COMP.SEC.300-2024-2025-1, Secure Programming, exercise work report

Lassi Rissanen
150312211
lassi.rissanen@tuni.fi
GitHub repository

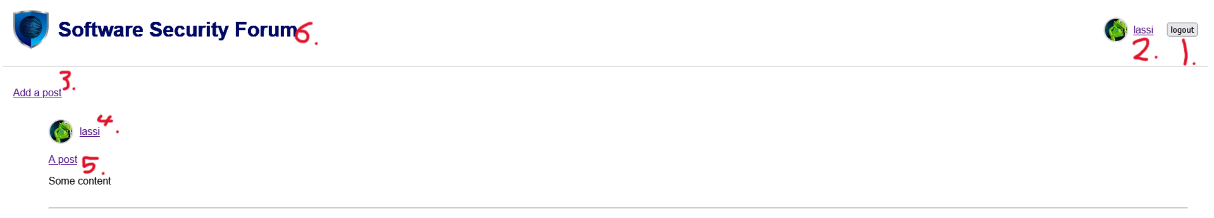
Introduction

The purpose of this exercise was to practise and deepen knowledge on secure programming principles and -practices through implementing a secure web application. The application implemented for this exercise work is a simple web forum for discussing topics related to security. Users can create accounts, login, make posts and comment on posts made by other users. The application is implemented using python Django framework. Django was chosen because it's a popular and actively maintained framework for implementing web applications and it comes with extensive documentation including security related issues.

Using the application

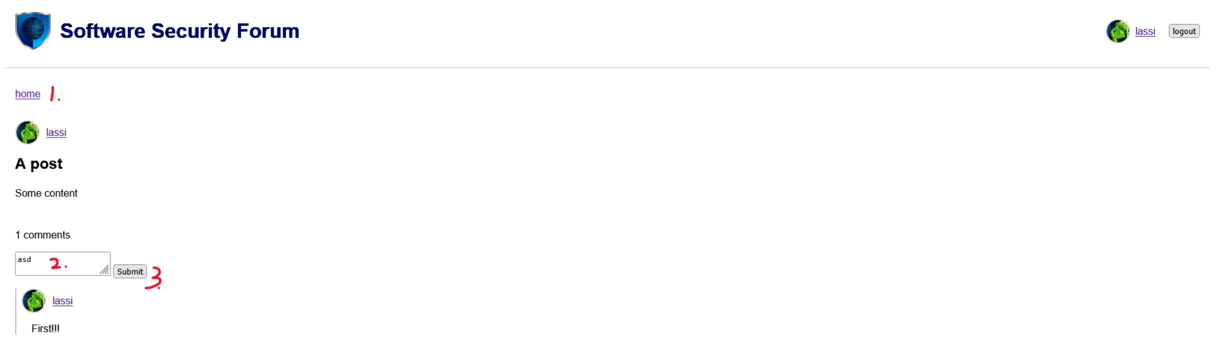
The instructions for running the application can be found in README on the projects' GitHub page. Using the application should be straightforward as there's not that many features. Only thing worth mentioning is the admin panel provided by Django that is accessible at "/admin-panel". To access this you have to create an admin user by running "*py manage.py createsuperuser*" in the "/project" directory on your computer or inside the docker container depending on how you run the app. The user interface is described below.

home page



1. login/logout
2. go to user profile
3. go to post form
4. go to user profile
5. go to post
6. go to home page

post page



lassi

A post

Some content

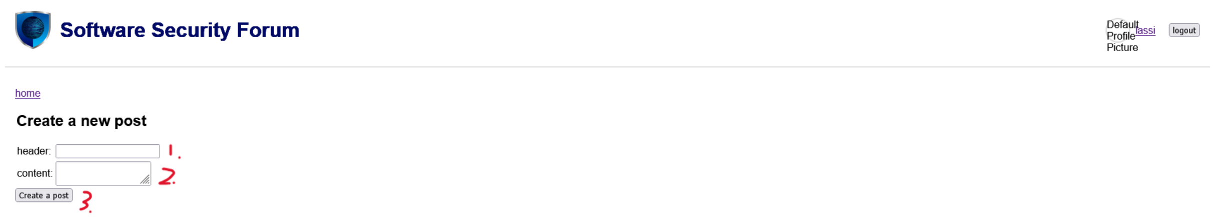
1 comments

asd

First!!!

1. go to home page
2. write comment
3. submit comment

post form



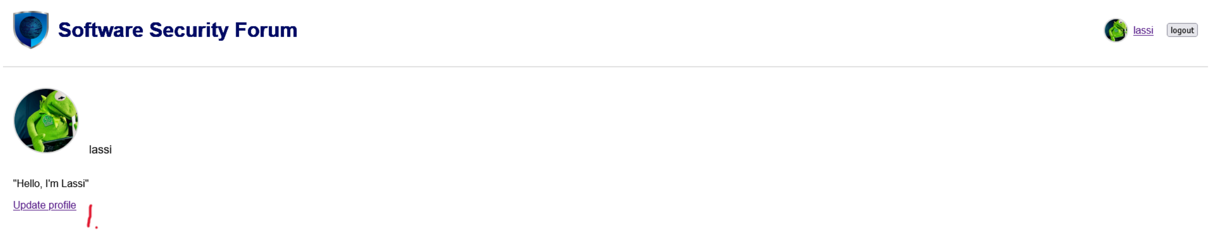
Create a new post

header:

content:

1. post header
2. post content
3. submit form

profile page


Software Security Forum

lassi logout

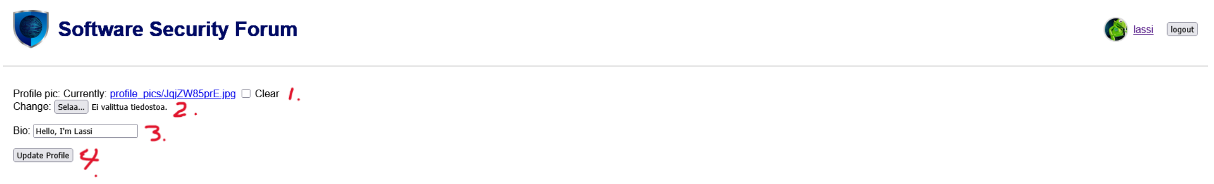
lassi

"Hello, I'm Lassi"

Update profile 1.

1. go to update profile form

update profile form


Software Security Forum

lassi logout

Profile pic: Currently: profile_pics/JqiZW85prE.jpg ☐ Clear 1.
Change: Selaa... Ei valittua tiedostoa. 2.

Bio: Hello, I'm Lassi 3.

Update Profile 4.

1. clear profile picture
2. set profile picture
3. set bio content
4. submit profile

Secure programming solutions

An effort was made to follow secure programming best practices during the implementation of this project. This includes choosing a well established and battle-tested framework, in this case Django, for the implementation, following general- and the framework's security guidelines, and implementing a security testing pipeline with github actions. The main resources to guide the secure implementation of the project were Django documentation, OWASP Django security cheat sheet, and OWASP top 10. The secure programming solutions implemented in this project are discussed below in the context of OWASP top 10 security risks.

owasp top 10 checklist

Owasp top 10 represents the most critical security risks in web applications. The following includes analysis on how the OWASP top 10 security risks are taken into account in this project application.

Broken access control

Broken access control means that sensitive data or actions are available to unauthorized actors. The sensitive actions in our project application are creating/deleting posts and comments, and updating the user profile. These should only be accessible to authenticated users, but unauthenticated users are allowed to see the posts, comments and profiles. The authorization is handled in "*securityforum/views.py*" by adding Django provided "*@login_required*" annotation for the views that require authentication. If the user is not authenticated they're forwarded to the login page.

Cryptographic failures

Cryptographic failures refers to exposure of sensitive data due to e.g. lack of encryption or weak hashing algorithms. The only data that really needs protection and should be cryptographically secured in the case of this project application is the user password.

The project application uses the Django provided user model for storing user information such as username and password. By default, Django uses BPKDF2 algorithm with SHA256 hash for hashing the password [1]. However, OWASP recommends using Argon2 for password hashing whenever possible [2]. To follow the best practices, Argon2 password hashing was implemented by using *argon2-cffi* python package. Using the package required configuring the "*Argon2PasswordHasher*" as the first element in the "*PASSWORD_HASHERS"* list in the main *settings.py* file.

Further protection for the user credentials would be to implement and force https connection between the client and server to encrypt the data sent between client and server. This would be something that would become necessary in production and is out of scope for this assignment.

injection

Injection happens when malicious user-provided data is passed to an interpreter and as a result the program does something it's not supposed to do. This could be for example revealing confidential data that the user shouldn't have access to. The project application implements protection against different kinds of injection attacks by largely taking advantage of features provided by the Django framework.

SQL injection is mitigated in Django by constructing the database queries using query parameterization. This means that query parameters, user-provided or not, are escaped by the underlying database driver making sure there's no unexpected side effects. Django also has the option to write raw SQL queries but this feature is generally not recommended and is not used in the project application.[3]

Django also provides some default protection against cross site scripting (XXS) by escaping angle brackets (<, >), quotes (', ") and "&" characters in html templates [4]. However, This by itself doesn't prevent all XSS attacks.

The user could also for example try to inject a script by uploading a html file as a profile picture. This has been prevented in "*securityforum/forms.ProfileUpdateForm*" by only allowing '*image/jpeg*' and '*image/png*' mime types. There's also a file size limit of 5 MB to prevent uploading very large files.

Insecure design

Insecure design refers to not taking into account or not properly addressing security related issues in the design of applications. To promote secure design, owasp recommends for example establishing and using a secure development lifecycle, using libraries of secure design patterns or paved road ready to use components, and using threat modeling for critical parts of the application. [5]

The biggest secure design solution in this project application is to use the Django framework, the benefit being that Django comes with battle tested paved road solutions to e.g. authentication/authorization, password storage and injection prevention.

To cover some threat modeling, the biggest and most probable threat in the project application is that the users try to inject some malicious data in posts/comments or username fields. This could for example make the page unavailable, alter some data in the database or make it possible to hijack the user session. The prevention measures for this were described previously in the section about injection.

There was really no secure development process or -lifecycle followed during the implementation of the project as it was just one person arbitrarily making decisions. However, an effort was made to follow secure programming practices and take software security in account as extensively as possible. In the real world, establishing a secure development process and -lifecycle with security professionals would be advisable.

security misconfiguration

Security misconfiguration vulnerabilities include for example default accounts with unchanged passwords and unnecessary features or privileges. The protection against most vulnerabilities in this category would become timely when deploying the application in production. The Django documentation provides a clear checklist for reviewing the project settings and configuration for deployment. The checklist includes for example enabling https connections and disabling debug mode[6].

The project application implements some secure configuration measures such as setting debug mode to false in production, loading *SECRET_KEY* from the environment variables, and configuring *ALLOWED_HOSTS*- and *CSRF_TRUSTED_ORIGINS* settings in *setting.py* to protect the site against CSRF attacks. However, some recommended security configuration such as setting up https is left as a todo for when the application would be deployed to production.

vulnerable and outdated components

Vulnerable and outdated components means for example using 3rd party dependencies that have vulnerabilities or are not up to date. To prevent vulnerabilities in this category, one has to make sure there are no unnecessary or unused components, keep track of the dependencies in their application and keep them up to date. Furthermore, one should only use trusted components and dependencies from official sources.

To help defend against vulnerabilities in this category, there's a github workflow that generates SBOM- file and runs OWASP dependency-check on the application when a pull request or push is made to the main branch. The code for the workflow can be found in the repository path "*.github\workflows\security-pipeline.yaml*". The workflow produces an artifact called "Depcheck report" that contains sbom.json file and the dependency-check report in html format. The artefacts can be downloaded from the GitHub "actions" tab of the repository. This by itself doesn't protect against the vulnerabilities, but helps in discovering them. It's also worth mentioning that the python related file analyzers in OWASP dependency-check are considered experimental at the moment.

identification and authentication failures

Identification and authentication failures include vulnerabilities such as allowing brute force attacks and the use of insecure passwords. To address vulnerabilities of this category, our project application uses the Django password validation for created accounts. The requirements for the password are that it has to be at least 8 characters long, can't be too similar to other user information such as username, can't be a common password and can't be entirely numeric. The password validation is configured by the *AUTH_PASSWORD_VALIDATORS* array in *settings.py*.

The application also has protection against brute-force attacks. This is implemented by using *python-axes* library which is one of the tools recommended by OWASP Django security cheat sheet [7]. Using the library is quite straightforward as it only needs to be installed and then configured in the *settings.py* where the related settings are prefixed with "*AXES_*". The library is configured so that it allows 4 login attempts and after that the account is locked for 30 minutes preventing further login attempts.

For session management, the application uses the session manager provided by the Django framework. The session cookies are invalidated on logout and after 2 weeks of inactivity. [8]

Some security features regarding this category that should be implemented in the future include multi-factor authentication and credential recovery process.

software and data integrity failures

Software and data integrity failures happen for example when an application uses plugins, libraries or modules that are acquired from untrusted sources. To tackle vulnerabilities of this category, the project application implements the following security measures: consumes 3rd party software only from trusted sources, checks dependencies using OWASP dependency-check, and limits the permissions of github workflows (e.g., the workflow running dependency-check has only read access to the repository)

Further safety measures that should be implemented in the future would include establishing a secure CI/CD pipeline to ensure code integrity throughout the build and deployment process. Also in the case where the development team would get bigger than one person, a secure development process with peer reviewed pull requests would be needed to prevent introducing malicious code to the codebase.

security logging and monitoring failures

Security logging and monitoring failures refer to insufficient logging and monitoring of critical events such as logins, failed logins and other high value transactions. Logging is implemented in the project application so that there's a log file for every day where logs are written, excluding DEBUG level logs. The log file path and name can be configured by setting the *DJANGO_LOG_PATH* environment variable. Most importantly, the authentication related events such as logins and failed logins are logged. The logs are kept for 30 days so that they can be retrospectively analyzed.

Further development could include developing a monitoring and alerting system for the logs so that suspicious activities could quickly be detected and responded to.

Server-side request forgery

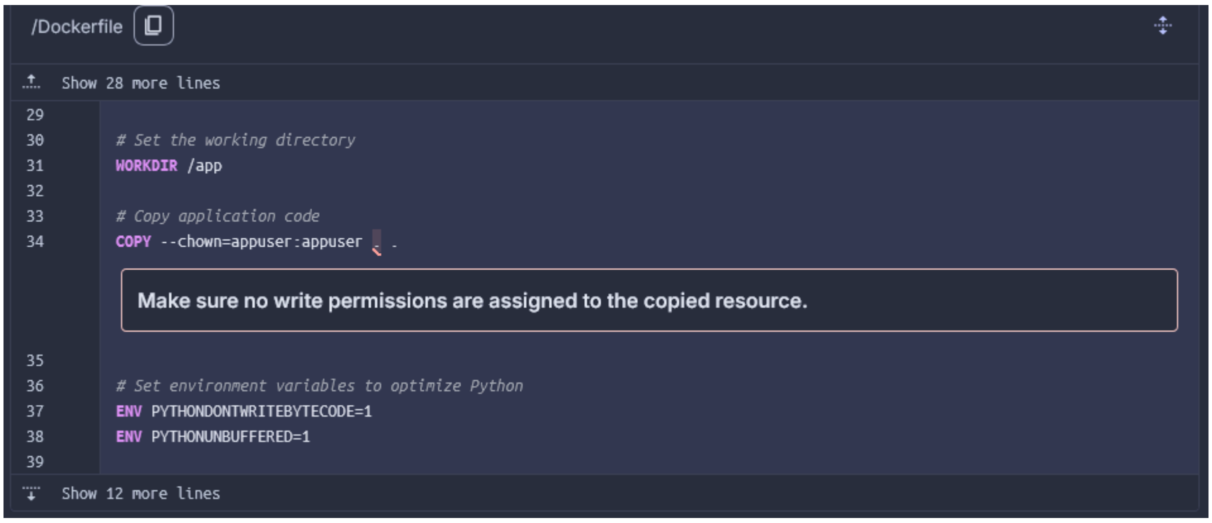
Server-side request forgery occurs when a server fetches a remote resource without validating user supplied url. In our project application, the only user provided thing relating to urls would be the file name of the profile picture. This should not be a problem, but just to be sure, the file names of profile pictures are transformed to random strings instead of using the user provided file name.

Testing

To cover security testing, the project application implements a security testing pipeline that includes static application security testing with SonarQube, software composition analysis with OWASP Dependency-Check, and dynamic application security testing with Checkmarx ZAP. The pipeline is implemented using github actions and the code for the pipeline can be found in the repository under ".github/workflows/security-pipeline.yaml". The workflow produces artifacts for each workflow run. These can be accessed by clicking a workflow run under the "actions" tab of the GitHub Repository. The artifacts include "Depcheck report" and "zap_scan", containing the corresponding reports. SonarQube reports are available in SonarCloud, but are not public.

findings

SonarQube analysis revealed at least one critical vulnerability in the docker configuration. The vulnerability had to do with loose write permissions for the user running the application. This was fixed by restricting the permissions for the user.



```

# Set the working directory
WORKDIR /app

# Copy application code
COPY --chown=appuser:appuser . .
```

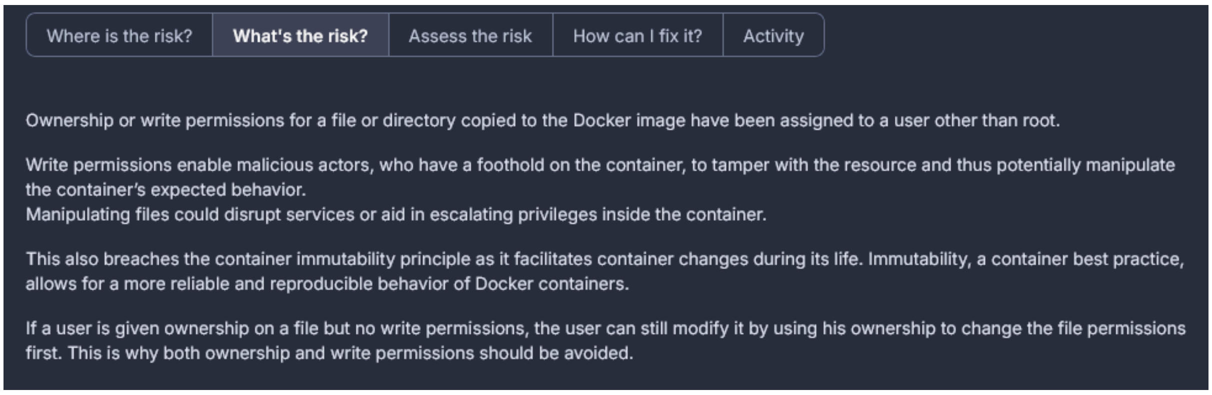
Make sure no write permissions are assigned to the copied resource.

```

# Set environment variables to optimize Python
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

```

Dockerfile vulnerability



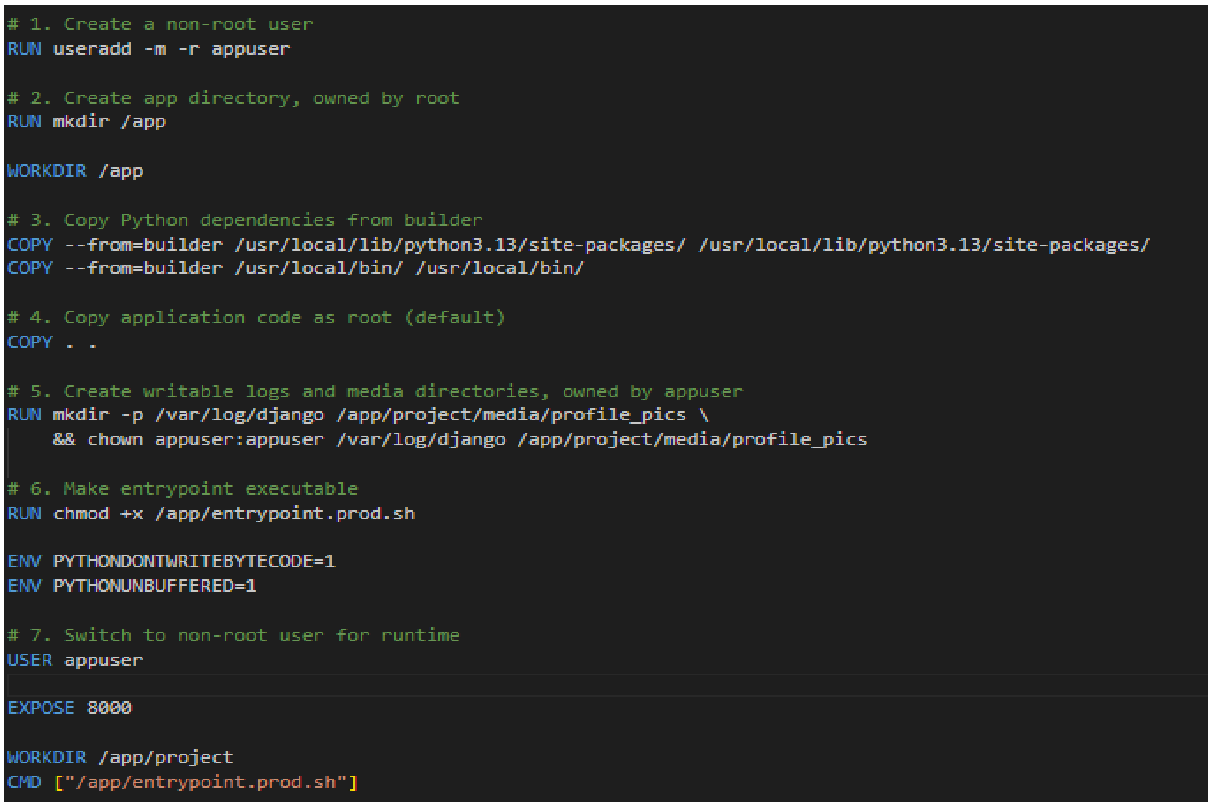
Ownership or write permissions for a file or directory copied to the Docker image have been assigned to a user other than root.

Write permissions enable malicious actors, who have a foothold on the container, to tamper with the resource and thus potentially manipulate the container's expected behavior.
Manipulating files could disrupt services or aid in escalating privileges inside the container.

This also breaches the container immutability principle as it facilitates container changes during its life. Immutability, a container best practice, allows for a more reliable and reproducible behavior of Docker containers.

If a user is given ownership on a file but no write permissions, the user can still modify it by using his ownership to change the file permissions first. This is why both ownership and write permissions should be avoided.

Vulnerability description



```
# 1. Create a non-root user
RUN useradd -m -r appuser

# 2. Create app directory, owned by root
RUN mkdir /app

WORKDIR /app

# 3. Copy Python dependencies from builder
COPY --from=builder /usr/local/lib/python3.13/site-packages/ /usr/local/lib/python3.13/site-packages/
COPY --from=builder /usr/local/bin/ /usr/local/bin/

# 4. Copy application code as root (default)
COPY . .

# 5. Create writable logs and media directories, owned by appuser
RUN mkdir -p /var/log/django /app/project/media/profile_pics \
    && chown appuser:appuser /var/log/django /app/project/media/profile_pics

# 6. Make entrypoint executable
RUN chmod +x /app/entrypoint.prod.sh

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# 7. Switch to non-root user for runtime
USER appuser

EXPOSE 8000

WORKDIR /app/project
CMD ["/app/entrypoint.prod.sh"]
```

Fixed dockerfile

The OWASP Dependency-Check analysis also revealed the **CVE-2025-27556** vulnerability in Django version 5.1.7 that was used in the application. This was fixed by updating Django to version 5.2.

The dynamic application security testing with Checkmarx ZAP raised two of medium-, 3 low-, and 5 informational level risk alerts. The medium level alerts were “http only site” and “Content Security Policy (CSP) Header Not Set”. The csp headers were added to the project using the *django-csp* library and configuring the *CONTENT_SECURITY_POLICY* setting in *settings.py.* The report states that the issue still exists after fixing the problem, but manual testing shows that the csp header is set. Setting the csp header adds security as it protects against cross-site scripting and clickjacking. The lower level alerts are ignored for now.

Known vulnerabilities

The communication happens through http which is insecure especially when sending credentials. This also prevents using some security related configuration such as setting *CSRF_COOKIE_SECURE* and *SESSION_COOKIE_SECURE* to *true*. In the actual production environment https should be used.

Missing features & further development

From a functional point of view, the project application is still very much immature and unfinished. Further development could include for example implementing search, up-/downvote and moderator features. However, the point of this project was to not focus on the application functionality itself, but to practise and deepen knowledge on secure programming principles and -practices.

From a security point of view, the application should be reasonably secure as an effort was made to follow secure programming practices. As described above the application takes into account at least the OWASP top 10 vulnerabilities quite extensively. One major feature that should be implemented in the case of actually deploying the application to production is configuring the communication through https instead of http. The security testing pipeline could also be enhanced by implementing file system- and container scanning using e.g. Snyk.

References

1. Django Password management: https://docs.djangoproject.com/en/5.1/topics/auth/passwords/
2. Owasp Password Storage Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
3. Django SQL injection protection: https://docs.djangoproject.com/en/5.1/topics/security/#sql-injection-protection
4. Django XSS protection: https://docs.djangoproject.com/en/5.1/topics/security/#cross-site-scripting-xss-protection
5. OWASP top 10, insecure design: https://owasp.org/Top10/A04_2021-Insecure_Design/
6. Django deployment checklist: https://docs.djangoproject.com/en/5.1/howto/deployment/checklist/
7. Django security cheat sheet: https://cheatsheetseries.owasp.org/cheatsheets/Django_Security_Cheat_Sheet.html
8. Django sessions: https://docs.djangoproject.com/en/5.1/topics/http/sessions/