**Cyber Security 2025 – Mooc.fi – Project I**

**LINK:** <https://github.com/nilofeliu/mooc_cyber_security_base_2025_project_01.git>

**Acknowledgments** This project builds upon the blog application created by salmanfarsi01 (<https://github.com/salmanfarsi01/Basic_Django_User_Profile_-_-Dashboard.git>). Special thanks for providing the foundational code structure.

**Installation Instructions**

1. Clone the repository: git clone https://github.com/nilofeliu/mooc\_cyber\_security\_base\_2025\_project\_01.git
2. Navigate to the project directory: cd mooc\_cyber\_security\_base\_2025\_project\_01/mooc\_cybersec\_2025\_blog
3. Install required dependencies: pip install django pillow selenium
4. Run database migrations: python manage.py migrate
5. Create a superuser (optional): python manage.py createsuperuser
6. Start the development server: python manage.py runserver
7. Access the application at http://127.0.0.1:8000

**Project Structure**

* mooc\_cybersec\_2025\_blog/ - Main application with security vulnerabilities
* documentation/ - Contains helper files for testing vulnerabilities and screenshots demonstrating each flaw

**Overview** This project demonstrates five critical security vulnerabilities from the OWASP Top 10 2021 list, implemented in a Django web application. Each vulnerability includes both the flawed implementation and commented fixes to illustrate proper security practices. Screenshots in the documentation folder show before/after states of each vulnerability.

FLAW 1: Broken Access Control - OWASP 2021 A01

**Description** The first security vulnerability is a Broken Access Control flaw, specifically an Insecure Direct Object Reference (IDOR). This allows any authenticated user to access other users' private profiles and thoughts by manipulating URL parameters.

**Location in Code** File: users/views.py (user\_page function, lines 30-40)

@login\_required

def user\_page(request):

# FLAW 1: Broken Access Control - any user can view any user's profile

user\_id = request.GET.get('user\_id', request.user.id)

try:

target\_user = User.objects.get(id=user\_id)

profile, created = Profile.objects.get\_or\_create(user=target\_user)

except User.DoesNotExist:

target\_user = request.user

profile, created = Profile.objects.get\_or\_create(user=request.user)

# FIX (commented): Proper access control

# profile, created = Profile.objects.get\_or\_create(user=request.user)

File: users/templates/users/user\_page.html (profile display section)

<!-- FLAW 1: Shows target user's data instead of logged-in user -->

<img class="profile-pic" src="{{ target\_user.profile.profile\_picture.url }}">

<h2>Welcome, {{ target\_user.username }}!</h2>

<!-- FIX (commented): Should always show logged-in user -->

<!-- <img class="profile-pic" src="{{ user.profile.profile\_picture.url }}">

<h2>Welcome, {{ user.username }}!</h2> -->

**Demonstration**

Attack Vector 1 - Viewing another user's profile: • Log in as User 1 • Navigate to /users/profile/?user\_id=2 • Application displays User 2's profile picture, username, and private thoughts • User 1 successfully accesses unauthorized data

Attack Vector 2 - Systematic data harvesting: • While logged in as any user, visit /users/profile/?user\_id=3 • Then visit /users/profile/?user\_id=4, /users/profile/?user\_id=5 • Each request reveals different users' private information • Attacker can systematically enumerate all user profiles

**Security Impact** This leads to privacy breaches, data harvesting, and trust violations as users can systematically access all user profiles and private information.

**The Fix** The commented fix restricts access to only the authenticated user's data:

# FIX (commented): Proper access control

# profile, created = Profile.objects.get\_or\_create(user=request.user)

# thoughts = Thought.objects.filter(user=request.user).order\_by('-created\_at')

FLAW 2: Security Misconfiguration - OWASP 2021 A05

**Description** The second vulnerability is a Security Misconfiguration where Django runs with debug mode enabled, exposing sensitive application information through detailed error pages.

**Location in Code** File: core/settings.py (line 170)

DEBUG = True # FLAW 2: Debug mode enabled exposes sensitive information

# FIX (commented): DEBUG = False

**Demonstration**

Attack Vector 1 - Triggering debug error page: • Navigate to any non-existent URL like /users/nonexistent-page/ • Django displays detailed error page • Page reveals all URL patterns, file paths, and internal configuration • Attacker gains complete application structure knowledge

Attack Vector 2 - Causing application errors: • Visit /fake-endpoint/ or any invalid route • Error page shows Django version, installed apps, and environment details • Information includes routing logic and available endpoints • Provides reconnaissance data for further attacks

**Security Impact** This provides attackers with complete application mapping, technology stack information, file system details, and reconnaissance data useful for planning sophisticated attacks.

**The Fix**

# FIX (commented): DEBUG = False

DEBUG = False

FLAW 3: Injection - OWASP 2021 A03

**Description** The third vulnerability is SQL Injection where user input is directly concatenated into SQL queries without sanitization, allowing manipulation of database queries.

**Location in Code** File: users/views.py (flaw\_sql\_injection function, lines 82-92)

def flaw\_sql\_injection(request):

thoughts = None

if request.GET.get('id'):

# FLAW 3: SQL Injection - Direct string formatting without parameterization

thought\_id = request.GET.get('id')

with connection.cursor() as cursor:

query = f"SELECT id, text FROM users\_thought WHERE id = {thought\_id}"

cursor.execute(query)

thoughts = cursor.fetchall()

return render(request, 'users/thought.html', {'thoughts': thoughts})

# FIX (commented): Use parameterized queries

# def flaw\_sql\_injection(request):

# thoughts = None

# if request.GET.get('id'):

# thought\_id = request.GET.get('id')

# try:

# thought\_id = int(thought\_id) # Validate input

# with connection.cursor() as cursor:

# query = "SELECT id, text FROM users\_thought WHERE id = %s"

# cursor.execute(query, [thought\_id])

# thoughts = cursor.fetchall()

# except ValueError:

# thoughts = None # Handle invalid input

#

# return render(request, 'users/thought.html', {'thoughts': thoughts})

**Demonstration**

Attack Vector 1 - Basic SQL injection: • Visit /users/search-thought/?id=1 OR 1=1 • Application executes: SELECT id, text FROM users\_thought WHERE id = 1 OR 1=1 • The OR 1=1 condition bypasses the ID filter • Returns all thoughts in database instead of one specific thought

Attack Vector 2 - Testing with non-existent ID: • Navigate to /users/search-thought/?id=999 OR 1=1 • Even though ID 999 doesn't exist, the OR 1=1 condition is always true • Query returns all user thoughts, confirming SQL injection vulnerability • Demonstrates how attackers can extract data regardless of valid input

**Security Impact** This enables data breaches, unauthorized access to all user content, database enumeration, and potential authentication bypass through SQL manipulation.

**The Fix** The fix implements parameterized queries with input validation:

# Use parameterized query with %s placeholder

query = "SELECT id, text FROM users\_thought WHERE id = %s"

cursor.execute(query, [thought\_id]) # Pass parameters safely

FLAW 4: Insecure File Upload - OWASP 2021 A05 (Security Misconfiguration)

**Description** The fourth vulnerability is Insecure File Upload allowing users to upload any file type through profile picture functionality, instead of restricting to safe image formats.

**Location in Code** File: users/models.py (Profile model, line ~8)

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

# FLAW 4: FileField allows any file type to be uploaded

profile\_picture = models.FileField(default='default.jpg', upload\_to='profile\_pics')

# FIX (commented): Proper image validation

# profile\_picture = models.ImageField(default='default.jpg', upload\_to='profile\_pics')

**Demonstration**

Attack Vector 1 - Malicious HTML file upload: • Create malicious HTML file (available at /documentation/helper files/malicious.html):

<script>alert('Malicious File Uploaded Successfully!');</script>

• Upload file through profile picture form • Access file directly at http://127.0.0.1:8000/media/profile\_pics/malicious.html • JavaScript executes in browser showing alert popup

Attack Vector 2 - PHP shell script upload: • Create PHP shell script (available at /documentation/helper files/shell.php):

<?php echo "Server compromised!"; ?>

• Upload PHP file through profile picture functionality • Access at http://127.0.0.1:8000/media/profile\_pics/shell.php • File is served as downloadable content (could execute if PHP enabled on server)

**Security Impact** This enables Cross-Site Scripting (XSS), potential remote code execution, malware distribution, server storage abuse, and social engineering attacks.

**The Fix** Use Django's ImageField with built-in validation:

# FIX (commented): Proper image validation

# profile\_picture = models.ImageField(default='default.jpg', upload\_to='profile\_pics')

FLAW 5: Cross-Site Scripting (XSS) - OWASP 2021 A03

**Description** The fifth vulnerability is Stored XSS allowing malicious users to inject client-side scripts that execute when other users view the affected pages.

**Location in Code** File: users/templates/users/user\_page.html (lines 77 – 82)

{% for thought in thoughts %}

<div class="thought">

<p>{{ thought.text|safe }}</p> <!-- VULNERABLE: |safe filter disables HTML escaping -->

<small>{{ thought.created\_at|date:"F d, Y, P" }}</small>

</div>

{% empty %}

**Demonstration**

Attack Vector 1 - Basic XSS payload: • Navigate to user profile page • Submit thought containing: <script>alert('XSS Attack!')</script> • Page reloads and immediately displays JavaScript alert popup • Any user viewing this page will see the same malicious popup

Attack Vector 2 - Alternative XSS payloads: • Submit thought with: <img src=x onerror=alert('XSS!')> • Or use: <svg onload=alert('XSS')> • Both payloads execute JavaScript when page loads • Demonstrates multiple ways to exploit the |safe filter vulnerability

**Security Impact** This enables session hijacking, account takeover, data theft, malware distribution, and social engineering through fake forms or messages.

**The Fix** Remove the |safe filter to enable Django's default HTML escaping:

{% for thought in thoughts %}

<div class="thought">

<p>{{ thought.text }}</p> <!-- FIXED: Removed |safe filter, Django auto-escapes by default -->

<small>{{ thought.created\_at|date:"F d, Y, P" }}</small>

</div>

{% empty %}

**Prevention Best Practices**

* Never trust user input - treat all user-generated content as potentially malicious
* Use Django's default escaping - only use |safe with trusted content
* Implement proper input validation and parameterized queries
* Use appropriate field types with built-in security features
* Disable debug mode in production environments
* Regular security testing during development and production