NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4237 Software Security and Data Privacy

Exercise 3. Mitigating Vulnerabilities

Group (number)

Author(s):
Stefano Iannello
Eleni Mandana

Table of Contents

List of Figures

1 Introduction

The purpose of this report is to explain the fixes we made on the preset of the pre-identified vulnerabilities in the SecureHelp application. We tried to solve as many as possible with state-of-the-art technologies.

We started by specifying a mitigation strategy for every vulnerability in order to obtain a plan. We first solved the vulnerabilities that we were more familiar with from Secure Code Warrior [1]. For the more complex ones we had to gather up as a group.

As one can observe from the contents, the flow of the report is determined by the alphabetical order of the OWASP [2] category and by the number of it. In each section we shortly explain the severity of the vulnerability, a strategy to solve it, the part of code we changed in order to fix it and some proof that the vulnerability is fixed.

The following section is a brief introduction to the authors of this report.

1.1 About the authors

Stefano Iannello: I am currently a first-year Master's student in Computer Science. During my bachelor studies, my academic focus was on Object-Oriented Programming (OOP) using the Java language. I am excited to share that this is my first experience in the realm of software security, which I find to be both engaging and challenging.

Eleni Mandana: I am a final-year computer science student at Aristotle University of Thessaloniki (AUTh) with a specialization in Data and Web Management. My primary interests lie in machine learning, information retrieval, and database technology. Throughout my academic career, I have been highly motivated to explore and develop my expertise in these areas.

Konstantinos Giotakos: I am a Computer Science Student at AUTh, with no specialization yet. I am interested in Artificial Intelligence and Machine Learning.

2 WSTG-ATHZ-02

The application allowed normal users to approve certifications. In this section, we discuss how we solved this vulnerability.

2.1 Mitigation strategy

A possible fix is to check the user's permissions before interacting with the certification requests.

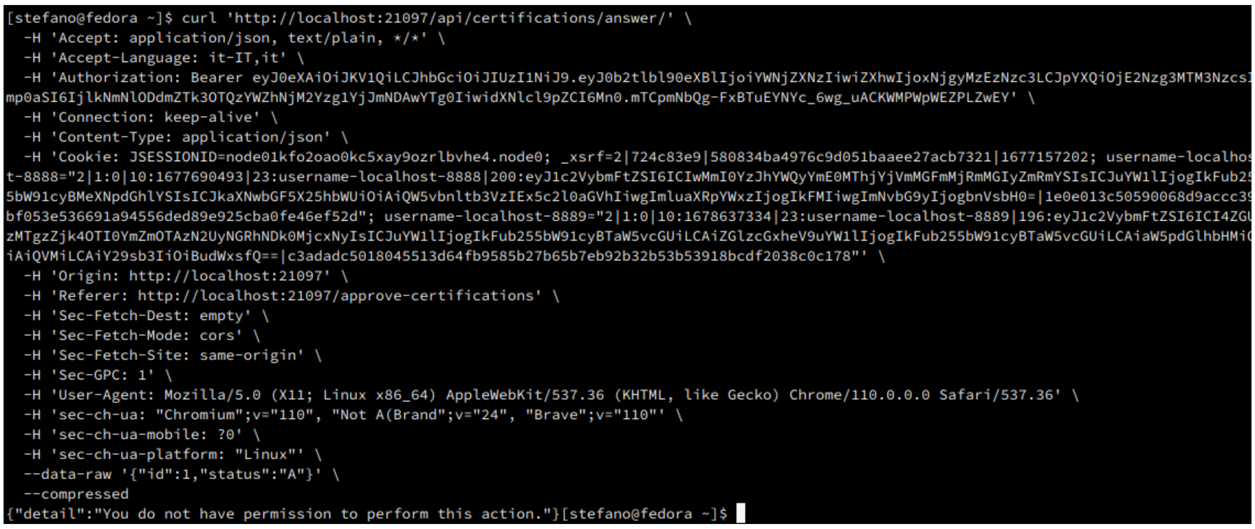
2.2 Code change

```python
# backend/apps/certifications/views.py

class AnswerCertificationRequest(generics.GenericAPIView):
    """View to answer certification requests"""
    #permission_classes = [permissions.IsAuthenticated]
    permission_classes = [permissions.IsAdminUser]

    def post(self, request):
        if self.request.user.is_staff:
            # check if id and status is provided
        ...
```

With this solution, when an attacker tries to exploit this vulnerability, a "missing authorization" error message appears



```
[stefano@fedora ~]$ curl 'http://localhost:21097/api/certifications/answer/' \
  -H 'Accept: application/json, text/plain, */*' \
  -H 'Accept-Language: it-IT,it' \
  -H 'Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNjgyMzEzNzc3LCJpYXQiOjE2Nzg3MTM3NzcsI
mp0aSI6IjlkNmNlODdmZTk3OTQzYWZhNjM2Yzg1YjJmNDAwYTg0IiwidXNlcl9pZCI6Mn0.mTCpmNbQg-FxBTuEYNYc_6wg_uACKWMPWpWEZPLZwEY' \
  -H 'Connection: keep-alive' \
  -H 'Content-Type: application/json' \
  -H 'Cookie: JSESSIONID=node01kfo2oao0kc5xay9ozrlbvhe4.node0; _xsrf=2|724c83e9|580834ba4976c9d051baaee27acb7321|1677157202; username-localhos
t-8888="2|1:0|10:1677690493|23:username-localhost-8888|200:eyJ1c2VybmFtZSI6ICIwMmI0YzJhYWQyYmE0MThjYjVmMGFmMjRmMGIyZmRmYSIsICJuYW1lIjogIkFub25
5bW91cyBMeXNpdGhlYSIsICJkaXNwbGF5X25hbWUiOiAiQW5vbnltb3VzIEx5c2l0aGVhIiwgImluaXRpYWxzIjogIkFMIiwgImNvbG9yIjogbnVsbH0=|1e0e013c50590068d9accc39
bf053e536691a94556ded89e925cba0fe46ef52d"; username-localhost-8889="2|1:0|10:1678637334|23:username-localhost-8889|196:eyJ1c2VybmFtZSI6ICI4ZGU
zMTgzZjk4OTI0YmZmOTAzN2UyNGRhNDk0MjcxNyIsICJuYW1lIjogIkFub255bW91cyBTaW5vcGUiLCAiZGlzcGxheV9uYW1lIjogIkFub255bW91cyBTaW5vcGUiLCAiaW5pdGlhbHMiO
iAiQVMiLCAiY29sb3IiOiBudWxsfQ==|c3adadc5018045513d64fb9585b27b65b7eb92b32b53b53918bcdf2038c0c178"' \
  -H 'Origin: http://localhost:21097' \
  -H 'Referer: http://localhost:21097/approve-certifications' \
  -H 'Sec-Fetch-Dest: empty' \
  -H 'Sec-Fetch-Mode: cors' \
  -H 'Sec-Fetch-Site: same-origin' \
  -H 'Sec-GPC: 1' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.0.0 Safari/537.36' \
  -H 'sec-ch-ua: "Chromium";v="110", "Not A(Brand";v="24", "Brave";v="110"' \
  -H 'sec-ch-ua-mobile: ?0' \
  -H 'sec-ch-ua-platform: "Linux"' \
  --data-raw '{"id":1,"status":"A"}' \
  --compressed
{"detail":"You do not have permission to perform this action."}[stefano@fedora ~]$
```

Figure 1: Not authorized message.

3 WSTG-ATHN-03

The vulnerability we are called to solve is about the unlimited times a user can attempt to login into his account. A crucial problem that needs to be solved because it makes the app vulnerable to brute force password cracking attacks.

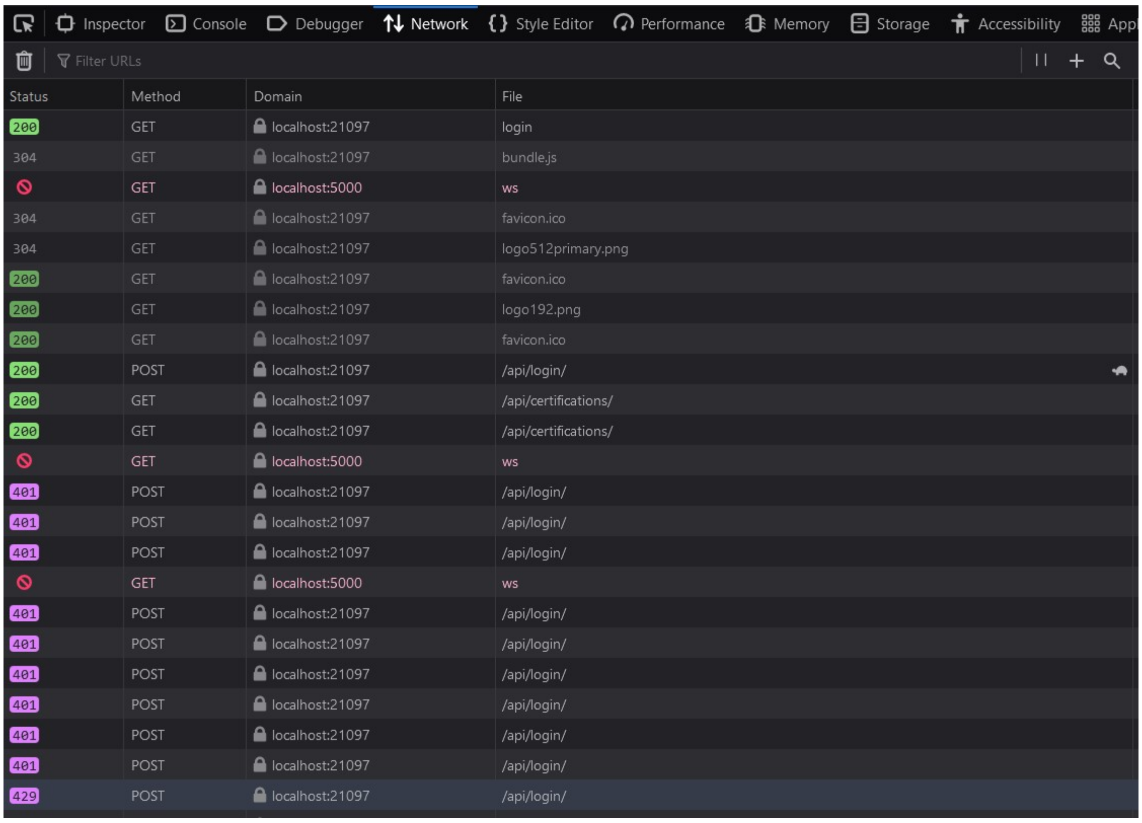
3.1 Mitigation strategy

Our approach was to solve it with DJANGO's Throttling feature [3]. We decided to block more than 10 requests per day by those who are registered as users to use the website. This blocks them by both sending login requests but also from creating new accounts. Registered users of the website have no problem using the website.

3.2 Code change

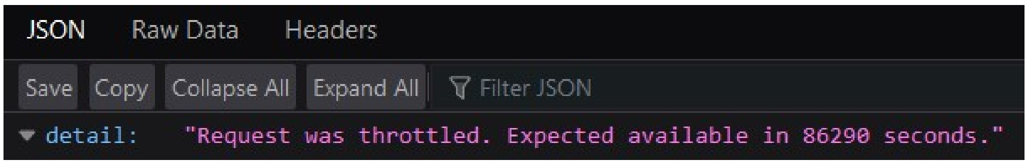
```
# backend\securehelp\settings.py

REST_FRAMEWORK = { # code
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '10/day',
    }
}
```



Status	Method	Domain	File
200	GET	localhost:21097	login
304	GET	localhost:21097	bundle.js
🚫	GET	localhost:5000	ws
304	GET	localhost:21097	favicon.ico
304	GET	localhost:21097	logo512primary.png
200	GET	localhost:21097	favicon.ico
200	GET	localhost:21097	logo192.png
200	GET	localhost:21097	favicon.ico
200	POST	localhost:21097	/api/login/
200	GET	localhost:21097	/api/certifications/
200	GET	localhost:21097	/api/certifications/
🚫	GET	localhost:5000	ws
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
🚫	GET	localhost:5000	ws
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
401	POST	localhost:21097	/api/login/
429	POST	localhost:21097	/api/login/

Figure 2: Blocked requests after 10 attempts.



```
detail:    "Request was throttled. Expected available in 86290 seconds."
```

Figure 3: Error 429 Details.

As we can see if a user tries to make more than 10 failed login attempts will return error 429 [4]. It does not protect the server from DoS attacks although it forbids brute-force password cracking.

4 WSTG-ATHN-01 / WSTG-CRYP-03

Sensitive information is sent via unencrypted channels [5]. Unencrypted channels refer to communication channels that do not use any encryption methods to protect the information being transmitted and could potentially put the information at risk of being intercepted and accessed by unauthorized individuals.

4.1 Mitigation strategy

To solve this vulnerability the HTTP protocol must be changed to HTTPS [6], email verification link must be sent using TLS [7].

4.2 Code change

```
# /securehelp/.env

GROUP_ID=97
PORT_PREFIX=210
DOMAIN=localhost
PRODUCTION=True
PROTOCOL=https

DJANGO_SUPERUSER_PASSWORD=password1
DJANGO_SUPERUSER_USERNAME=admin1
DJANGO_SUPERUSER_EMAIL=admin@mail.com

# backend/securehelp/settings.py
EMAIL_BACKEND = "django.core.mail.backends.smtp.EmailBackend"
EMAIL_HOST = "smtp.stud.ntnu.no"
EMAIL_USE_TLS = True
EMAIL_PORT = 25
DEFAULT_FROM_EMAIL = "tdt4237-group" + GROUP_ID + " " + "<noreply@idi.ntnu.no>"
```

After generating (self-signed) SSL certificates and setting up the nginx configuration, HTTPS is now used as the transmission protocol.

5 WSTG-CRYP-04

The passwords in the application were hashed with the SHA-1 algorithm, which is not recommended for this use [8].

5.1 Mitigation strategy

In order to store passwords securely we need to change the password hashers. We used the PBKDF2 algorithm [9] which is recommended and provided by Django framework [10].

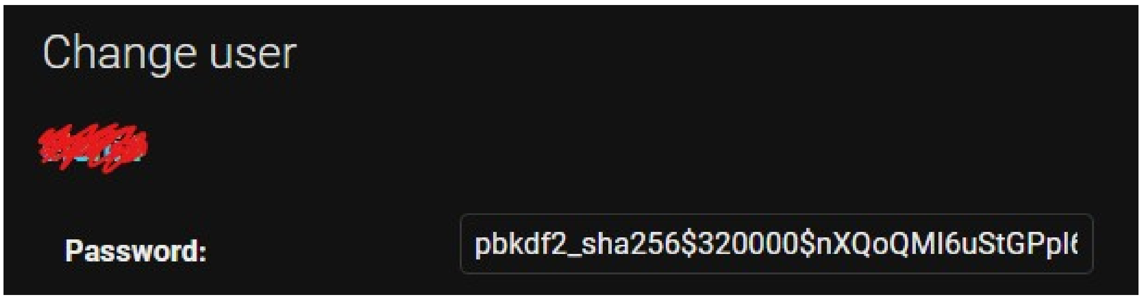
5.2 Code change

In order to change the password hasher we simply needed to add the desired one first in the PASSWORD_HASHERS section.

```
# backend\securehelp\settings.py

PASSWORD_HASHERS = [
# 'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher',

'django.contrib.auth.hashers.PBKDF2PasswordHasher',
'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
'django.contrib.auth.hashers.Argon2PasswordHasher',
'django.contrib.auth.hashers.ScryptPasswordHasher',
]
```


Change user

[redacted]

Password: pbkdf2_sha256$320000$nXQoQMI6uStGPpl

Figure 4: Blocked requests after 10 attempts.

The passwords that are stored from now on are hashed by PBKDF2.

6 WSTG-CONF-12

By default, if a CSP policy is not specified, the browser will allow content from any source to be executed on the page [11].

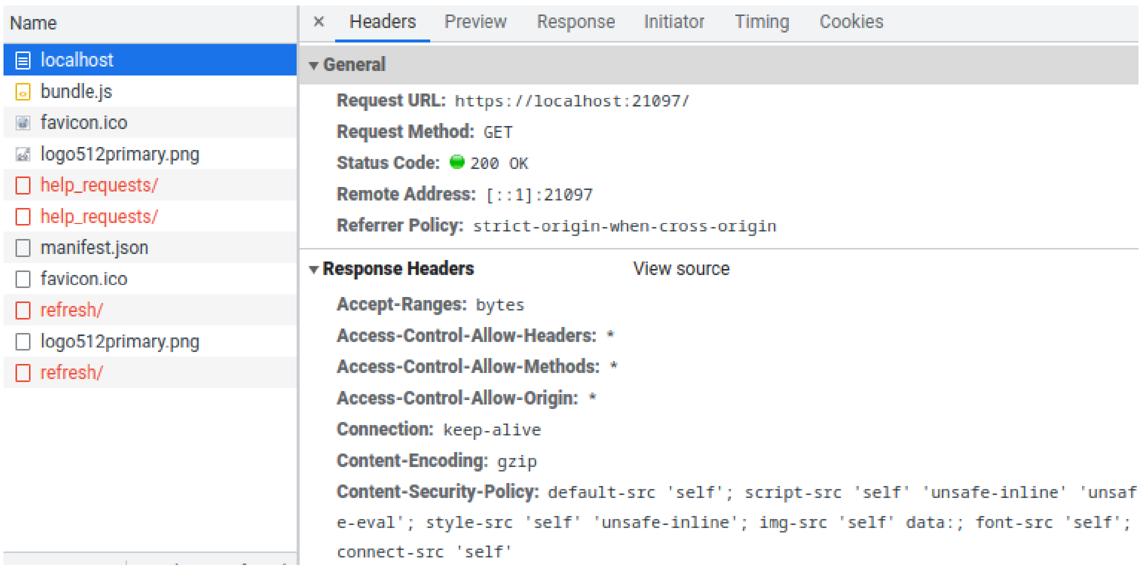
6.1 Mitigation strategy

A possible fix for this problem is to use set the default source to be our own domain, by modifying the nginx.conf file.

6.2 Code change

```
# nginx/nginx.conf

server {
    listen      80 ssl;
    server_name localhost;
    ssl_certificate   server.crt;
    ssl_certificate_key server.key;
    add_header X-Content-Type-Options nosniff;
    add_header Content-Security-Policy "default-src 'self'; script-src 'self'
        'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline'; img-src
        'self' data:; font-src 'self'; connect-src 'self' wss://localhost:5000/ws
        https://molde.idi.ntnu.no:21097/;";
```



Request URL: https://localhost:21097/
Request Method: GET
Status Code: ● 200 OK
Remote Address: [::1]:21097
Referrer Policy: strict-origin-when-cross-origin

Accept-Ranges: bytes
Access-Control-Allow-Headers: *
Access-Control-Allow-Methods: *
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Encoding: gzip
Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline'; img-src 'self' data:; font-src 'self'; connect-src 'self'

Figure 5: CSP header.

7 WSTG-INPV-05

The application was vulnerable to SQL injections, the 'id' variable is directly inserted into the SQL query without any validation or sanitization. In order to keep the integrity of the web server and also protect the application from injections, every input should be sanitized.

7.1 Mitigation strategy

A possible fix for this problem is to use Django's built-in escape function from django.utils.html [12] to escape sanitize the input when checking for the id of the request [13].

7.2 Code change

```
# backend/apps/help_requests/views.py

def post(self, request):
        # check if id is provided
        if not(request.data.get('request_id')):
            return Response({'error': 'id is required'},
                 status=status.HTTP_400_BAD_REQUEST)

        # check if id is valid
        try:
            #rId = base64.b64decode(request.data.get('request_id'))
            #rId = pickle.loads(rId)
            rId = escape(request.data.get('request_id'))
            help_request = HelpRequest.objects.raw(
                "SELECT * FROM help_requests_helprequest WHERE id = %s", [rId])[0]
```

When the attacker tries to exploit the vulnerability with the line

```
{"request\_id":"1' or 1=1; --"}
```

in the payload, an error is sent as a response.

8 WSTG-INPV-02

The HelpRequest info input field allows users to input information regarding their request for help. However, this input field does not have any mechanism in place to sanitize the input from the user. This makes it vulnerable to various injection attacks, where an attacker can inject malicious code or characters into the input field.

Input sanitization is a crucial aspect of application security as it helps prevent a wide range of attacks such as injection attacks, cross-site scripting (XSS) attacks, and many more. Input sanitization involves cleaning or filtering the user's input data to ensure that it does not contain any malicious code or characters that can be used to exploit the application [13].

We are called to take measures in order to mitigate this vulnerability.

8.1 Mitigation strategy

A possible fix for this problem is to use DOMPurify [14] library to escape the help request description input. We chose to escape the output instead to sanitize the input because is simpler and more effective at preventing attacks such as SQL injection and cross-site scripting [15].

8.2 Code change

```
# frontend/src/components/HelpRequest.jsx

    <div
      dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(helpRequest.description) }}
    ></div>
```

The help request section of the application is now secured from XSS attacks.

9 WSTG-SESS-01

When a user registers in the application, a verification process takes place. During that process, the email verification link has no expiration time. This increases the risk of unauthorized access to the user's account [16].

9.1 Mitigation strategy

A possible fix for this problem is to set a ”deadline timestamp” when the verification link is sent. When the user verifies the email, the current timestamp is checked, if the current timestamp is greater than the deadline timestamp, the user is not validated.

9.2 Code change

```
# backend/apps/users/serializers.py
   def create(self, validated_data):
        ...
        # create email to send to user
        ...
        # Set expiration time for the verification link
        expiration_time = now() + timedelta(minutes=30)

        url = f"{settings.PROTOCOL}://{domain}{link}?
        token={secrets.token_urlsafe()}/?exp={expiration_time.timestamp()}"
        ...

# backend/apps/users/views.py
class VerificationView(generics.GenericAPIView):
        ...
        # Check if the user exists and the link hasn't expired
        try:
            expiration_time = request.GET.get('exp')
            expiration_time = datetime.fromtimestamp(float(expiration_time))
            expiration_str = expiration_time.strftime('%Y-%m-%d %H:%M:%S')
            current_time = datetime.now()
            current_str = current_time.strftime('%Y-%m-%d %H:%M:%S')
            print("Current time: ", current_str)

            if current_time < expiration_time:
                user.is_active = True # Activate user
                user.save()
                return redirect(verified_url)
            else:
                return redirect(invalid_url)
            ...
```

10 WSTG-SESS-01

The application creates email verification links encoding the user's name. It makes it easier for hackers or malicious actors to guess or brute-force the link and gain access to the user's account. [17]

10.1 Mitigation strategy

There are multiple steps to strengthen weak email verification links. Firstly, a protected channel for transmission (TLS) must be used. Secondly, the token should be unique and unpredictable, this can be obtained using a secure random generator (Django secrets). Finally, a time to live should be taken into consideration for the verification link [17].

In this section, we focus on creating an unpredictable token by the Secrets module, and token_urlsafe function. The reason we trust Secrets is that it uses CSPRNG to create the token. [18]

10.2 Code change

```
# backend/apps/users/serializers.py - line 106

url = f"{settings.PROTOCOL}://{domain}{link}?
token={secrets.token_urlsafe()}/?exp={expiration_time.timestamp()}"
```

11 WSTG-SESS-06

The vulnerability to solve in this section is that access tokens are deleted only on the client side. Tokens should be deleted also from the backend.

11.1 Mitigation strategy

A possible fix for this problem is to send a request to the server using Django's authentication system to invalidate the session of the client when it signs out [19]. Furthermore, tokens' lifetime should be set to reasonable values.

11.2 Code change

```
# backend/apps/users/views.py

    def logout_view(request):
        logout(request)
        return redirect("/login")

#frontend/src/App.jsx

  const signOut = () => {
    fetch('/logout/')
    .then(response => console.log("User logged out"));
    localStorage.removeItem("user");
    localStorage.removeItem("access_token");
    localStorage.removeItem("refresh_token");
    setUser(null);
  };

# backend/securehelp/settings.py

SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
    'REFRESH_TOKEN_LIFETIME': timedelta(minutes=60),
    'ROTATE_REFRESH_TOKENS': True,
}
```

12 Conclusion

We managed to solve most vulnerabilities and although it seems like a significant achievement, it is important to acknowledge that this alone does not necessarily make the app completely secure. We identified some crucial vulnerabilities in the previous exercise which remain unsolved.

Something that we learned from this assignment is that fixing is not always the cure, and just like medicine prevention is the key in some situations.

References

[1] *Secure Code Warrior.* URL: https://portal.securecodewarrior.com/#/intro-splash.

[2] *OWASP Web Security Testing Guide.* URL: https://owasp.org/www-project-web-security-testing-guide/v42/.

[3] *Django Throttling.* URL: https://www.django-rest-framework.org/api-guide/throttling/.

[4] *What Does HTTP Error 429: Too Many Requests Mean? How to Fix It.* URL: https://blog.hubspot.com/website/http-error-429.

[5] *Is HTTP Secure? and Is It Really Safe to Visit HTTP Sites?* URL: https://technocript.com/is-http-secure/.

[6] *Introduction to HTTPS.* URL: https://https.cio.gov/faq/.

[7] *What exactly is TLS when it comes to email Encryption?* URL: https://www.agari.com/blog/transport-layer-security-tls-emailencryption.

[8] *Is SHA-1 secure for password storage?* URL: https://stackoverflow.com/questions/2772014/is-sha-1-secure-for-password-storage.

[9] *Secure password storage with PBKDF2.* URL: https://dewni-matheesha.medium.com/secure-password-storage-with-pbkdf2-33c6d51a3bb2.

[10] *Password management in Django.* URL: https://www.w3cschool.cn/doc_django_1_8/django_1_8-topics-auth-passwords.html.

[11] *Content Security Policy (CSP) Header Not Set.* URL: https://scanrepeat.com/web-security-knowledge-base/content-security-policy-csp-header-not-set.

[12] *Django Documentation — Django Utils.* URL: https://docs.djangoproject.com/en/4.1/ref/utils/#module-django.utils.html.

[13] *What is Input Sanitization?* URL: https://www.webopedia.com/definitions/input-sanitization/.

[14] *DOMPurify.* URL: https://github.com/cure53/DOMPurify.

[15] *Don't try to sanitize. Escape Output.* URL: https://benhoyt.com/writings/dont-sanitize-do-escape/.

[16] *Should email Verification Links Expire?* URL: https://ux.stackexchange.com/questions/33014/should-email-verification-links-expire.

[17] *Implementing the right Email Verification flow.* URL: https://supertokens.com/blog/implementing-the-right-email-verification-flow.

[18] *Python secrets module for strong random number and token generation.* URL: https://pythonsimplified.com/python-secrets-module-for-strong-random-number-and-token-generation/.

[19] *How to delete the Token from backend in Django.* URL: https://stackoverflow.com/questions/64822888/how-to-delete-the-token-from-backend-in-django.