

# MAKERERE

P.O Box 7062 Kampala Uganda  
Email: [principal.cis@mak.ac.ug](mailto:principal.cis@mak.ac.ug)  
Website: [www.cocis.mak.ac.ug](http://www.cocis.mak.ac.ug)



# UNIVERSITY

Telephone: +256-414 545029  
Fax: +256-41 532780

---

## COLLEGE OF COMPUTING AND INFORMATION SCIENCES

## SCHOOL OF COMPUTING AND INFORMATICS TECHNOLOGY

Course Bachelor Of Science In Computer Science

Course Unit Advanced Programming

Course Code CSC 3115

### COURSE WORK 3

	Students Names	Registration No	Student No
1	Owiny Marvin	22/U/3783/EVE	2200703783
2	Natukunda Phionah	22/U/3658/PS	2200703658
3	Magino Daniel	22/U/21768/PS	2200721768
4	Kaitesi Joan	22/U/3141/PS	2200703141

Submitted on 17<sup>th</sup> November, 2024

Explain to what extent the fix and reasoning are justified and follow good secure programming principles.

Implementing additional checks to validate both the session ID and user ID for every URL request — is a valid and effective solution. It follows key secure programming principles, particularly those of **input validation** and **session management**.

**Session Integrity:** By validating the session ID and user ID, the application can confirm that the request originates from the same user who initiated the session. This prevents unauthorized users from hijacking sessions, thus maintaining session integrity.

**Authentication and Authorization:** By verifying the user ID associated with the session ID, the fix ensures that the application only processes requests if the session corresponds to the correct user, mitigating the risk of session hijacking.

**Secure Programming Principles:** The fix demonstrates **input validation**, where both the session ID and the user ID are checked before processing the request, preventing unauthorized access to sensitive data and operations.

Briefly identify and describe any four security Requirements that have been showcased in the case scenario above

**Authentication:**

Ensures that only authorized users can access the application. The requirement to check the user ID with every request reinforces this by verifying that the user is who they claim to be.

**Session Management:**

Involves maintaining a secure session for each user. The fix by validating session ID and user ID helps to protect session integrity and prevent session hijacking.

**Access Control:**

Controls who can access specific resources and perform certain actions. The vulnerability and subsequent fix highlight the need for stringent access control to ensure that unauthorized users cannot perform administrative tasks.

**Data Integrity:**

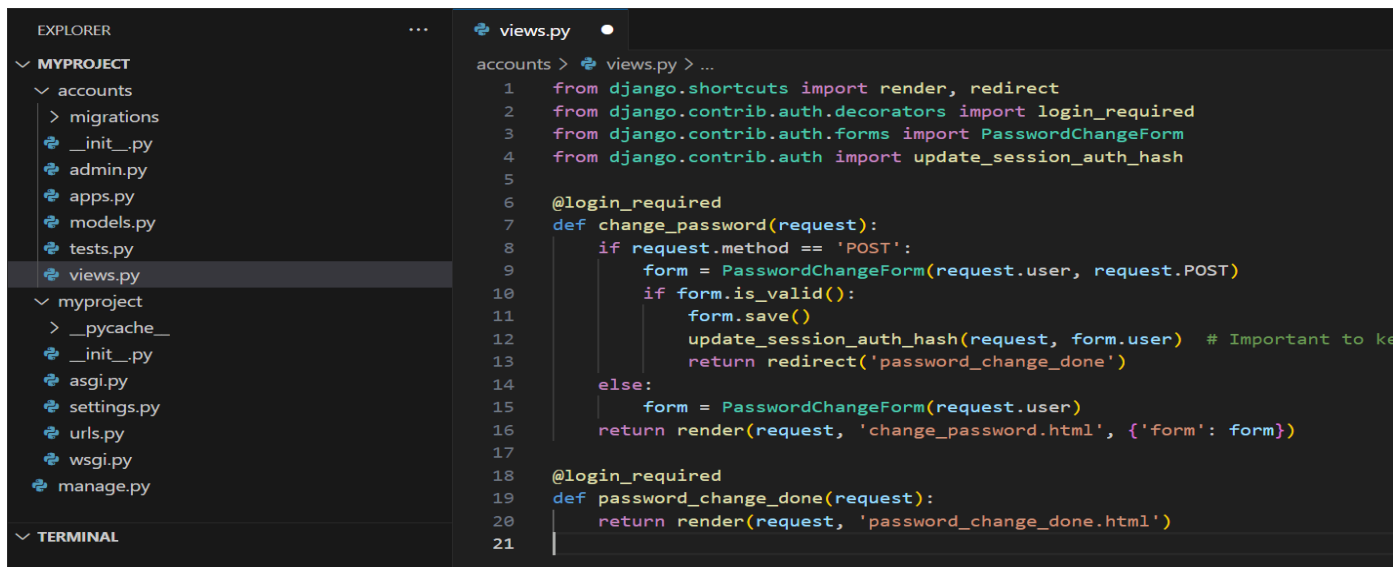
Ensures that data is accurate and consistent during transmission and storage.  
Validating session and user IDs for each request helps in maintaining the integrity of the session data, preventing unauthorized modifications

## Simulating the Security Vulnerability in Django

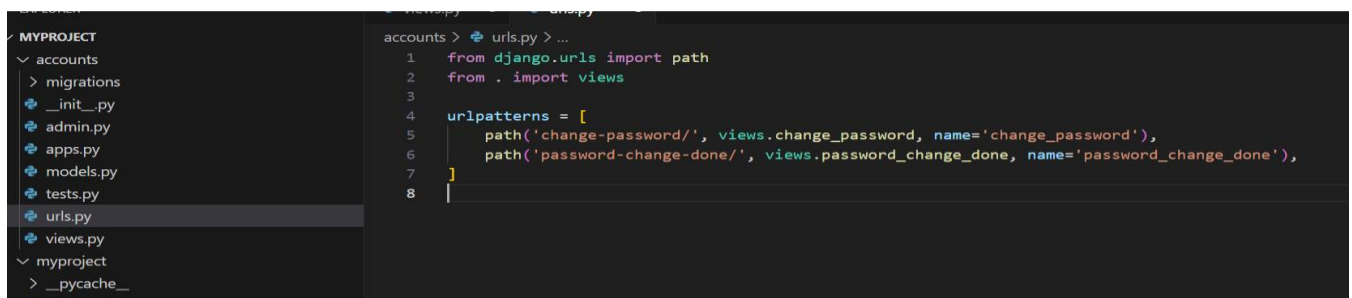
create a Django application where an unauthorized user can copy a session token from an authorized user, paste it into their session, and gain access to the privileged functionality of the authorized user (e.g., changing the admin password).

A view where authenticated users can change their password.

**A scenario where the session is hijacked by copying the session token**



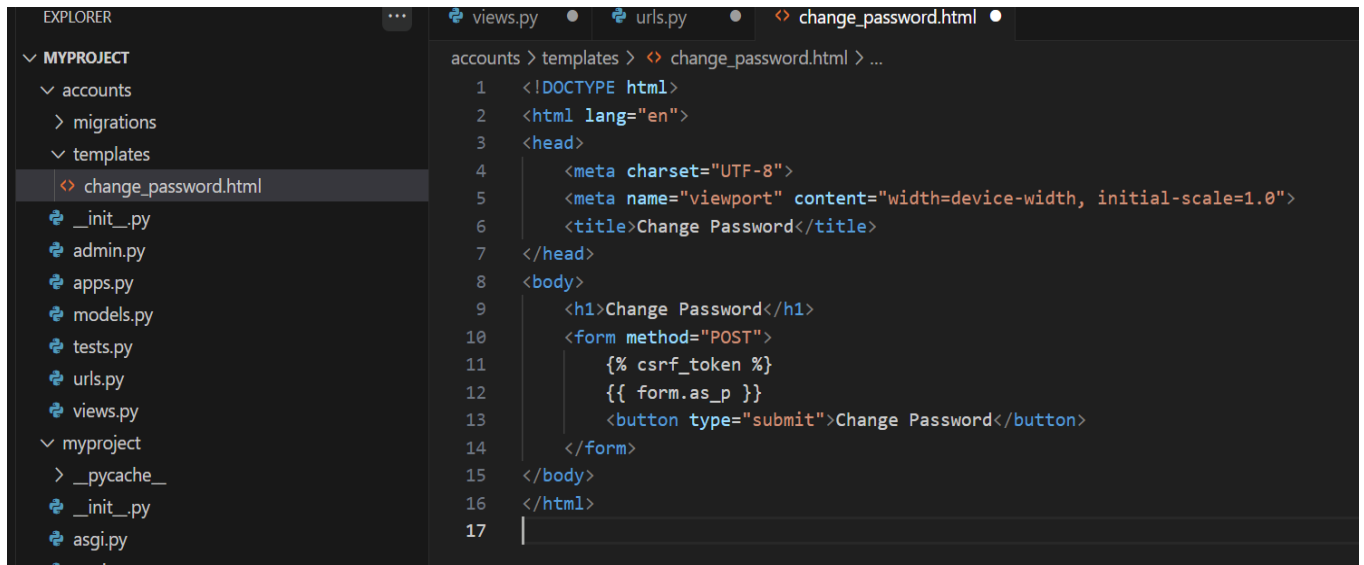
```
1 from django.shortcuts import render, redirect
2 from django.contrib.auth.decorators import login_required
3 from django.contrib.auth.forms import PasswordChangeForm
4 from django.contrib.auth import update_session_auth_hash
5
6 @login_required
7 def change_password(request):
8     if request.method == 'POST':
9         form = PasswordChangeForm(request.user, request.POST)
10        if form.is_valid():
11            form.save()
12            update_session_auth_hash(request, form.user) # Important to keep session valid
13            return redirect('password_change_done')
14        else:
15            form = PasswordChangeForm(request.user)
16            return render(request, 'change_password.html', {'form': form})
17
18 @login_required
19 def password_change_done(request):
20     return render(request, 'password_change_done.html')
```



```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('change-password/', views.change_password, name='change_password'),
6     path('password-change-done/', views.password_change_done, name='password_change_done'),
7 ]
```

URLs for both the password change form and the confirmation page

templates to render the password change form and a confirmation page.

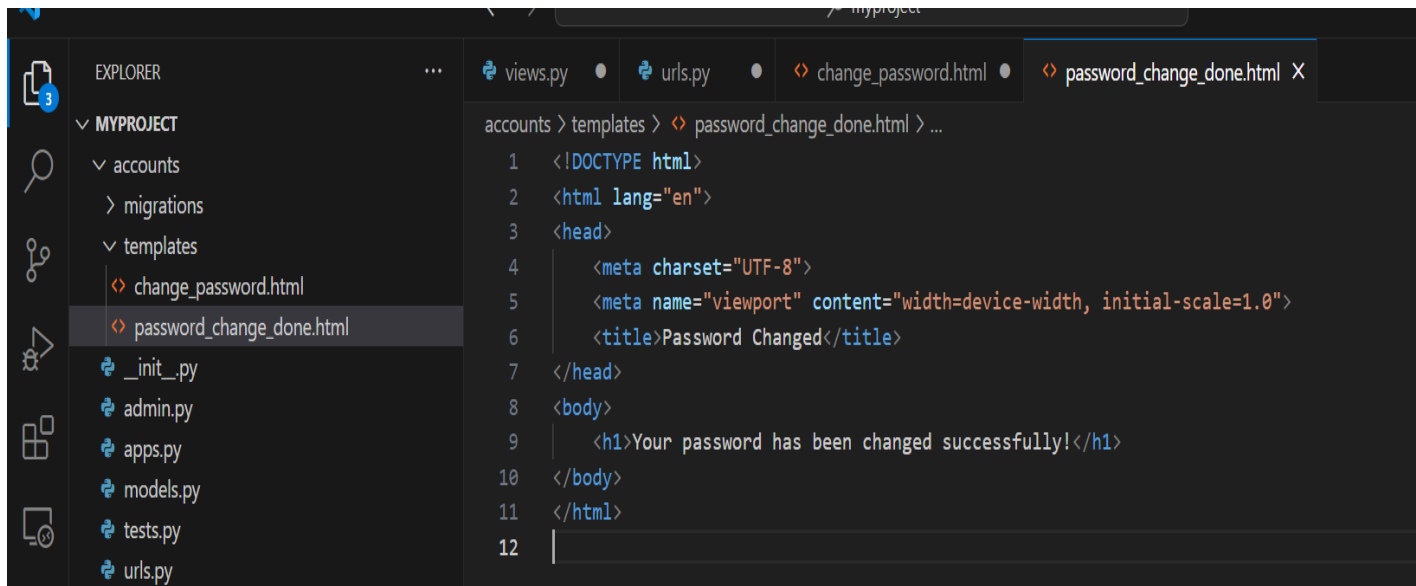


The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the Editor window on the right. The Explorer sidebar shows a project structure with 'MYPROJECT' containing 'accounts', 'migrations', and 'templates'. The 'templates' folder is expanded, showing 'change\_password.html' selected. The Editor window displays the content of 'change\_password.html' with the following code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Change Password</title>
7 </head>
8 <body>
9   <h1>Change Password</h1>
10  <form method="POST">
11    {% csrf_token %}
12    {{ form.as_p }}
13    <button type="submit">Change Password</button>
14  </form>
15 </body>
16 </html>
17
```

## Unauthorized User Changes Password

- Show the attacker changing the password on the page, despite being unauthorized



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the Editor window on the right. The Explorer sidebar shows the project structure with 'MYPROJECT' containing 'accounts', 'migrations', and 'templates'. The 'templates' folder is expanded, showing 'change\_password.html' and 'password\_change\_done.html' selected. The Editor window displays the content of 'password\_change\_done.html' with the following code:

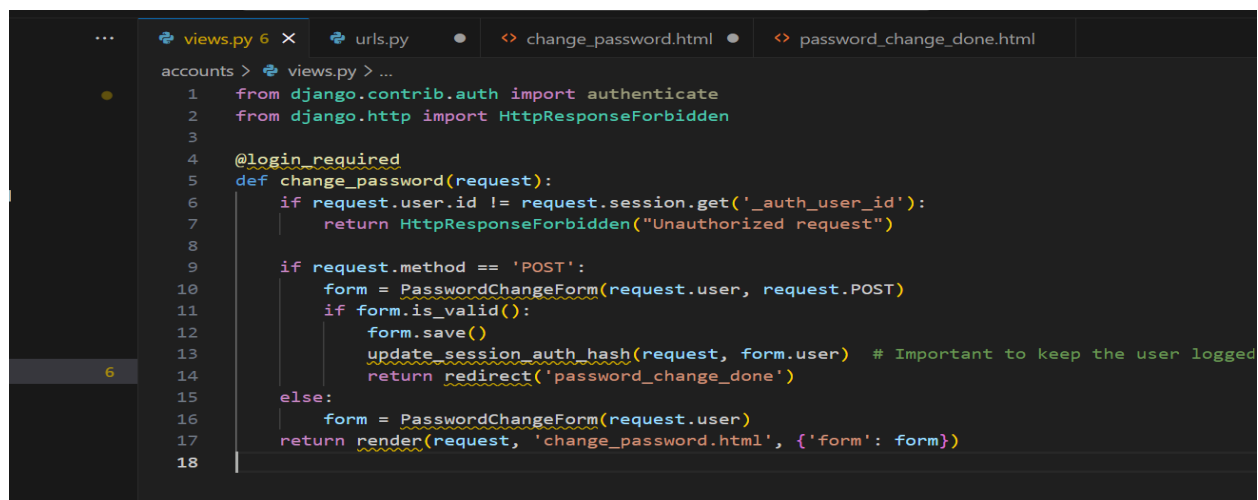
```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Password Changed</title>
7 </head>
8 <body>
9   <h1>Your password has been changed successfully!</h1>
10 </body>
11 </html>
12
```

## Steps to Simulate the Attack:

1. **Authorized User Logs In:** The authorized user logs in to the application through the login page and accesses the password change page. They see the page at `/change-password/`.
2. **Unauthorized User Copies Session:** After logging in, the session ID (which is stored in the browser's cookies) is sent with each request to the server. The attacker (unauthorized user) can copy the URL from the authenticated user's session or directly copy the session cookie from their browser.
3. **Unauthorized User Pastes the Session Token:** The attacker then logs into their own session and manually changes the session ID (cookie) in their browser's developer tools or pastes the copied session URL into their browser's address bar.
4. **Unauthorized User Gains Access:** If there are no checks in place to validate the session ID with the user ID (session hijacking), the attacker is able to perform actions like changing the password on the authorized user's account, as they are now using the same session token.

## Fix the Vulnerability (Implement Proper Validation)

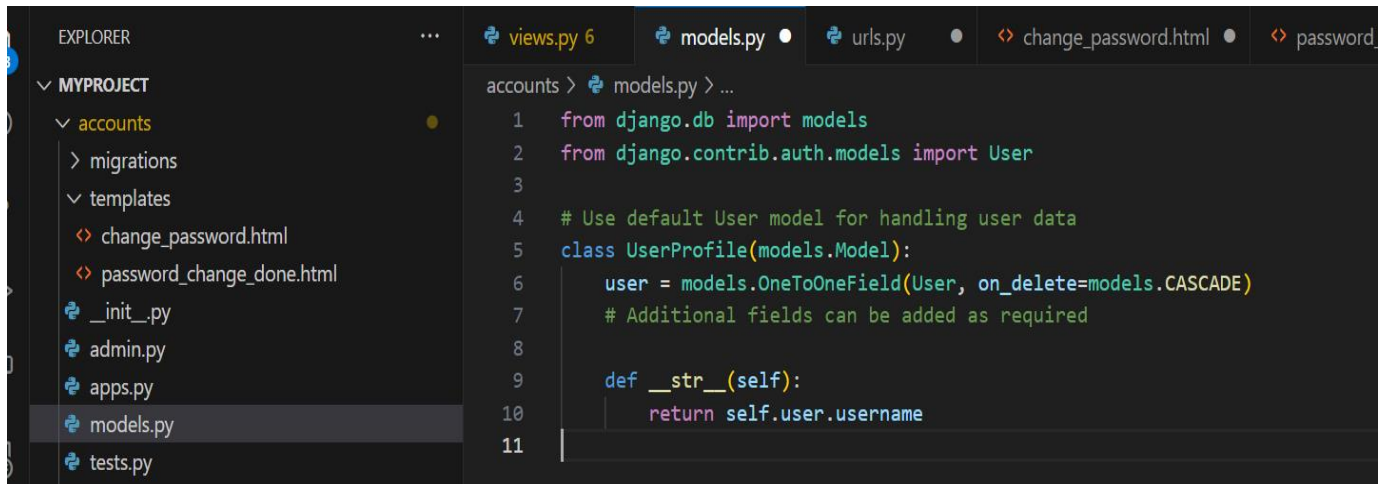
check if the current user's ID (`request.user.id`) matches the user ID stored in the session (`request.session.get('_auth_user_id')`). If they do not match, we return a 403 Forbidden response, thus preventing session hijacking.



```
... views.py x urls.py change_password.html password_change_done.html
accounts > views.py > ...
1 from django.contrib.auth import authenticate
2 from django.http import HttpResponseRedirect
3
4 @login_required
5 def change_password(request):
6     if request.user.id != request.session.get('_auth_user_id'):
7         return HttpResponseRedirect("Unauthorized request")
8
9     if request.method == 'POST':
10        form = PasswordChangeForm(request.user, request.POST)
11        if form.is_valid():
12            form.save()
13            update_session_auth_hash(request, form.user) # Important to keep the user logged
14            return redirect('password_change_done')
15        else:
16            form = PasswordChangeForm(request.user)
17        return render(request, 'change_password.html', {'form': form})
18
```

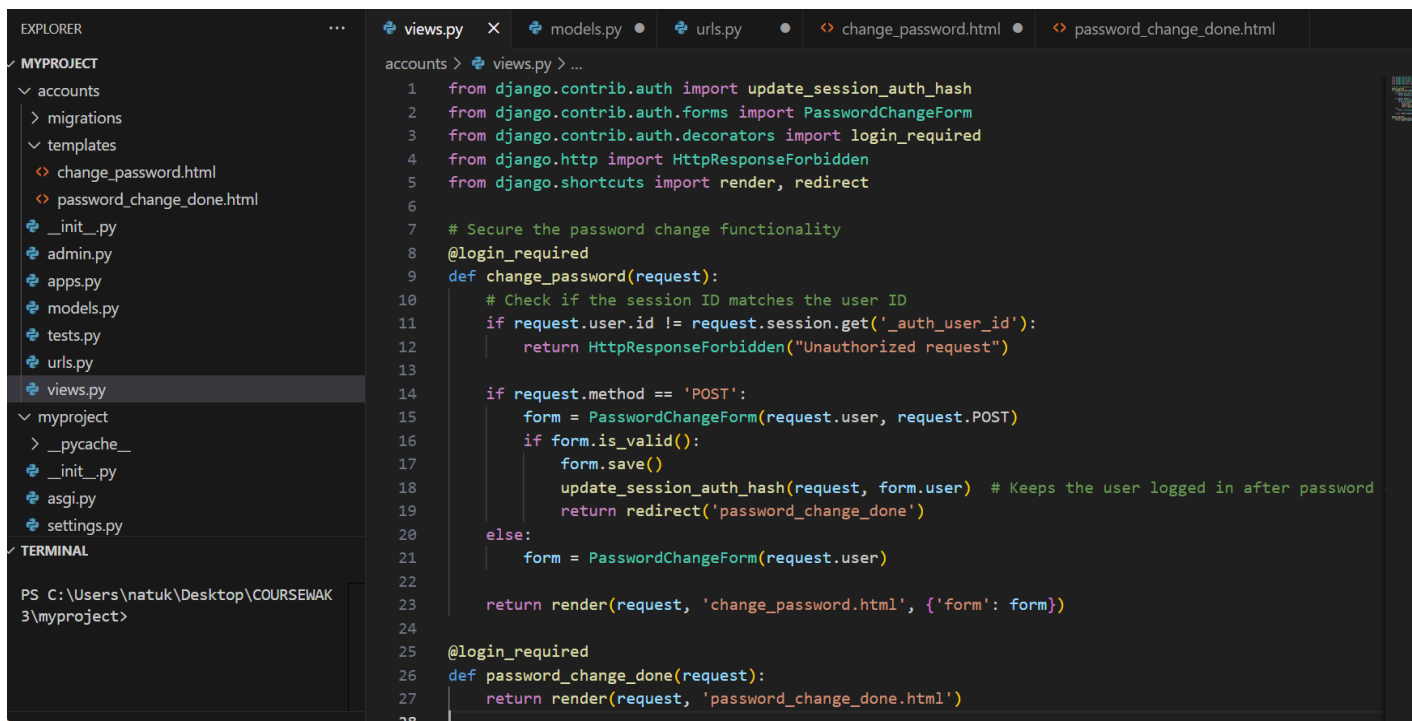
Code Snippet to Prevent Unauthorized Access in Django MVC

**focus on ensuring the model supports secure data handling.**



```
accounts > models.py > ...
1  from django.db import models
2  from django.contrib.auth.models import User
3
4  # Use default User model for handling user data
5  class UserProfile(models.Model):
6      user = models.OneToOneField(User, on_delete=models.CASCADE)
7      # Additional fields can be added as required
8
9      def __str__(self):
10         return self.user.username
11
```

**check if the session ID matches the authenticated user and prevent unauthorized actions like changing the password.**



```
accounts > views.py > ...
1  from django.contrib.auth import update_session_auth_hash
2  from django.contrib.auth.forms import PasswordChangeForm
3  from django.contrib.auth.decorators import login_required
4  from django.http import HttpResponseRedirect
5  from django.shortcuts import render, redirect
6
7  # Secure the password change functionality
8  @login_required
9  def change_password(request):
10     # Check if the session ID matches the user ID
11     if request.user.id != request.session.get('_auth_user_id'):
12         return HttpResponseRedirect("Unauthorized request")
13
14     if request.method == 'POST':
15         form = PasswordChangeForm(request.user, request.POST)
16         if form.is_valid():
17             form.save()
18             update_session_auth_hash(request, form.user) # Keeps the user logged in after password
19             return redirect('password_change_done')
20     else:
21         form = PasswordChangeForm(request.user)
22
23     return render(request, 'change_password.html', {'form': form})
24
25 @login_required
26 def password_change_done(request):
27     return render(request, 'password_change_done.html')
28
```

**Controller/URL (urls.py):**

```
accounts > urls.py > ...
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('change-password/', views.change_password, name='change_password'),
6     path('password-change-done/', views.password_change_done, name='password_change_done'),
7 ]
8
```

No changes are required in the template as the form to change the password remains the same, but it will now be protected by the session validation logic in the view.

```
accounts > templates > change_password.html > ...
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Change Password</title>
7 </head>
8 <body>
9     <h1>Change Password</h1>
10    <form method="POST">
11        {% csrf_token %}
12        {{ form.as_p }}
13        <button type="submit">Change Password</button>
14    </form>
15 </body>
16 </html>
17
```

confirms the successful password change.

```
accounts > templates > password_change_done.html > ...
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Password Changed</title>
7 </head>
8 <body>
9     <h1>Your password has been changed successfully!</h1>
10 </body>
11 </html>
12
```

## Steps Taken in the MVC to Mitigate Vulnerability:

1. **Model:** Ensured that the user data is securely stored, with no changes needed to address the vulnerability directly.
2. **View:** Implemented the check for session ID validation (`request.user.id` matches `request.session.get('_auth_user_id')`) to ensure the request is from the correct user.
3. **Controller:** Directed the user to the correct views for password change and confirmation, enforcing the view's security logic.
4. **Template:** Remains unchanged, but is now secure because it is protected by the view's session validation logic.

## Recent Trends in Programming Paradigms

**Functional Programming (FP)** emphasizes **immutability** and **pure functions**, improving **predictability** and reducing side effects, making it ideal for reliable and scalable systems.

**Object-Oriented Programming (OOP)** is increasingly used with **microservices** to create **modular** and **reusable** components, enabling better **scalability** and **maintainability**.

**Declarative Programming** is gaining traction, especially in **cloud computing** and **automation**, allowing systems to be described by their desired state, enhancing **scalability** and **automation**.

**Reactive Programming** enables **asynchronous** and **event-driven** systems, enhancing **responsiveness** and **performance** in applications dealing with real-time data.

### Leveraging OOP for Modularity, Scalability, and Performance

The team should leverage **Object-Oriented Programming (OOP)** to structure the application into **modular** classes for reusability, **scalability** through **inheritance** and **polymorphism**, and **performance** by optimizing memory and process management in a large-scale system.



Briefly describe at least eight secure design principles that you will use to evaluate and arrive at the security requirements and the threats that are most likely to affect your application.

**Least Privilege:** Limit user access to only what's necessary for their role to reduce exposure. This minimizes the risk of unauthorized actions.

**Defense in Depth:** Implement multiple layers of security to reduce the chance of a single breach compromising the system. Each layer adds another barrier for attackers.

**Minimize Attack Surface:** Limit the number of exposed services and interfaces to reduce potential points of attack. This makes it harder for attackers to find vulnerabilities.

**Separation of Duties:** Split responsibilities to ensure no single user or component has full control. This reduces the risk of malicious actions or errors.

**Input Validation:** Validate and sanitize all inputs to ensure they conform to expected types. This prevents malicious data from entering the system and causing harm.

**Authentication and Session Management:** Use secure authentication methods and protect session data. This prevents unauthorized access and session hijacking.

**Audit and Monitoring:** Continuously monitor and log activities to detect suspicious behavior. Early detection allows quick responses to potential security threats.

**Fail Securely:** Ensure the system defaults to a secure state during failures to prevent exposing vulnerabilities. This avoids unintended security risks from errors.

One should identify and describe at least four potential threats and weaknesses to this application after evaluation of the design principles. The important thing to remember is that in order to be useful, principles must be evaluated, interpreted, and applied to address specific problems.

1. **Session Hijacking:** If an attacker gains access to a user's session ID, they can impersonate the user and perform unauthorized actions. This can occur when session tokens are not properly validated
2. **Cross-Site Scripting (XSS):** If user input is not properly sanitized, malicious scripts can be injected into the system. These scripts can execute on the client side, compromising the user's session or allowing attackers to steal authentication tokens, escalate privileges,
3. **Insecure URL Handling:** If session IDs or other sensitive data are passed via URLs, attackers can exploit this by copying and using URLs that contain session tokens or user-specific data. This can grant unauthorized access or allow attackers to change system settings or perform actions with elevated privileges.
4. **Insufficient Access Controls:** If access controls are not well defined or incorrectly implemented, users could access parts of the application they should not be able to. This can lead to unauthorized changes to critical system settings, such as altering administrative passwords, viewing confidential data, or altering user permissions.

## h)Session Hijacking

Session hijacking occurs when an attacker steals a valid session token (e.g., from cookies or the URL) and impersonates a legitimate user, gaining unauthorized access to the application.

```
4
5 # Vulnerable code that does not secure the session cookie
6 # Django automatically stores session IDs in cookies
7 # But if session cookies are not marked as HttpOnly or Secure, they can be hijacked
8
9 # settings.py (Vulnerable configuration)
10 SESSION_COOKIE_SECURE = False # This should be True to use secure cookies
11 CSRF_COOKIE_SECURE = False   # This should also be True for CSRF protection
12
```

### Mitigation (Fix):

To prevent session hijacking, ensure that the session cookies are marked as Secure and HttpOnly to make them harder to steal through JavaScript or via insecure connections.

```
# settings.py (Secured configuration)
SESSION_COOKIE_SECURE = True # Ensures cookies are only sent over HTTPS
CSRF_COOKIE_SECURE = True   # Ensures CSRF cookie is only sent over HTTPS
SESSION_COOKIE_HTTPONLY = True # Prevents JavaScript access to cookies
```

## Cross-Site Scripting (XSS)

XSS occurs when an attacker injects malicious scripts into web pages viewed by other users. This can lead to session hijacking, stealing personal information, or defacing the site.

```
0 <!-- Vulnerable template where user input is rendered without escaping -->
1 <p>Welcome, {{ username }}</p> <!-- If username contains malicious JavaScript, it will run -->
2
```

Fix

```
<!-- Safe code where user input is escaped -->
<p>Welcome, {{ username|escape }}</p> <!-- Django automatically escapes user input -->
```

## SQL Injection

SQL Injection occurs when an attacker is able to manipulate an SQL query by inserting malicious input, potentially gaining unauthorized access or manipulating data in the database.

The attacker exploits vulnerable SQL queries that concatenate user input directly into the SQL command.

Code Snippet (Vulnerable Code):

```
cursor = connection.cursor()
cursor.execute("SELECT * FROM users WHERE username = '" + request.GET['username'] + "' AND password = '" + request.GET['password'] + "'")
```

### Mitigation (Fix):

use Django's ORM or parameterized queries to prevent SQL injection. Django ORM automatically handles user input safely.

```
user = User.objects.filter(username=request.GET['username'], password=request.GET['password']).first()
```

### Buffer Overflows

Buffer overflows occur when data exceeds the allocated buffer's size, overwriting adjacent memory, which can lead to arbitrary code execution.

This is primarily an issue in lower-level languages like C and C++, where user input can exceed the buffer limits.

### Code Snippet (Vulnerable Code)

```
user_input = request.POST.get('data', '')
if len(user_input) > 1000: # No check for input size, potential overflow risk
    process_data(user_input)
```

### Mitigation (Fix):

```
# Safe code with input validation
user_input = request.POST.get('data', '')
if len(user_input) > 1000:
    raise ValueError("Input is too large")
```

## Cross-Site Request Forgery (CSRF)

CSRF occurs when an attacker tricks a user into making a request to a website where the user is authenticated, causing actions to be performed without the user's consent.

The attacker can trick the user into clicking a malicious link or submitting a form, causing the server to perform an action as the authenticated user.

### Code Snippet (Vulnerable Code):

```
accounts > templates > <> change_password.html > ...
1  <!-- Vulnerable form without CSRF protection -->
2  <form action="/change_password/" method="POST">
3      <input type="text" name="password" value="newpassword">
4      <button type="submit">Change Password</button>
5  </form>
6
```

### Mitigation (Fix):

Django provides built-in CSRF protection, which is enabled by default. use {% csrf\_token %} in your f

```
1  <!-- Safe form with CSRF protection -->
2  <form action="/change_password/" method="POST">
3      {% csrf_token %}
4      <input type="text" name="password" value="newpassword">
5      <button type="submit">Change Password</button>
6  </form>
7
```

## six attack techniques

### Session Hijacking

- **Secure Cookies:** Use Secure, HttpOnly, and SameSite attributes.
- **Session Expiry:** Set short session lifetimes and regularly regenerate session IDs.

### Cross-Site Scripting (XSS)

- **Input Sanitization:** Sanitize and validate user input.
- **Content Security Policy (CSP):** Limit allowed sources for scripts.

### SQL Injection

- **Parameterized Queries:** Use placeholders in database queries.
- **Use ORM:** Leverage ORM frameworks to safely handle database interactions.

### Buffer Overflow

- **Limit Input Size:** Restrict the length of inputs.
- **Use Safe Libraries:** Rely on libraries that handle memory safely.

### Cross-Site Request Forgery (CSRF)

- **CSRF Tokens:** Require unique tokens with each user request.
- **SameSite Cookies:** Set cookies to SameSite to prevent cross-origin requests.

J )In the scenario above, the organization seeks to analyze customer trends over time to optimize its marketing strategies. Using data from your group's application, which addresses the chosen problem, write Python code that visualizes this data upon input using a suitable visualization library. (4 Marks)

```
accounts > views.py > ...
1  from django.shortcuts import render
2  from django.contrib.auth.decorators import login_required
3  from .models import LandAdvice
4  import matplotlib.pyplot as plt
5  import base64
6  from io import BytesIO
7  import pandas as pd
8
9  # Data visualization view for customer trends over time
10 @login_required
11 def visualize_trends(request):
12     query = request.GET.get('query', '') # User inputs a query to filter data (e.g., "Land Ownershi
13     start_date = request.GET.get('start_date', '2023-01-01')
14     end_date = request.GET.get('end_date', '2023-12-31')
15
16     # Filter data based on the user's query and date range
17     data = LandAdvice.objects.filter(
18         user_query__icontains=query,
19         date__range=[start_date, end_date]
20     )
21
22     # Prepare data for visualization
23     if data.exists():
24         # Convert data to DataFrame for time-based aggregation
25         df = pd.DataFrame(list(data.values('date', 'user_query')))
26         df['date'] = pd.to_datetime(df['date'])
27         trend_data = df.groupby('date').size() # Group by date to count occurrences
```

```

# Plotting the trend over time
plt.figure(figsize=(10, 5))
trend_data.plot(kind='line', marker='o', color="skyblue")
plt.title(f"Trend of '{query}' Queries Over Time")
plt.xlabel("Date")
plt.ylabel("Number of Queries")

# Save plot to buffer
buffer = BytesIO()
plt.savefig(buffer, format='png')
buffer.seek(0)
image_png = buffer.getvalue()
buffer.close()
plt.close()

# Encode image to base64 to render in HTML
graphic = base64.b64encode(image_png).decode('utf-8')
return render(request, 'visualize_trends.html', {'graphic': graphic, 'query': query})

else:
    # No data for the input query or date range
    return render(request, 'visualize_trends.html', {'error': 'No data found for the specified query or date range.'})
```



k) Given that the software has undergone recent updates, the team needs to verify that no new bugs have been introduced into existing features. Identify and Describe four types of regression tests that should be conducted to validate the system's stability.

Additionally, provide a simple Python code snippet to demonstrate how automated regression testing will be implemented effectively within your computational system

**(8 Marks)**

**Unit Regression Testing:** Focuses on testing individual components or functions to verify that updates haven't impacted isolated functionalities. Unit tests are often isolated and independent of external dependencies.

**Partial Regression Testing:** Verifies that code changes haven't negatively impacted related modules. This involves testing specific modules that are most likely affected by recent changes.

**Complete Regression Testing:** Examines the entire application to ensure that updates have not introduced any bugs across any of the functionalities. This is typically done when a major update affects many areas of the code.

**Corrective Regression Testing:** Used when no significant changes are made to the application's existing features. It checks if previously found and corrected bugs remain fixed.

l) Given that the software company aims to build a maintainable platform. Describe three key code programming practices that can be applied to support code maintainability, readability, and scalability in rapidly evolving large-scale projects.

### **Modularization and Code Reusability**

- Break code into self-contained modules and reusable functions, allowing independent updates and debugging.

### **Consistent Coding Standards and Documentation**

- Follow naming conventions and use clear comments and documentation. This makes the code easy for team members to read and maintain.

### **Scalability via Design Patterns and Efficient Data Handling**

Use design patterns like MVC (Model-View-Controller) and optimize data handling to support growth. This helps keep the code adaptable for future needs without major rewrites.

### **Code Reviews and Pair Programming**

Regularly review code with peers to catch issues early and ensure consistency.

### **Automated Testing and Continuous Integration**

Implement unit tests, integration tests, and CI pipelines to quickly identify issues with new changes

### **Use of Version Control and Branching Strategies**

Leverage tools like Git for version control and adopt clear branching practices (e.g., feature, development, and main branches) to manage code changes efficiently.

### **Dependency Management**

- Keep external libraries up-to-date and pin versions to avoid compatibility issues, ensuring the codebase remains stable as it scales.

m) Based on the application from your group idea briefly describe five security components that are found in the Software Requirements Specifications Document

**User Authentication;** Verifies user identity through secure methods like passwords, tokens, or multi-factor authentication.  
Ensures only verified users can access the application.

**User Authorization** Controls user access to specific data and functionalities based on roles (e.g., client vs. legal advisor).

Limits actions users can perform, protecting sensitive legal information.

**Data Encryption** Encrypts data both in storage and during transmission to protect sensitive legal and personal data.

**Audit Logging;**Records user actions, including queries and data retrieval, for monitoring and security auditing.

n) Use the methods showcased below appropriately with the help of diagrams to briefly describe how you shall identify four critical threats in the design phase of the application modules in in the extract above that need to be investigated and analyzed to identify the RISK associated with these critical areas (5 Marks@)

**(i) DREAD**

**(ii) Threat trees**

## **DREAD**

DREAD is a risk assessment model used to quantify, compare and prioritize the amount of risk presented by each evaluated threat. The five components of DREAD are:

**Damage Potential:** How much damage can be done if the threat is realized?

**Reproducibility:** How easy is it to reproduce the attack?

**Exploitability:** How easy is it to exploit the threat?

**Affected Users:** How many users are affected if the threat is realized?

**Discoverability:** How easy is it to discover the threat?

By assessing each of these components, you can assign a score to each threat and determine which ones are the most critical.

### (i) DREAD Analysis Diagram

For a security vulnerability like **session hijacking**

how the DREAD model can evaluate the risk factors. This analysis helps prioritize mitigation strategies by scoring the severity of each threat

### DREAD Analysis Diagram

Threat	Damage potential	Reproducibility	Exploitability	Affected Users	Discoverability	Total score
Session Hijacking	8	9	7	8	6	38
Cross-site scripting	7	8	6	8	7	36
Sql injection	9	8	8	10	8	43
CSRF	7	8	5	6	8	34

Each score ranges from 1 to 10, with higher scores representing greater risk. The overall scores highlight **SQL Injection** and **Session Hijacking** as priority threats due to their high impact and exploitability

## (ii) Threat Trees

Threat trees are used to identify and analyze potential threats to a system. The root of the tree is a primary goal that an attacker might want to achieve (e.g., gain unauthorized access), and the branches represent different ways the attacker might try to achieve that goal.

### Threat Tree Analysis Diagram

**Threat Trees** illustrate the steps leading to each threat, showing possible points of failure and ways attackers might exploit vulnerabilities.

#### Threat Tree for Session Hijacking

