LINK: NikiPOU/Cyber-Security-Project-1: For Cyber Security Project 1 course.

INSTRUCTIONS:

1. git clone https://github.com/NikiPOU/Cyber-Security-Project-1.git
2. cd personal_blog
3. python3 manage.py runserver

This project demonstrates 5 common vulnerabilities from the OWASP Top Ten 2017 list, and their fixes. For this project, I created a simple Django based blog application with basic functionality such as adding and deleting posts. Each flaw below is described with the source code link, an explanation of why it is dangerous, along with the fix.

FLAW 1: A5:2017-Broken Access Control

Source link:

Example 1:

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/views.py#L20

Example 2 (IDOR):

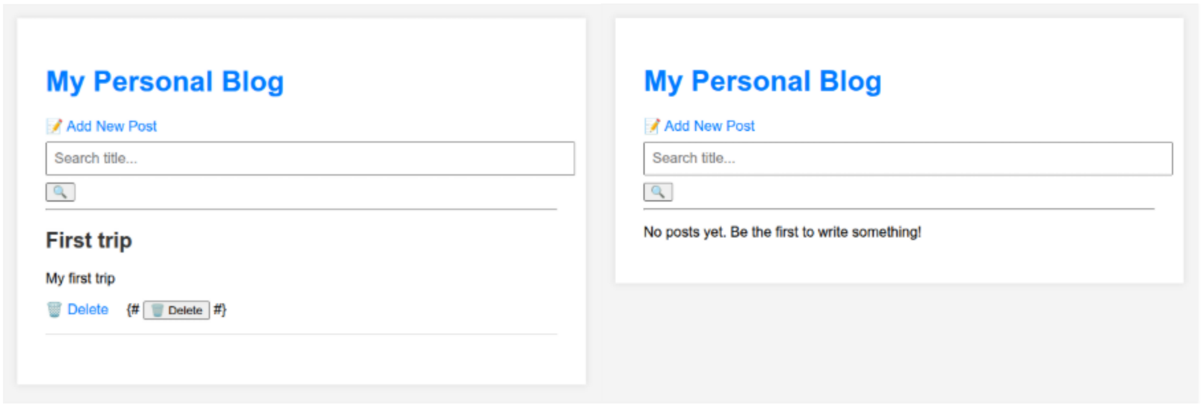
https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L6

Description:
Broken Access Control occurs when an application fails to properly restrict what authenticated or unauthenticated users are allowed to do. Here, the delete_post view allows anyone to delete a post just by visiting /delete/<post_id>/. There is no authentication, no session check, and no ownership verification. This means that an arbitrary visitor could delete all posts in the blog by, for example, guessing post IDs.

The project additionally suffers from an Insecure Direct Object Reference (IDOR). Attackers can manipulate the URL to delete posts that belong to others, making our app especially susceptible to this, since the post ID's are sequential.

Broken access control is one of the most common and damaging flaws in real life scenarios. If attackers can bypass restrictions, they can access data or perform actions which they shouldn't be able to otherwise. This can lead to unauthorized data leakage and mass deletion of records.

Demonstration of example 1:

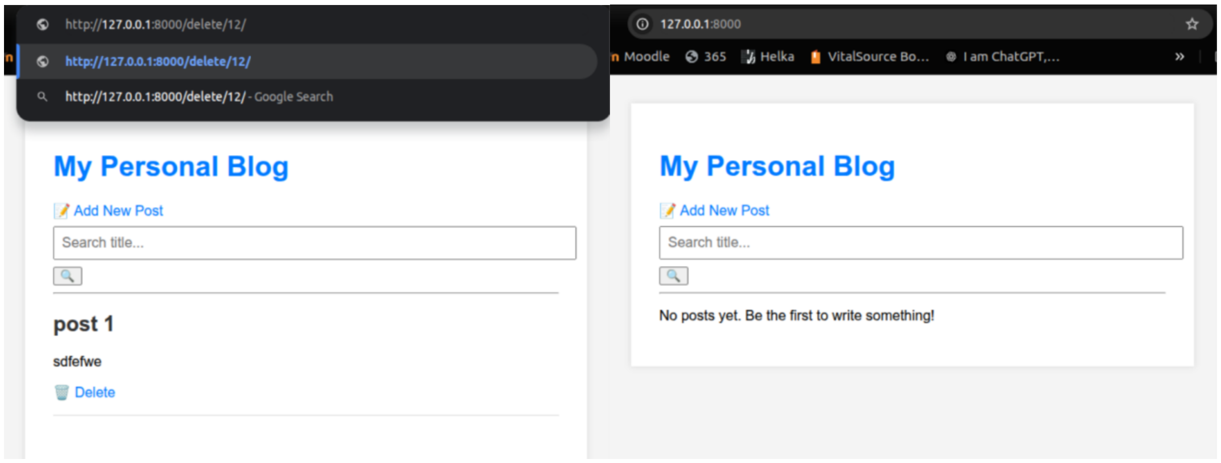


First trip

My first trip

No posts yet. Be the first to write something!

Demonstration of example 2 (IDOR):



http://127.0.0.1:8000/delete/12/

http://127.0.0.1:8000/delete/12/

http://127.0.0.1:8000/delete/12/ - Google Search

My Personal Blog

📝 Add New Post

Search title...

post 1

sdfefwe

🗑️ Delete

127.0.0.1:8000

My Personal Blog

📝 Add New Post

Search title...

No posts yet. Be the first to write something!

FIX:

For example 1: Add if statement to check request method.

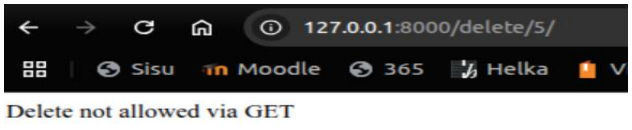
https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/views.py#L23

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/views.py#L24

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/views.py#L27

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/views.py#L28

After fix:



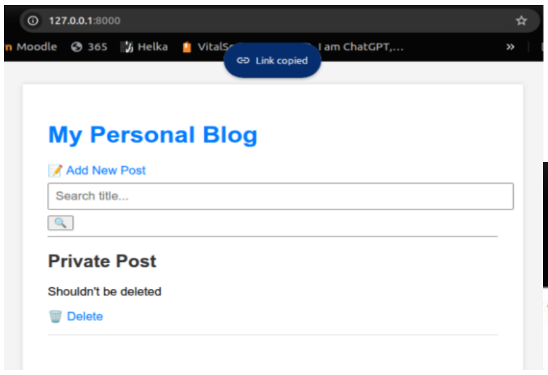
127.0.0.1:8000/delete/5/

```
Delete not allowed via GET
```

For example 2 (IDOR): Allow deletion only if by post owner (admin).

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L12

After fix:


127.0.0.1:8000

My Personal Blog

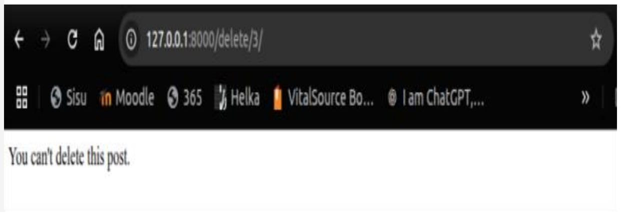
📝 Add New Post

Search title...

Private Post

Shouldn't be deleted

🗑️ Delete


127.0.0.1:8000/delete/3/

You can't delete this post.

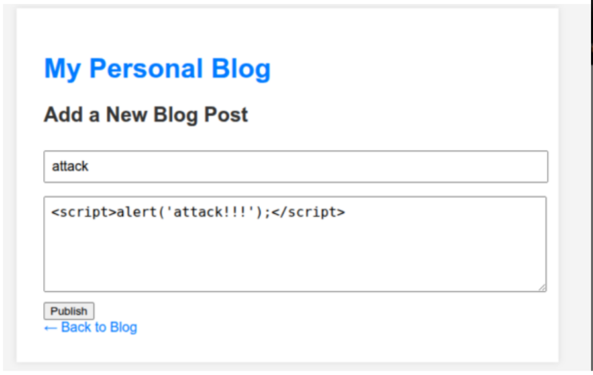
FLAW 2: A7:2017-Cross-Site Scripting (XSS)

Source link:

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/templates/home.html#L14

Description:

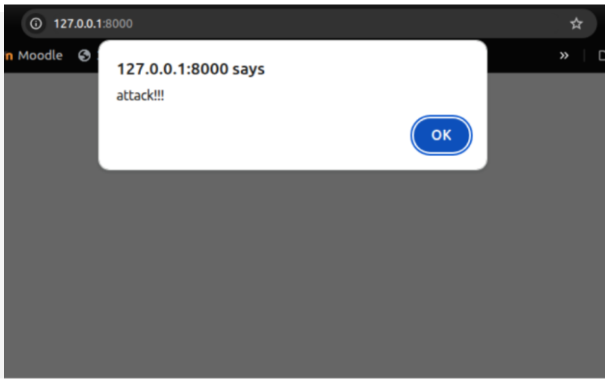
XSS occurs when user input is rendered directly into HTML without escaping. In this project, post content is displayed inside the template without filtering. This means an attacker can create a blog post with malicious JavaScript, such as:


My Personal Blog

Add a New Blog Post

attack

```
<script>alert('attack!!!');</script>
```



127.0.0.1:8000 says

attack!!!

XSS is one of the most prevalent vulnerabilities on the web. Attackers can use it to steal session cookies, perform actions on behalf of users, or redirect victims to malicious websites.

FIX: Ensure user input properly escaped (explicitly active for the post content), making HTML tags show as plain text instead of executing.

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/templates/home.html#L16

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/templates/home.html#L18

FLAW 3: A6:2017-Security Misconfiguration

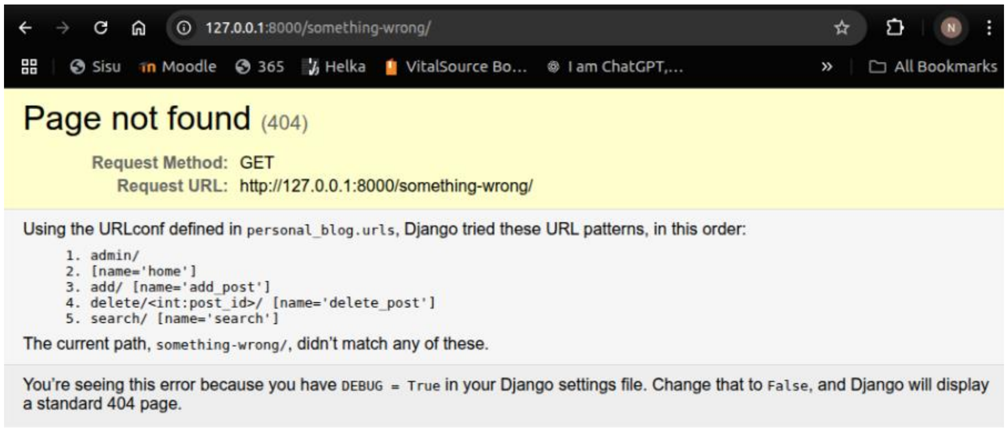
Source link:

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/personal_blog/settings.py#L29

Description:

Security misconfiguration refers to insecure default settings, unnecessary features enabled, or poor configuration of security headers. Here, the misconfiguration was leaving DEBUG = True in production. With this setting enabled, Django provides tedious error pages with system details (intended for development) whenever an invalid link is visited.

The error pages reveal sensitive information such as file paths, configuration details, and environment variables. Attackers can use this information for exploits.



Page not found (404)

Request Method: GET
Request URL: http://127.0.0.1:8000/something-wrong/

Using the URLconf defined in `personal_blog.urls`, Django tried these URL patterns, in this order:

```
1. admin/
2. [name='home']
3. add/ [name='add_post']
4. delete/<int:post_id>/ [name='delete_post']
5. search/ [name='search']
```

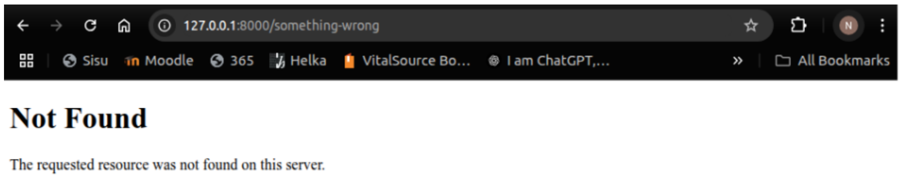
The current path, `something-wrong/`, didn't match any of these.

You're seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

FIX: Disable debug mode before deployment (change Debug = True to Debug = False).

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/personal_blog/settings.py#L31

After fix:



Not Found

The requested resource was not found on this server.

FLAW 4: A3 Sensitive Data Exposure

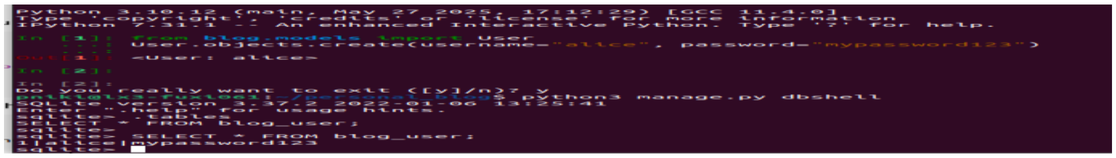
Source link:

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L19

Description:

Sensitive data exposure occurs when applications fail to protect sensitive information such as passwords, credit card numbers, or personal information. Here, I originally stored passwords in plain text inside the database (for example, creating a user with password="mypassword123" meant the string was saved exactly as typed).

Storing passwords as plaintext is a massive security flaw. If the database is leaked, not only is every account in the application compromised, but since users often reuse passwords, attackers can then access accounts on other websites. This has been the cause of many known breaches.



```
Python 3.10.12 (main, May 27 2025, 17:12:29) [GCC 11.4.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from blog.models import User
   ...: User.objects.create(username="alice", password="mypassword123")
Out[1]: <User: alice>

In [2]:

In [2]:
Do you really want to exit ([y]/n)? y
pniki@LX3-fuxl061:~/personal_blog$ python3 manage.py dbshell
SQLite version 3.37.2 2022-01-06 13:25:41
Enter ".help" for usage hints.
sqlite> .tables
SELECT * FROM blog_user;
sqlite>
sqlite> SELECT * FROM blog_user;
1|alice|mypassword123
sqlite> 
```

FIX: Instead of using plaintext, we use Djangos built in hashing, saving the password as a long hash.

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L25

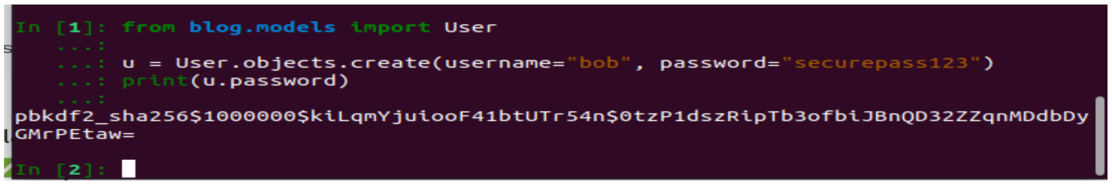
https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L26

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L27

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L28

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/blog/models.py#L29

After fixes, password is saved as hash:



```
In [1]: from blog.models import User
   ...:
   ...: u = User.objects.create(username="bob", password="securepass123")
   ...: print(u.password)
   ...:
pbkdf2_sha256$1000000$kiLqmYjuiooF41btUTr54n$0tzP1dszRipTb3ofbiJBnQD32ZZqnMDdbDy
GMrPEtaw=

In [2]: 
```

FLAW 5: A9-Using Components with Known Vulnerabilities

Source link:

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/requirements.txt#L4

Description:

When using external libraries and frameworks for applications, it is important to check if these dependencies are outdated or vulnerable, putting the entire application is at risk. In this project, I demonstrated the flaw by intentionally using Django 2.2.0, which is outdated and contains multiple known CVEs. I also pinned requests==2.19.0, another package with documented vulnerabilities.

These flaws can be exploited. Since public databases maintain lists of vulnerabilities, using outdated components is extremely irresponsible and risky.

FIX: Updating these outdated dependencies to latest versions to fix the issue and eliminate known vulnerabilities.

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/requirements.txt#L7

https://github.com/NikiPOU/Cyber-Security-Project-1/blob/main/requirements.txt#L8

All in all, this project has opened my eyes to how easy it is to unintentionally mess up and put your whole project at security risk, even in a simple app. Just from these five flaws: broken access control, XSS, misconfigurations, plain-text passwords, and outdated libraries, you can see how something small can turn into a big problem. Fixing them wasn't too complicated, but it makes you realize that you can't just write code and hope for the best. Hopefully, after doing this, the blog app is at least a bit safer and won't get hacked the first time someone tries. My main takeaway is to: keep the application up to date, double-check my settings, and add small changes to the app that make a huge difference in the long run.