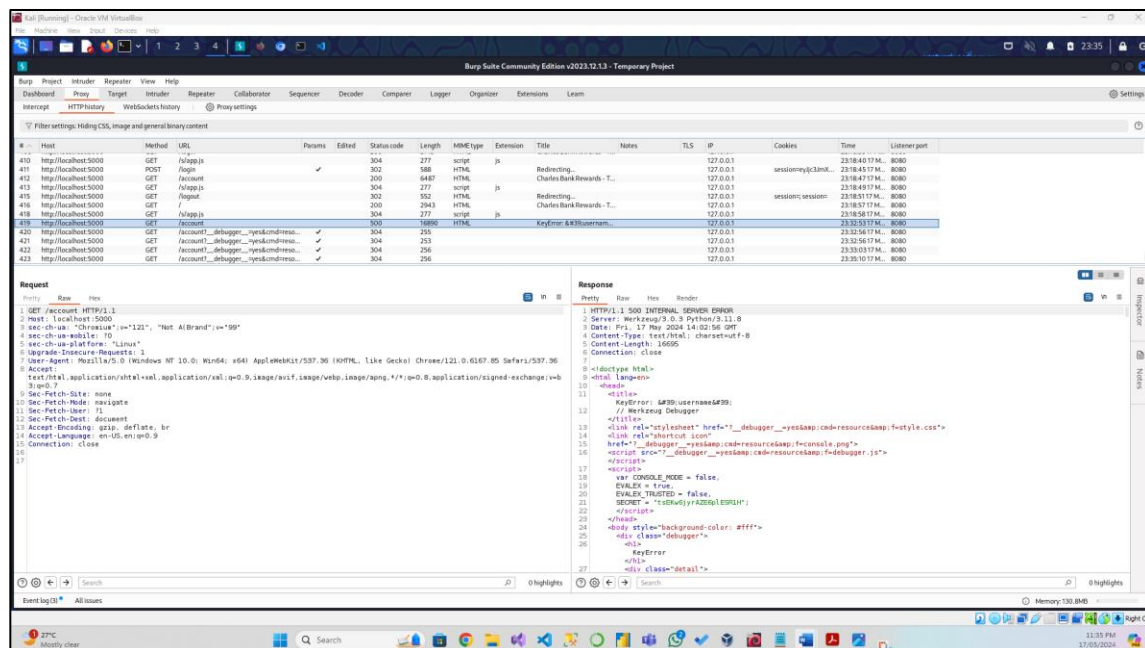
- **Invalidate Session on Logout:** Ensure the session is properly invalidated when the user logs out by clearing the session cookie on the server side.

Actions Taken:

The logout route was updated to clear the session and invalidate the session cookie. This ensures that the session cookie is not reused after logout.

```
56
57 @app.route("/logout")
58 def logout():
59     session.clear() # Clear the session
60     response = make_response(redirect(url_for("base.index")))
61     response.set_cookie('session', '', expires=0) # Invalidate the session cookie
62     return response
63
64
```

- `session.clear()`:
 - Clears all data stored in the session.
 - Removes all session variables when a user logs out, ensuring no session data is left behind.
 - `make_response(redirect(url_for("index")))`:
- Creates a response object that redirects the user to the index page.
 - Uses `make_response` to construct the response.
 - Uses `redirect(url_for("index"))` to ensure the user is redirected to the homepage after logging out.
 - `response.set_cookie('session', '', expires=0)`:
- Invalidates the session cookie.
 - Sets the session cookie to an empty value and its expiration to 0.
 - Instructs the browser to delete the cookie immediately, preventing it from being reused.



From the above screenshot, now if we try to access /account page directly without authentication, its return 500 internal server error.

References

Insecure Login Password Authentication and SQL Injection:

Risk: Critical

Likelihood: Likely

**Consequence:
Catastrophic**

Description:

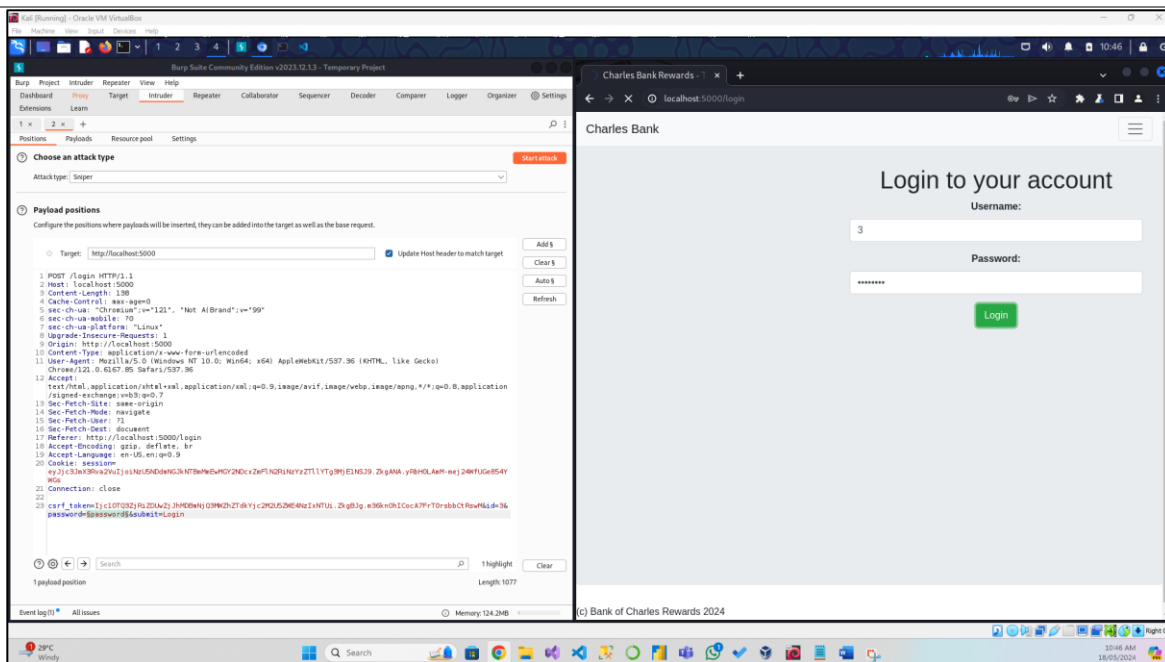
The login form allows users to authenticate with incomplete or single-character passwords due to improper validation logic. Using Burp Suite's Intruder tool, a script containing all numbers, special characters, and alphabets was created and passed into the password field. The response showed that any single character or even just a special character \$ in the password field allows users to log in successfully. Additionally, the user_id parameter in the login query is vulnerable to SQL injection due to improper sanitization.

Impact:

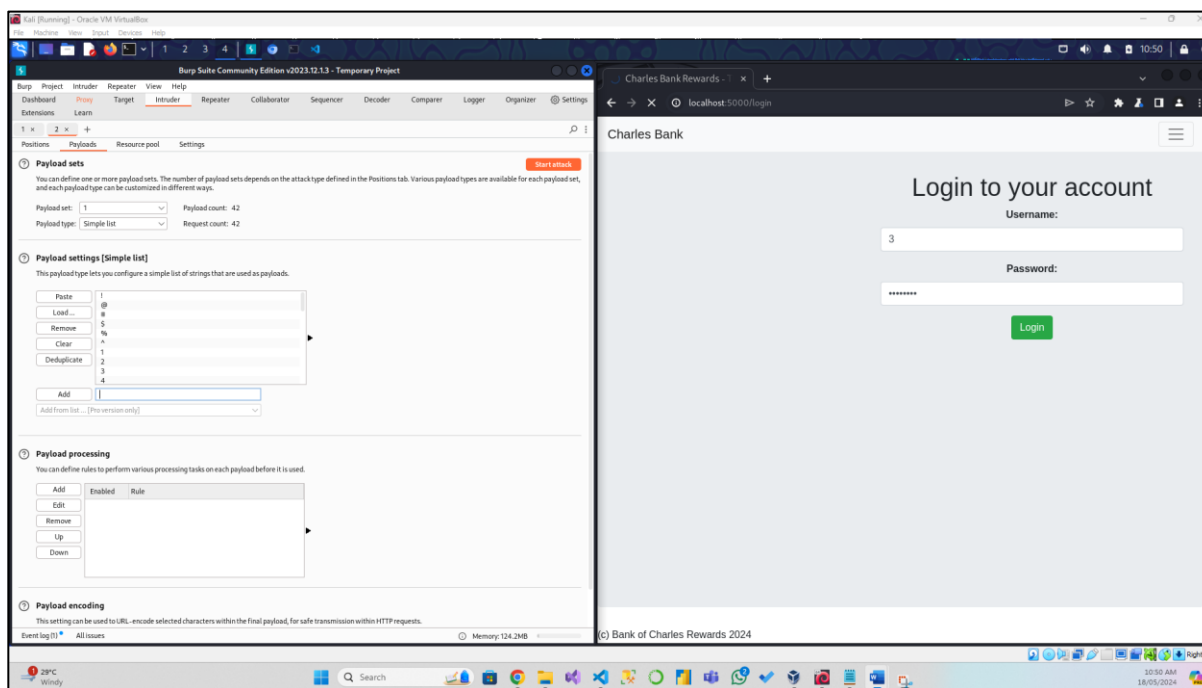
- **Unauthorized Access:** Attackers can easily gain access to user accounts without knowing the full password, leading to potential data theft and unauthorized actions.
 - **Security Breach:** Weak password validation can lead to further exploitation of the system and unauthorized access to sensitive information.
 - **Weak SQL Injection Prevention:** Basic input validation methods can be easily bypassed, potentially allowing more sophisticated SQL injection attacks.
-

Evidence:

- **Navigate to `http://localhost:5000/login`,** enter any valid username and any invalid password, intercept the request with Burp Suite, and observe the username and password in the request.
 - **Set Up Burp Suite Intruder:** Configure Burp Suite Intruder to target the login form.
-



- **Create Payload List:** Generate a payload list containing all numbers, special characters, and some alphabets all together 42 payloads for testing.



- **Run the Attack:** Pass the payloads into the password field and observe the responses.
- **Check Responses:** Notice that any single character or special character “\$” in the password field allows successful login and return status code 302.

References

Insecure Cookie-Based Access Control:

Risk: High

Likelihood: Likely

Consequence: Major

Description:

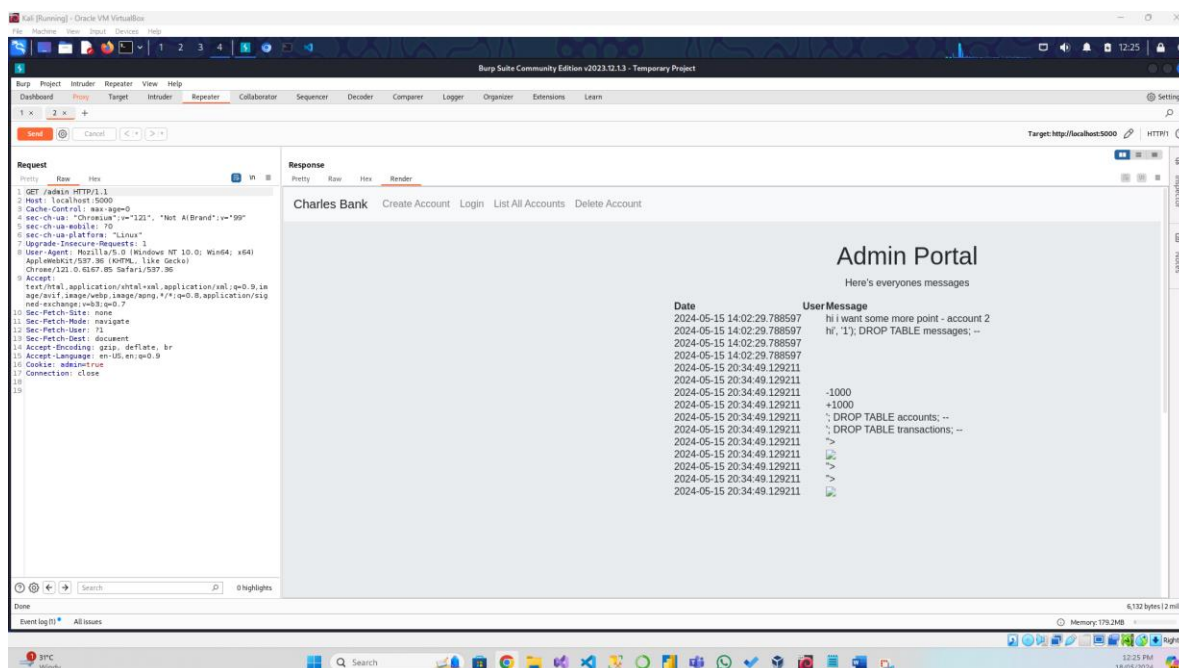
The admin page is accessible by manipulating the cookies. When a user navigates to `http://localhost:5000/admin`, the page checks for an admin cookie. By intercepting the GET request and modifying the cookie value to `admin=true`, any user can gain access to the admin page. This indicates a severe flaw in the access control mechanism, as it relies solely on client-side cookies for authorization.

Impact:

- **Unauthorized Access:** Attackers can gain unauthorized access to the admin page by manipulating cookies, leading to potential data theft, unauthorized actions, and system compromise.
- **Security Breach:** Compromised admin access can result in further exploitation of the system and unauthorized access to sensitive information.

Evidence:

- **Navigate to the Admin Page:** URL: `http://localhost:5000/admin`
- **Intercept the Request:** Use Burp Suite to intercept the GET request to the admin page.
- **Modify the Cookie:** Change the cookie value to `admin=true`.
- **Forward the Request:** Forward the modified request to the server.
- **Access Admin Page:** Observe that the admin page is accessible with the modified cookie.

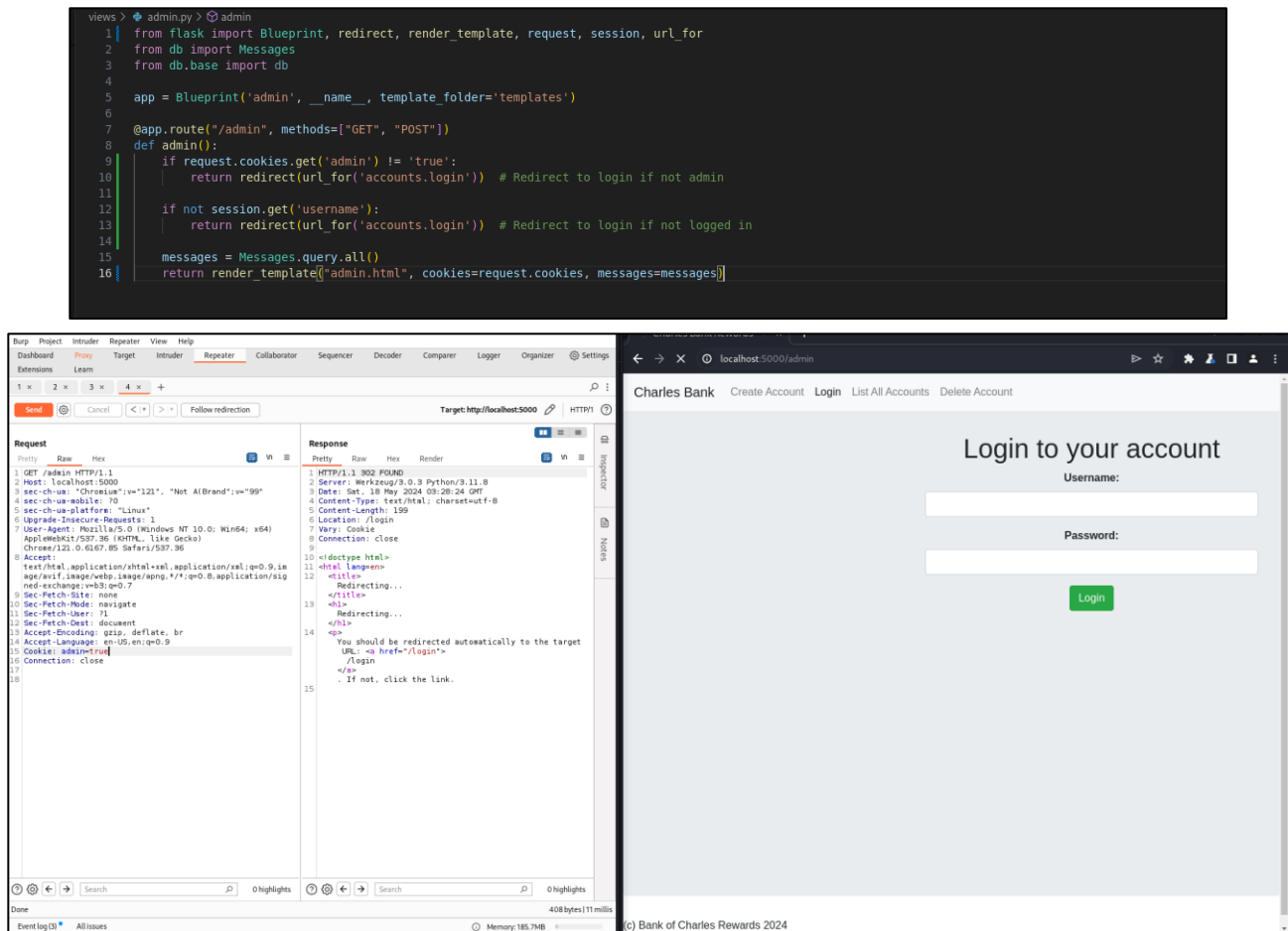


Recommendations:

- **Server-Side Access Control:** Implement proper server-side access control mechanisms to verify admin privileges.
- **Session Management:** Use server-side session management to store and verify user roles securely.

Actions Taken:

- **Check Admin Cookie:** Verifies if the admin cookie is set to true. If not, redirect to the login page.
- **Check User Session:** Checks if the user is logged in by verifying the presence of username in the session. If not, redirect to the login page.



References

Cross-Site Scripting (XSS):

Risk: High

Likelihood: Likely

Consequence: Major

Description:

The application is vulnerable to Cross-Site Scripting (XSS) attacks. After login or creating an account, the user will redirect to /account page. When a user enters a malicious script in the "Request More

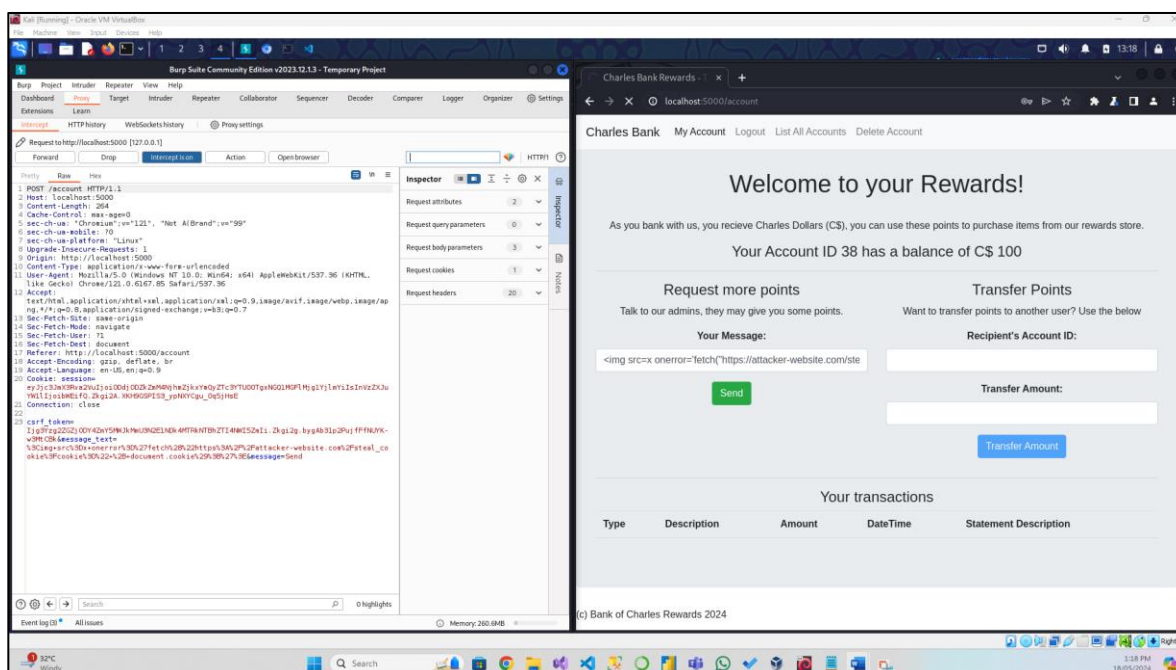
Points" section of the account.html page and sends it to the admin, the script is executed in the admin's browser. This allows an attacker to steal cookies or perform other malicious actions.

Impact:

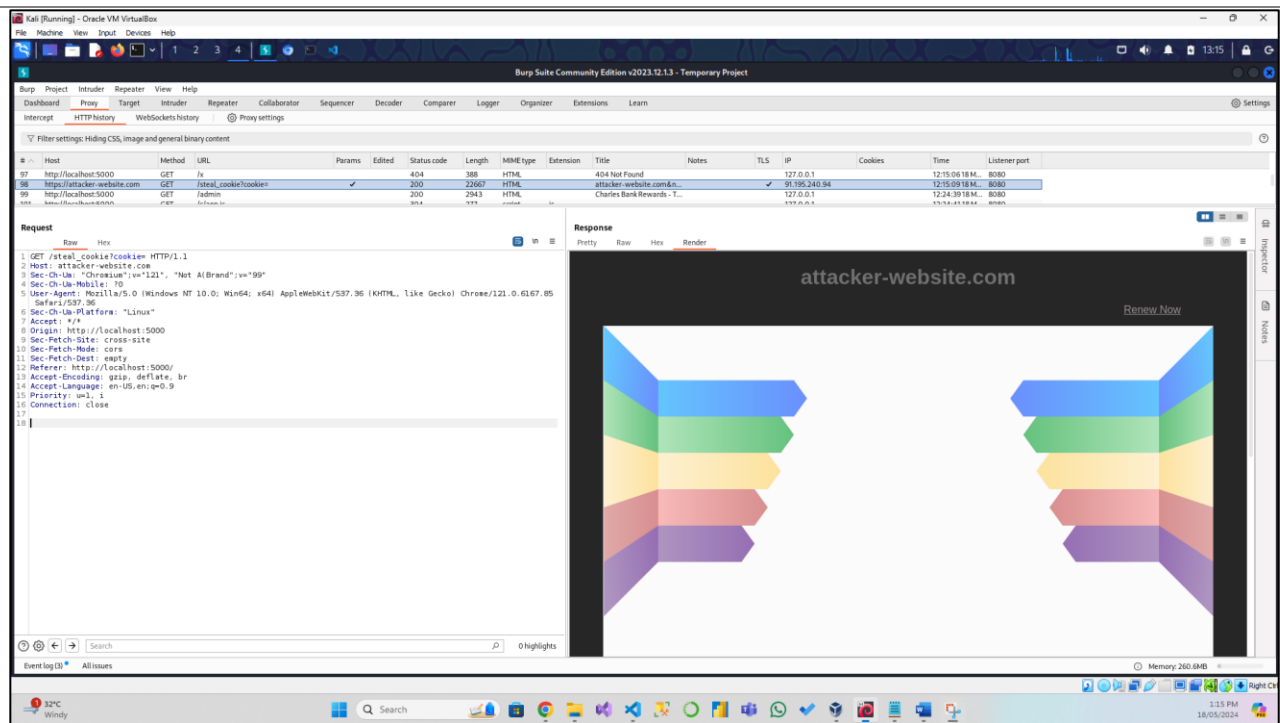
- **Cookie Theft:** Attackers can steal session cookies, allowing them to impersonate users.
- **Unauthorized Actions:** Attackers can perform actions on behalf of the user or admin.
- **Data Theft:** Sensitive information can be stolen.
- **Compromised Accounts:** Users' accounts can be compromised, leading to unauthorized access and actions.

Evidence:

- **Login or Create an Account:** After logging in or creating an account, the user is redirected to the accounts page.
- **Enter Malicious Script in "Request More Points" Section:** In the text field, enter the following script:
- ``



- **Send the Message:** Send the message containing the malicious script.
- **Admin Page Execution:** When the admin views the message, the script is executed, sending the admin's cookies to the attacker's website.



Recommendation:

- Sanitize User Input: Implement input sanitization to remove or escape malicious scripts before storing them in the database.
- Encode Output: Ensure that user-provided data is properly encoded before rendering it in HTML to prevent the execution of scripts.
- Set HTTPOnly Cookies: Configure cookies to be HTTPOnly to prevent JavaScript access to cookies.

Actions Taken:

- Sanitize User Input: Used the bleach library to sanitize user inputs, removing or escaping potentially malicious scripts before storing them in the database.

```
@app.route("/account", methods=["GET", "POST"])
def my_account():
    withdraw_form = WithdrawForm()
    message_form = MessageForm()
    transfer_form = TransferForm()
    if session["username"] is None:
        return render_template("my_account.html")
    user = session["username"]
    account = Account.query.filter_by(name=user).first()
    transactions = Transaction.query.filter_by(account_id=account.id).order_by(
        Transaction.date.desc()
    )

    if message_form.message.data and message_form.validate():
        id = account.id
        message = message_form.message.text.data
        sanitized_message = bleach.clean(message) # Sanitize user input
        messageDb = Messages(account.id, sanitized_message)
        db.session.add(messageDb)
        db.session.commit()
        flash('Message sent successfully!')
        return redirect(url_for("display_my_account"))
```

- Encode Output: Updated the templates to use Jinja2's |e filter to encode user-provided data before rendering it in the HTML.

```

1  {% extends "base.html" %} {% block content %}
2
3  <div class="jumbotron">
4    <div class="wrapperDiv">
5      {% if cookies.admin %}
6      <h1>Admin Portal</h1>
7      <p>Here's everyone's messages</p>
8      <table class="messageTable">
9        <thead>
10         <th>Date</th>
11         <th>User</th>
12         <th>Message</th>
13       </thead>
14       <tbody>
15         {% if messages %}
16         {% for m in messages %}
17         <tr>
18           <td>{{m.date}}</td>
19           <td>{{m.user_id}}</td>
20           <td>{{m.message|e}}</td> <!-- Encode output to prevent XSS -->
21         </tr>
22         {% endfor %}
23         {% else %}
24         <tr>
25           <td>No messages</td>
26         </tr>
27         {% endif %}
28       </tbody>
29     </table>
30     {% else %}
31     <h1>GO AWAY</h1>
32     {% endif %}
33   </div>
34 {% endblock %}

```

- Set HTTPOnly Cookies: Configured the Flask application to use HTTPOnly cookies to prevent JavaScript from accessing session cookies.

```

23 app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///app.db"
24 app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = True
25 app.config["DEBUG"] = True
26 app.config.update({
27     'SESSION_COOKIE_HTTPONLY': True
28 })

```

References

Insecure Authentication Bypass via Cookie Manipulation:

Risk: High

Likelihood: Likely

Consequence: Major

Description:

The account deletion functionality in the web application is vulnerable to a Cookie-Based Password Bypass. By manipulating the session cookie, an attacker can bypass password authentication and delete an account. Additionally, even after an account is marked as deleted, the user can still log in.

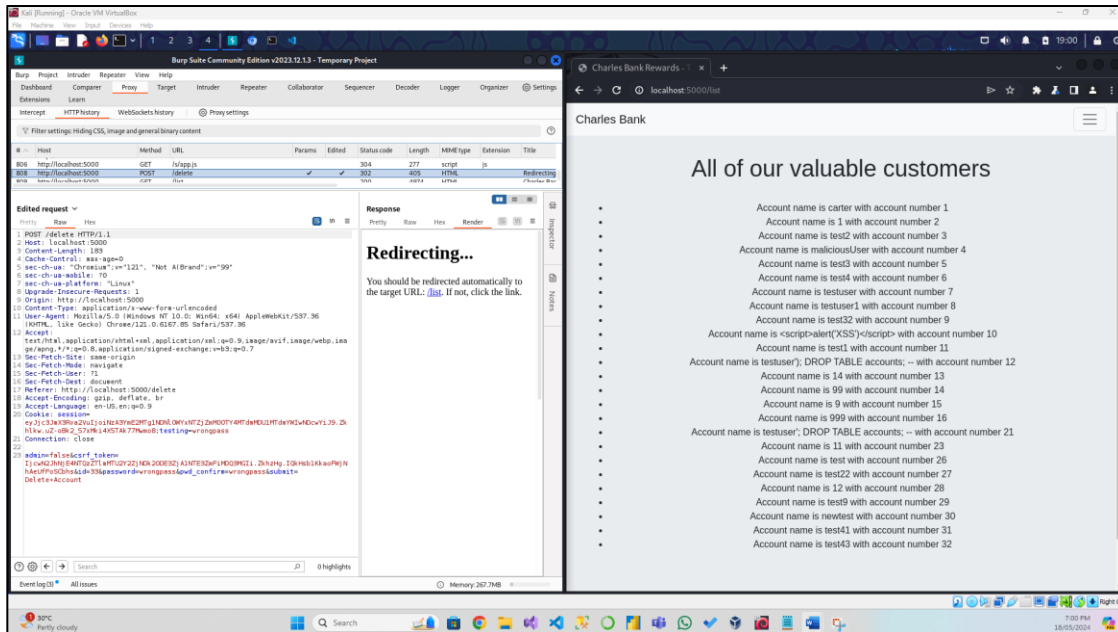
Impact:

- Unauthorized Account Deletion: Attackers could delete user accounts without knowing the correct password by manipulating cookies.
- Incomplete Account Deletion: Even after an account is marked inactive, the user could still log in, leading to inconsistent application state and potential security risks.

Evidence:

Intercept the Delete Account Request: Navigate to the delete account page localhost:5000/delete, fill out the form, and intercept the POST request with Burp Suite.

Send the modified request to the server and Verify the Response. Check the server's response for a successful deletion indication.



Recommendation:

- **Remove Insecure Cookie Handling:** Eliminate the use of cookies for authentication and authorization.
- **Implement Proper Account Status Checks:** Ensure that only active accounts can log in.
- **Complete Account Deletion:** Ensure that deleted accounts cannot log in.

Actions Taken:

- Remove Insecure Cookie Handling:
 - Removed the use of the testing cookie parameter for authentication.
 - Used `check_password_hash` to securely verify passwords.
- Enforce Strong Authentication:
 - Updated the account deletion logic to securely verify passwords.
- Implement Proper Account Status Checks:

- Updated the login logic to ensure only active accounts can log in.
- Complete Account Deletion:
 - Ensured that deleted accounts cannot log in by marking them as inactive and checking their status during login.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user_id = form.id.data
        password = form.password.data

        if "" in user_id:
            return "<h1>STOP TRYING TO HACK US</h1>"

        try:
            id = int(user_id)
        except ValueError:
            return "<h1>Invalid user ID. Must be an integer.</h1>"

        user = None
        with db.get_engine().connect() as con:
            result = con.execute(text("SELECT name, password, active FROM accounts WHERE id = :id LIMIT 1"), {"id": id})
            user = result.fetchone()

        if user and check_password_hash(user[1], password) and user[2]: # Accessing tuple elements by index
            session["username"] = user[0] # Accessing 'id' from the tuple
            flash('Login successful!')
            return redirect(url_for("display.my_account"))
        else:
            flash('Invalid Account ID & Password combination or inactive account.')
            return redirect(url_for('accounts.login'))

    return render_template("login.html", form=form)
```

```
@app.route("/delete", methods=["GET", "POST"])
def delete_account():
    form = DeleteForm()

    if form.validate_on_submit():
        id = form.id.data
        password = form.password.data
        account = Account.query.get(id)

        # Verify password securely
        if account and check_password_hash(account.password, password):
            account.active = False
            db.session.commit()
            flash('Account deleted successfully!')
        else:
            flash('Invalid user ID or password.')

        return redirect(url_for("display.list_accounts"))

    return render_template("delete_account.html", form=form)
```

References

Information Disclosure via Debug Parameter:

Risk: Medium	Likelihood: Possible	Consequence: Moderate
--------------	----------------------	-----------------------

Description:

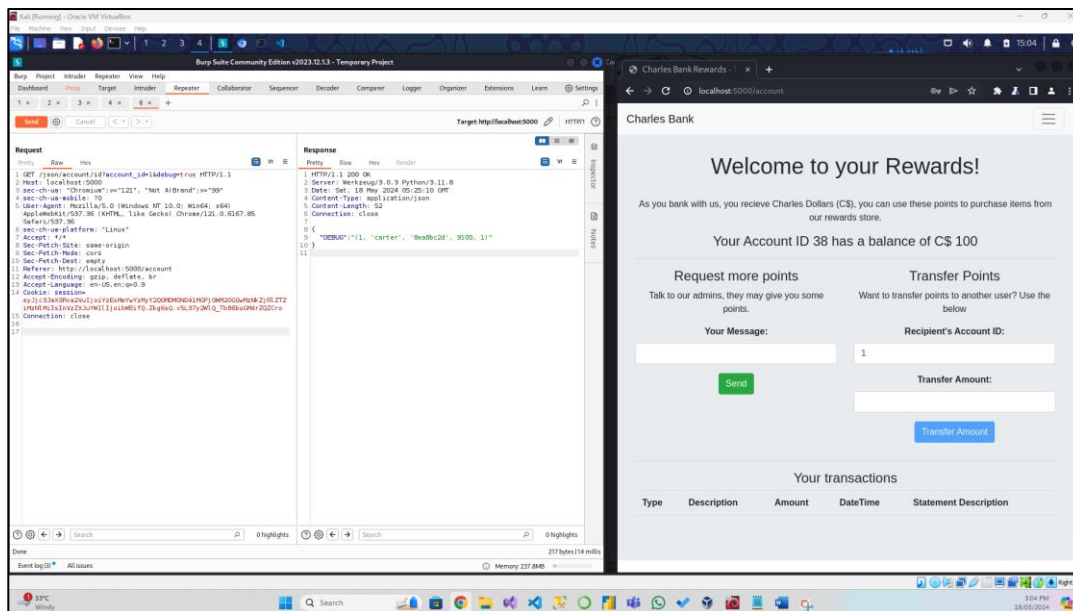
The application's endpoints for checking account validity (/json/account/id and /json/account/name) contain a debug parameter (debug=true). When this parameter is included in the request, the server responds with detailed and confidential information about the user, including their username, account balance, and other sensitive data. This can lead to severe information disclosure if exploited by an attacker.

Impact:

- **Exposure of Sensitive Information:** Attackers can retrieve confidential information about users, such as usernames, account balances, and other details.
- **Privacy Violation:** User privacy is compromised as sensitive data is exposed.
- **Increased Risk of Targeted Attacks:** With access to detailed user information, attackers can perform targeted attacks such as social engineering or identity theft.

Evidence

- **Navigate to Account Page:** After logging in or creating an account, the user is redirected to the /account page.
- **Check Valid Recipient Account ID:** Enter a valid recipient account ID. A GET request is sent to /json/account/id to verify the account ID.
- **Intercept and Modify the Request:** Intercept the request and modify the request to include the debug=true parameter

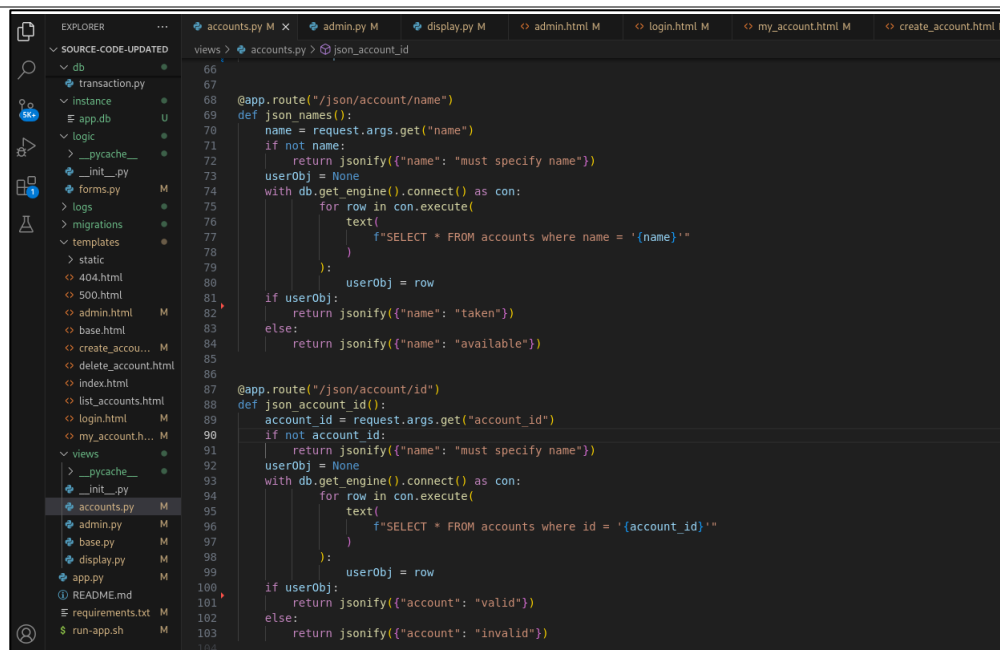


Recommendation:

Remove Debug Functionality in Production: Ensure that any debug-related code is removed or disabled in production environments.

Actions Taken:

- **Remove Debug Functionality:** Removed the debug parameter handling from the /json/account/id and /json/account/name endpoints.



```
EXPLORER
SOURCE-CODE-UPDATED
db
transaction.py
instance
app.db
logic
__pycache__
__init__.py
forms.py
logs
migrations
templates
static
404.html
500.html
admin.html
base.html
create_accou...
delete_account.html
index.html
list_accounts.html
login.html
my_account.h...
views
__pycache__
__init__.py
accounts.py
admin.py
base.py
display.py
app.py
README.md
requirements.txt
run-app.sh

accounts.py X
admin.py M
display.py M
admin.html M
login.html M
my_account.html M
create_account.html M

views >
accounts.py > json_account_id
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104

@app.route("/json/account/name")
def json_names():
    name = request.args.get("name")
    if not name:
        return jsonify({"name": "must specify name"})
    userObj = None
    with db.get_engine().connect() as con:
        for row in con.execute(
            text(
                f"SELECT * FROM accounts where name = '{name}'"
            )
        ):
            userObj = row
    if userObj:
        return jsonify({"name": "taken"})
    else:
        return jsonify({"name": "available"})

@app.route("/json/account/id")
def json_account_id():
    account_id = request.args.get("account_id")
    if not account_id:
        return jsonify({"name": "must specify name"})
    userObj = None
    with db.get_engine().connect() as con:
        for row in con.execute(
            text(
                f"SELECT * FROM accounts where id = '{account_id}'"
            )
        ):
            userObj = row
    if userObj:
        return jsonify({"account": "valid"})
    else:
        return jsonify({"account": "invalid"})
```

References

Command Injection via Query Parameters:

Risk: Critical

Likelihood: Possible

Consequence: Catastrophic

Description:

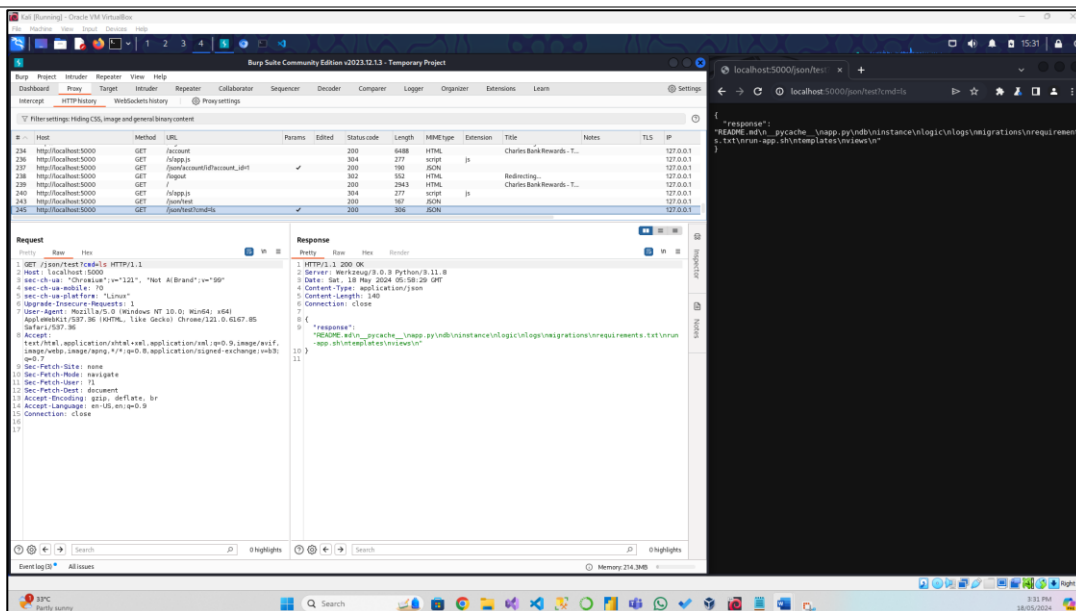
The /json/test endpoint allows an attacker to execute arbitrary system commands by passing them as query parameters. This is possible because the endpoint directly uses the os.popen function with user-provided input without any validation or sanitization. This can lead to unauthorized access, data manipulation, or system compromise.

Impact:

- Unauthorized Access: Attackers can gain unauthorized access to the server's file system and execute arbitrary commands.
- Data Manipulation: Sensitive data can be read, modified, or deleted.
- System Compromise: The entire server can be compromised, leading to potential further exploitation.

Evidence:

- Navigate to the Endpoint: <http://localhost:5000/json/test?cmd=ls>
 - The server responds with a list of files and directories, indicating that the command has been executed.
-



Recommendation:

- Remove Command Execution Functionality: Disable any functionality that allows executing system commands from user inputs.

Actions Taken:

- Removed the functionality that executes system commands from user inputs.

```
base.py - source-code-updated - Visual Studio Code
Run Terminal Help
accounts.py M base.py M admin.py M display.py M admin.html M login.html M my_account.html M create_account.html M app.py M
views > base.py > extra_flag
56
57 @app.route("/json/test")
58 def extra_flag():
59     if not request.args.get("cmd", False):
60         return jsonify({})
61     return jsonify({"response": "Command execution is disabled for security reasons."})
```

References

Appendices

Appendix A – Risk Matrix

		Consequence				
		<i>Insignificant</i>	<i>Minor</i>	<i>Moderate</i>	<i>Major</i>	<i>Catastrophic</i>
Likelihood	<i>Very Likely</i>	Low	Medium	High	Critical	Critical
	<i>Likely</i>	Low	Low	Medium	High	Critical
	<i>Possible</i>	Informational	Low	Medium	High	High
	<i>Unlikely</i>	Informational	Low	Low	Medium	High
	<i>Very Unlikely</i>	Informational	Informational	Low	Medium	Medium