NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4237 Software Security and Data Privacy

Exercise 2. Mitigating Vulnerabilities

Group 100

Author(s):
Ana Barrera Novas
Andrea Cicinelli
Nicola Katja Gisela Ferger

Table of Contents

1 Introduction

The application SecFit has several vulnerabilities that can be exploited by attackers, making it unsafe. This work explains how to mitigate some of these vulnerabilities and make the application more secure.

In this report, we focus on 11 vulnerabilities. For each vulnerability, we follow the same structure: Firstly, the vulnerability is briefly explained. We then explain on an abstract level how we went about mitigating that vulnerability. After that, we discuss the specific changes made in the code. When we refer to commit codes, we mean commits made to this repository: `https://git.ntnu.no/TDT4237-2025/group-100`.

2 WSTG-CRYP-04 - Insecure password hasher

The previous password hasher was unsalted SHA-1 which is insecure and crackable. This means, it should not be used in practice.

2.1 Mitigation strategy

To mitigate this vulnerability, we instead use the password hasher Argon2, developed by Biryukov et al. [1]. This solves the problem since this password hasher was found to be the most secure one during the Password Hashing Competition 2015 [2]. Because of that it is also one of the hashers recommended to be used in the Django documentation, Version 5.1 [3].

2.2 Code change

There were not many changes to be made in the code. The mitigation of this vulnerability was done in commit `ac86959` in branch `WSTG-CRYP-04`.

In `/backend/secfit/settings.py`, `PASSWORD_HASHERS` was modified to list Argon2 first, therefore ensuring that it is used instead of unsalted SHA-1. Furthermore, since Argon2 is not included in Django by default, it was added to `/backend/requirements.txt`.

3 WSTG-SESS-06 - Session tokens are not deleted/blacklisted after user logs out

The access and refresh tokens are only deleted client-side, so if an attacker accesses the refresh token they can use it even after the user has logged out.

3.1 Mitigation strategy

To solve this problem, we have to ensure that the refresh token is invalidated in the server when logging out. This should be done by calling an endpoint in the backend that invalidates the token by blacklisting before deleting them from the client local storage.

3.2 Code change

At first we tried to create a new endpoint for the logout that had the procedure to invalidate the token, but then we realized that there was already an endpoint for this that calls the TokenBlack-

listView which is already designed to manage the invalidation of refresh tokens using Django REST Framework.

So for the changes themselves:
1. Call the endpoint before deleting token from local storage

```
    #frontend/components/AuthContext.jsx

    const logout = () => {
    const accessToken = SessionService.getLocalAccessToken();
    const refreshToken = SessionService.getLocalRefreshToken();

    // Invalidate token in server
    AxiosInstance.post("/api/logout/", {refresh: refreshToken })
      .then(() => {
        console.log("Tokens invalidated on the server");
      })
      .catch((error) => {
        console.error("Error invalidating tokens on the server", error);
      })
      .finally(() => {
      // Erase token from local storage
      SessionService.removeLocalAccessToken();
      SessionService.removeLocalRefreshToken();
      SessionService.removeUserId();
      SessionService.removeUserName();
      SessionService.removeIsCoach();

      setIsAuthenticated(false);
    });
  };
```

2. Ensure endpoint is calling the right view

```
#backend/users/urls.py

    path("api/logout/", TokenBlacklistView.as_view(), name="token_blacklist"),
```

3. Ensure blacklist configuration

```
#backend/secfit/settings.py

    SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(hours=72),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=60),
    'ROTATE_REFRESH_TOKENS': True,
    "BLACKLIST_AFTER_ROTATION": True,
    "AUTH_TOKEN_CLASSES": ("rest_framework_simplejwt.tokens.AccessToken",),
    "TOKEN_BLACKLIST_ENABLED": True,
}
```

4 WSTG-SESS-07 - Session tokens timeout is too long

4.1 Mitigation strategy

The access token and the refresh token expires in a very long time. This is dangerous in case an exploiter can get access to them. The longer they lasts the longer is the time for the exploiter to

get access to the account.
Obvsiously if the given time is to low the user experience will be effected.
The optimal time, usually, is fo the access token to be between 1 and 15 minutes.
For the refresh token around 15 days.

4.2 Code change

In backend/secfit/settings.py

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(hours=72),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=60),
    'ROTATE_REFRESH_TOKENS': True,
}
```

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=15),
    'ROTATE_REFRESH_TOKENS': True,
}
```

We just changed the time from 72hours to 15 minutes for the access token and from 60 days to 15 days for the refresh token.

5 WSTG-INPV-05 - Input field vulnerable for SQL Injection

In the original application, we were able to insert SQL queries in an input section, and an attacker could exploit this to obtain information stored in the related database.

5.1 Mitigation strategy

Before, the code used string interpolation with the "username" parameter, which was the direct cause of the vulnerability. To prevent this, it should be used a parameterized query.

After applying the change (explained below), we made the black-box testing again and checked that now the SQL Injection wasn't possible.

5.2 Code change

The changes were made in commit `567f9dd`.

```
#/backend/users/views.py line 80

query = "SELECT * FROM users_user WHERE username = %s"
with connection.cursor() as cursor:
    cursor.execute(query, [username])
    columns = [col[0] for col in cursor.description]
    rows = cursor.fetchall()
```

6 WSTG-ATHZ-02 - User is able to bypass authorization schema

In this vulnerability, a user that is not the recipient of an athlete-to-coach offer can accept it or decline on behalf of others, which could be used for malicious purposes.

6.1 Mitigation strategy

To solve this vulnerability, the procedure is very straightforward. There needs to be an explicit verification to ensure that the authenticated user is the recipient of the offer before allowing any modification.

6.2 Code change

The affected code is the following, and it was only necessary to add that check of the recipient==user at the beginning for it to work properly. The change was made in commit `ec38eea`.

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

def put(self, request, *args, **kwargs):
    id = kwargs.get('pk')
    offer = Offer.objects.get(pk=id)

    if offer.recipient != request.user:
        return Response({'error': 'You are not authorized to modify this offer'},
            status=403)

    try:
        offer.status = request.data.get('status')

        # Add the sender of the offer to the recipients list of athletes, and add the
            recipient to the senders coach list
        if offer.status == 'a':
            if not offer.recipient.isCoach:
                return Response({'error': 'Recipient is not a coach'}, status=400)

            offer.recipient.athletes.add(offer.owner)
            offer.recipient.save()

            # Delete other pending offers from the same sender to the same recipient
            Offer.objects.filter(owner=offer.owner, recipient=offer.recipient,
                status='p').delete()

    except Exception as e:
        return Response({f'error: {e}'}, status=400)

    partial = kwargs.pop('partial', False)
    serializer = self.get_serializer(offer, data=request.data, partial=partial)
    serializer.is_valid(raise_exception=True)
    serializer.save()

    return Response(serializer.data)
```

User List

```
GET /api/users/

HTTP 401 Unauthorized
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
WWW-Authenticate: Bearer realm="api"

{
    "detail": "Authentication credentials were not provided."
}
```

Figure 1: We don't have access anymore

7 WSTG-ATHN-04 - User is able to bypass authentication schema

7.1 Mitigation strategy

With this link, we can get EVERYONE (with an account) access to important user information that should remain private.
http://tdt4237-100.idi.ntnu.no/api/users/
The problem lies `backend/users/view.py` in the following class:

```python
class UserList(mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView):
    serializer_class = UserSerializer

    def get(self, request, *args, **kwargs):
        self.serializer_class = UserGetSerializer
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

    def get_queryset(self):
        qs = get_user_model().objects.all()

        if self.request.user:
            # Return the currently logged in user
            status = self.request.query_params.get("user", None)
            if status and status == "current":
                qs = get_user_model().objects.filter(pk=self.request.user.pk)

        return qs
```

There are no permission classes such as `permission_classes = [IsAuthenticated, IsAdminUser]` added, which allows everyone to access the URL.
Look at figure 1

7.2 Code change

```
class UserList(mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView):
    serializer_class = UserSerializer
    permission_classes = [IsAuthenticated]
    def get(self, request, *args, **kwargs):
        self.serializer_class = UserGetSerializer
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs ):
       return self.create(request, *args, **kwargs)

    def get_queryset(self):
        qs = get_user_model().objects.all()
        if self.request.user:
            status = self.request.query_params.get("user", None)
            if status and status == "current":
                qs = get_user_model().objects.filter(pk=self.request.user.pk)
        return qs
```

8 WSTG-ATHN-03 - Weak lockout mechanism

In this vulnerability, a user can insert wrong credentials when logging in as many times as they want. This makes it possible for someone to brute-force into finding the right credentials.

8.1 Mitigation strategy

To solve this, we need to implement a lockout mechanism. Django has a useful package for tracking failed login attempts and locking out users after a certain number of failed attempts, so it is not necessary to implement a custom solution. The idea is to configure the backend settings to use this package to implement the mechanism in our own personalized way.

First, we needed to install it and add the middleware. We had to make some modifications in versioning and applied the migrations in order to let the package fully integrate, else it would lead to docker container problems. We then defined an example of a configuration, that locks the person out for 1 hour after 5 failed attempts:

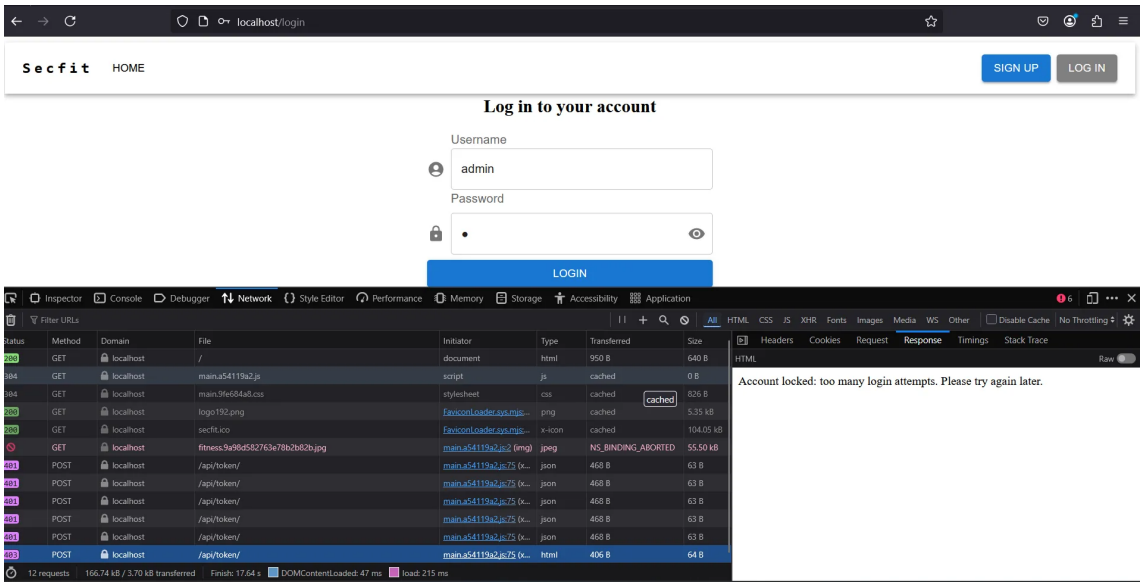

Figure 2: Black-box test once implemented lockout mechanism

8.2 Code change

There were many small code additions in different files in order to work, in `backend/secfit/settings.py`, `backend/users/urls.py`, and `backend/requirements.txt`. All new code is reflected in commit `021e631`, but here we add the code snippet of the actual configuration, which should be changed depending on the lockout desires of the app developer.

```
# Lockout configuration in backend/secfit/settings.py
AXES_FAILURE_LIMIT = 5 #num of attempts
AXES_COOLOFF_TIME = 1 #lockout time in hours
AXES_LOCKOUT_CALLABLE = None #use default configuration
AXES_ONLY_USER_FAILURES = False #who do you want the lockout to affect
AXES_RESET_ON_SUCCESS = True #reset after a correct login
```

9 WSTG-INPV-02 - Stored cross-site scripting

9.1 Mitigation strategy

A stored cross-site scripting is a very dangerous vulnerability for a website. It may execute a code inside the person browser.
In order to mitigate this risk I used the library "DOMPurify" in order to be sure to not being vulnerbale to XSS.
After importing the library to the react components I added the package with the command

```
npm install dompurify
```

in the frontend folder

9.2 Code change

In frontend/src/components/CommentSectionForm.jsx
This was the original code

```
<p dangerouslySetInnerHTML={{ __html: comment.content }}></p>
```

Then I changed it in order to fix the XSS vulnerability

```
import DOMPurify from "dompurify";
.
.
.
<p dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize(comment.content) }}></p>
```

10 WSTG-ATHN-01 - Credentials transported over an unencrypted channel

The website traffic was not encrypted. This meant that credentials, such as passwords, could be sniffed during login.

10.1 Mitigation strategy

To ensure that the webiste traffic is encrypted, we follow the instructions in the assignment description. Using OpenSSL, we generate a certificate and a private key. Then we configure TLS on the nginx server, and additionally reroute any http:// urls to https://

This ensures that TLS is used website wide, meaning that the traffic is encrypted.

To ensure the VM's access to the private key, it is uploaded into the git repository. In practice, this should not be done. Instead, it has to be stored in a secure location, where it can not be read be attackers. We have a similar case with the certificate which should not be modifiable, but might be due to storing it in the git repo.

10.2 Code change

In commit `89f72a9`, the certificate and private key which were created using OpenSSL are uploaded and stored in `nginx/ssl`.

The majority of the code changes were done in commit `03f3a0d`. As specified in the nginx documentation [4], we alter the `nginx.conf` file to

```
http {
    server {
        listen 443 ssl;
        server_name _;

        ssl_certificate     ssl/cert.pem;
        ssl_certificate_key ssl/key.pem;
        ssl_protocols       TLSv1.2 TLSv1.3;
        ssl_ciphers         HIGH:!aNULL:!MD5;
        ...
```

Additionally, we add the listening port 443 as well as the the folder containing the certificate and key to the `docker-compose.yml` file.

In commits `03f3a0d` and `47d7330`, another server with listening port 80 is added into `nginx.conf`, that reroutes traffic to the secure connection.

11 WSTG-BUSL-08 - Upload of unexpected file types and WSTG-BUSL-09 - Upload of malicious files

In the application, users were able to upload files of types other than pdf or images, and could also upload malicious files, such as executables. The `FileValidator` in SecFit only checked the file size and nothing else.

11.1 Mitigation strategy

Firstly, we limit the file types that the `FileValidator` accepts to pdf, png, jpg and jpeg in addition to having it check the file size.

However, the `FileValidator` only checks the extension of the file since it uses the outdated method `mimetypes.guess_type` to determine (or guess) the mime type [5]. An attacker could disguise a malicious file `malicious.exe` by renaming it to `safe.png` and the `FileValidator` would not detect it. This is why we also use the `python-magic` library. It checks the content of the file [6], which is our desired outcome.

It is useful and secure to practice Defense in Depth [7, 8]. For this reason, we additionally check the file size server-side and have the nginx server block executables.

11.2 Code change

The mitigation of these vulnerabilities was done in commit `af9a61b`.

As mentioned above, in `/backend/users/models.py` the allowed extensions and mimetypes of the `FileValidator` were set to pdf, png, jpg and jpeg. In addition to that, a second validator was added which is the following method:

```
# /backend/users/models.py

def validate_file_type(file):
    """Validate file type based on actual content."""
    mime = magic.Magic(mime=True)
    file_mimetype = mime.from_buffer(file.read(2048))
    file.seek(0) # Reset file pointer

    if file_mimetype not in ALLOWED_MIMETYPES:
        raise ValidationError("Unsupported file type.")
```

Here, the actual file type is determined based on the file's content and checked against the allowed types using the library `python-magic`. This library was also added to `backend/requirements.txt`.

Server-side, it is now also ensured in `nginx/nginx.conf` that the files are not too large and the extensions php, exe, sh, py, cgi, pl, and js are all denied in the location `/media/`.

12 Conclusion

We solved all the vulnerabilities that were required to be solved.
The website works perfectly after all the changes we made with the code without the vulnerabilities we had beforehand, making the website, generally speaking, safer.

References

[1] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: new generation of memory-hard functions for password hashing and other applications". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 292–302.

[2] Jean-Philippe Aumasson et al. *Password Hashing Competition*. 2025. URL: `https://www.password-hashing.net/` (visited on 03/05/2025).

[3] Django Software Foundation. *Password management in Django*. 2025. URL: `https://docs.djangoproject.com/en/5.1/topics/auth/passwords/` (visited on 03/05/2025).

[4] Sysoev, Igor. *Configuring HTTPS Servers in NGINX*. 2025. URL: `https://nginx.org/en/docs/http/configuring_https_servers.html` (visited on 03/07/2025).

[5] Python Software Foundation. *mimetypes — Mapping of filenames to MIME types*. Accessed: 2025-03-06. 2023. URL: `https://docs.python.org/3/library/mimetypes.html`.

[6] Johan P. Stokking. *python-magic — A Python interface to libmagic*. Accessed: 2025-03-06. 2023. URL: `https://pypi.org/project/python-magic/`.

[7] Ronald S Ross. "Guide for conducting risk assessments". In: (2012).

[8] Ron Ross, Victoria Pillitteri, and Kelley Dempsey. "Assessing enhanced security requirements for controlled unclassified information". In: *NIST Special Publication* 800 (2022), 172A.