There are some imperfect or even inappropriate approaches to write our code and do the testing, which probably lead to corresponding potential risk and vulnerability of our program, which might also negatively affect the security of the user-end once the web-app being used and attacked by other malicious users.

The dangers and the reasons behind them we could have come up with yet are listed as following, we also wrote some possible solution which has not been applied to our correspondent program due to the limit of time. This document will be keeping updated later.

Dangers & Reasons:

1 The user authorization should be performed more properly and consistently. The program files, even the log file(which only be able to be written as root user), are written with not much access authentication concerns. The directory is writeable by the user that the server runs as, which is "nobody" under default configurations. This can make the effect of some bugs in the code much worse than they normally be, especially when a bug enable s a remote user to run arbitrary commands on the server;

2 XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. Django templates escape spesific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. If the following statement:
<style class={{ var }}>...</style>
the **var** is set to **'class1 onmouseover=javascript:func()'**, this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML.

3 If using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping, however either our program and the django is confident about considering about them all ;

4 Since our data deals with the database a lot, there is some operation such as storing HTML in the database could also bring dangers, especially when that HTML is retrieved and displayed;

5 CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent. Django has built-in protection against most types of CSRF attacks, However, there are limitations. For example, it is possible to disable the CSRF module globally or for particular views

6 Django uses the **Host** header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, a fake **Host** value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails. Django validates **Host** headers against the ALLOWED_HOST setting in the django.http.HttpRequest.get_host() method. This validation only applies via get_host(); if your code accesses the **Host** header directly from **request.META** you are bypassing this security protection.

7 SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage. Django gives developers power to write raw queries or execute custom sql. However, these capabilities should be used sparingly and we should always be careful to properly escape any parameters that the user can control.

8 Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site. Django contains protection in the form of one of the middlewares which in a supporting browser can prevent a site from being rendered inside a frame. However, it is possible to disable the protection on a per view basis or to configure the exact header value sent.

9 If a browser connects initially via HTTP, which is the default for most browsers, it is possible for existing cookies to be leaked。 For this reason, we actually should instruct the browser to only send these cookies over HTTPS connections. Furthermore, HSTS is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS. Combined with redirecting requests over HTTP to HTTPS, this will ensure that connections always enjoy the added security of SSL provided one successful connection has occurred.

10 We don't want users to be able to execute arbitrary code by uploading and requesting a specially crafted file； Django's media upload handling poses some vulnerabilities when that media is served in ways that do not follow security best practices. Specifically, an HTML file can be uploaded as an image if that file contains a valid PNG header followed by malicious HTML.

11 Our Python code should be ensured to be outside of the Web server's root. This will ensure that the Python code is not accidentally served as plain text (or accidentally executed).

12 Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, we should consider deploying a Django plugin or Web server module to throttle these requests.

13 The configuration information in settings.py such as SECRET_KEY and ALLOWED_HOST should be kept properly safe.

14 Maybe we should limit the accessibility of our caching system and database using a firewall.

Reference: https://www.djangoproject.com/start/overview/