## Part 1: Conceptual Questions

- **Q1:** Read Django's documentation and answer the following question:
    - What is the purpose of `{% csrf_token %}` in the template files?

The CSRF token protects against Cross-Site Request Forgeries. This token needs to be included with any "unsafe" request. This verifies the identity of the user with the user session ID.

- **Q2:** What could happen if a developer forgets to add the `{% csrf_token %}` tag on their template that uses an HTML form?

The Django CSRF Middleware would look for the token in the request. However, it would not find anything. The request would be rejected and return with a 403 error. However, if you remove the CSRF middleware settings in the Django settings.py, this would make a serious vulnerability and allow for Cross-Site Request Forgery attacks.

- **Q3:** What is the CWE ID associated with the security problem that arises with forgetting to use `{% csrf_token %}`?

The CWE-ID is CWE-352: Cross-Site Request Forgery (CSRF). This occurs when a web application does not verify that a request was intentionally sent by a user.

## Part 2: Finding Issues

In this homework part, you will play the role of a ***software consultant*** hired to identify the security vulnerabilities in a Web application. This is a simple Web application where a user can add/delete/lists tasks (i.e., a task management app). In light of that, answer the following question:

- **Q1:** Enumerate all the vulnerabilities you found in the web application. Include in your answer the following:

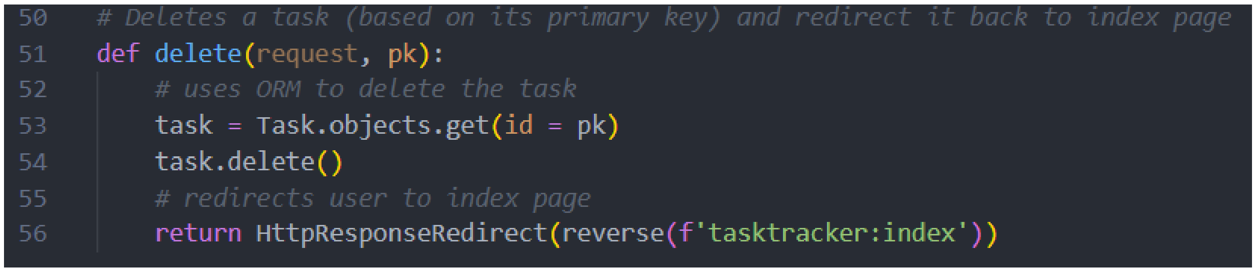
Vulnerability #1:

- website/tasktracker/templates/index.html
    - Line 11
- First, navigate to this page: http://127.0.0.1:8000/tasktracker/add/. Then you can input any HTML/JS code into the task title box. For example, I input the following into the title box: <script>alert("Hello");</script>. This JS code is run, and an alert is displayed. This could be used in a Cross-Site Scripting attack with malicious code. This vulnerability is possible because of this line: `<b>Task #{{t.id}}: {{t.title | safe}}</b>`. The problem here is the "| safe" part. This is a Django template filter that tells the application to render the title as is, rather than escaping variables to avoid running malicious code.

- CWE-79: Improper Neutralization of Input During Web Page Generation (‘Cross-site Scripting’)

Vulnerability #2

- The problem is located in website/tasktracker/views.py

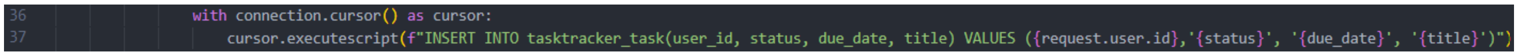


```python
# Deletes a task (based on its primary key) and redirect it back to index page
def delete(request, pk):
    # uses ORM to delete the task
    task = Task.objects.get(id = pk)
    task.delete()
    # redirects user to index page
    return HttpResponseRedirect(reverse(f'tasktracker:index'))
```

- The problem here is that the view for delete() does not check authentication or permission. This problem also arises because the user can control the primary key from the URL. The ID for each task is just the task number, which counts up sequentially. Therefore, you can navigate from the login page to http://127.0.0.1:8000/tasktracker/delete/20 to delete task #20 without even being logged in as that user or on the task tracker main page. Someone could send requests to this site for every ID from 0 to 999999 and delete every task in the database.
- There are several CWE-IDs that I believe this could fall under
    - CWE-639: Authorization Bypass Through User-Controlled Key
    - CWE-285: Improper Access Control
    - CWE-284: Improper Authorization

Vulnerability #3

- The problem is located in website/tasktracker/views.py



```python
            with connection.cursor() as cursor:
                cursor.executescript(f"INSERT INTO tasktracker_task(user_id, status, due_date, title) VALUES ({request.user.id},'{status}', '{due_date}', '{title}')")
```

- The problem here is that we use raw SQL code and concatenate user inputs into the SQL command. This allows the user to perform SQL injection to manipulate the database with malicious code. For example, if you navigate to http://127.0.0.1:8000/tasktracker/add and input the following as the title: **Test'); DROP TABLE tasktracker_task;--**, It will delete the entire database. In my testing, I actually successfully deleted the database and just had to reclone the repo!
- CWE-89: Improper Neutralization of Special Elements used in a SQL Command (‘SQL Injection’)

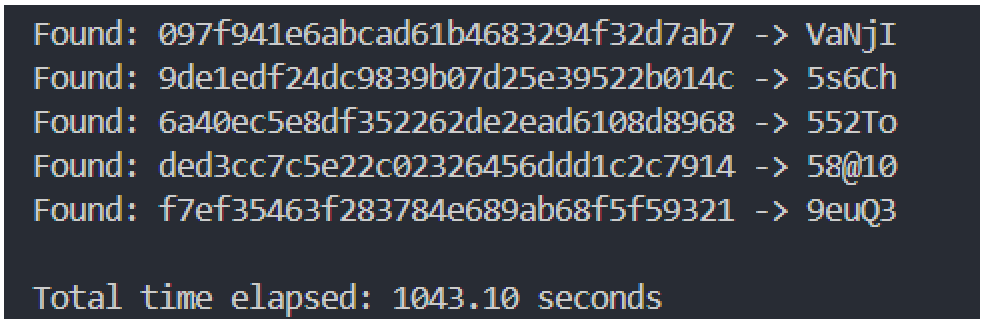
Vulnerability #4

- This problem is located on the login page. It is a general problem with login.html and the background logic.

- The problem here is that there is no rate limiting on the login page. I repeatedly entered incorrect passwords to test this. I was never limited or stopped. Therefore, a malicious attacker could brute force the login page to discover users' passwords and access their accounts.
- CWE-307: Improper Restriction of Excessive Authentication Attempts

## Part 3: Cracking Passwords

67 passwords found in 1043.10 seconds with get-pws-parallel.py



```
Found: 097f941e6abcad61b4683294f32d7ab7 -> VaNjI
Found: 9de1edf24dc9839b07d25e39522b014c -> 5s6Ch
Found: 6a40ec5e8df352262de2ead6108d8968 -> 552To
Found: ded3cc7c5e22c02326456ddd1c2c7914 -> 58@10
Found: f7ef35463f283784e689ab68f5f59321 -> 9euQ3

Total time elapsed: 1043.10 seconds
```

## Part 4: Research Synthesis

Graduate students are expected to have evidence of *research synthesis* during the semester. To fulfill this requirement, please read the following paper and answer the questions below: Pythia: Identifying Dangerous Data-flows in Django-based Applications

1) **What is the motivation and problem being tackled by this paper and how they solve this problem?**
   *Answer will be 2-3 paragraphs in length.*

Django, as well as other web frameworks, provide default security checks to allow developers to easily protect their applications. However, developers have the option to disable these checks. This often leads to Cross-site scripting (XSS) and Cross-site request forgery (CSRF) vulnerabilities. However, identifying these vulnerabilities can be very difficult due to the complexity of applications, features, and inheritance patterns. The authors developed Pythia, which analyzes applications based on the Django framework to find dangerous data flows that could lead to XSS and CSRF. Other tools like this usually do not incorporate framework specific features. Therefore, the authors claim Pythia to be the first of its kind.

Pythia analyzes applications to identify data-paths involving dangerous constructs. It does this by analyzing the templates and views files. The tool first attempts to identify sinks, which is a coding construct where the hazard might take place. I'm assuming this means cases like those we found in Part 2 (Ex: Uses of “| safe”). It then determines whether malicious users could reach these constructs and if other sinks are being used by those views. I’m unsure whether this tool would identify vulnerability #2 or not. It talked about looking for key decorators, tags, etc., but the problem here is the lack of authentication code in the delete view.

2) **Would Pythia help you in finding the vulnerabilities you identified in Part 2?**
   *The answer will be* ***at least*** *half a page, explaining in details how Pythia could have helped you in finding vulnerabilities in Part 2. In case you think the tool could not help you, explain in details why it is the case (i.e., what is it on this tool that makes it unsuitable to find the vulnerabilities you have found).*

Yes, I believe this tool would have been extremely useful in Part 2. However, I do not believe it would have been able to identify every vulnerability I found. This could be problematic if users completely rely on this tool to find every vulnerability. This tool would have certainly identified Vulnerability #1 (in Part 2) because the paper explicitly talks about the mark safe feature in Django. This is known to be a problematic feature so Pythia always identifies it as a sink. Upon further examination, Pythia would probably also identify that this sink could be reached with untrusted data, making it a vulnerability. I believe this tool would have also identified the SQL injection vulnerability (#3). This would be an in-view sink and should be identifiable because there is SQL code concatenated with user inputs without proper validation. Finally, I don’t believe vulnerability #4 would be detected by this tool because the problem is a lack of rate-limiting on this page. I don’t believe this would be identified as a sink and would not be found.