LINK: https://github.com/NikiPOU/Cyber-Security-Project-1

INSTRUCTIONS:

1. git clone https://github.com/NikiPOU/Cyber-Security-Project-1.git
2. cd personal_blog
3. python 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:

https://github.com/NikiPOU/Cyber-Security-Project-1/blog/main/blog/views.py#L37

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

Screenshots: Flaw-1-Before.jpg, Flaw-1-After.jpg

Description:

Broken Access Control happens when an application does not properly restrict what users are allowed to do. In the vulnerable version of this project, the `delete_post` view allowed anyone to delete a post simply by visiting `/delete/<post_id>/`. There was no check to see if the user was logged in and no verification that the post belonged to the user making the request. Particularly, because the post IDs are sequential, an attacker could easily guess different IDs and delete multiple posts. This also created an Insecure Direct Object Reference (IDOR), since users could modify the URL to access and delete posts that were not theirs.

Fix:

To fix this issue, I added proper access control checks in the `delete_post` view. The application now first checks whether a user is logged in by verifying the session. Then it checks whether the logged-in user is actually the owner of the post before allowing deletion. If these conditions are not met, the application returns a forbidden response instead of deleting the post. This ensures that only authenticated users can delete posts, and only their own posts.

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

Source link:

https://github.com/NikiPOU/Cyber-Security-Project 1/blob/main/blog/templates/home.html#L28

Screenshots: Flaw-2-Before.jpg, Flaw-2-After.jpg

Description:

Cross-Site Scripting (XSS) occurs when user input is displayed in a web page without being properly escaped. In the vulnerable version of this project, post content was rendered using `{% autoescape off %}`, which disabled Django’s automatic HTML escaping. This meant that if a user created a post containing malicious JavaScript code, such as a `<script>` tag, the browser would execute it instead of displaying it as text. As a result, an attacker could inject JavaScript that triggers pop-ups, steals session information, or performs actions on behalf of other users. XSS is one of the most common web vulnerabilities and can have serious consequences, especially if sensitive data like session cookies are exposed.

Fix:

To fix this issue, I removed the `{% autoescape off %}` block from the template so that Django’s default autoescaping remains enabled. With autoescaping turned on, any HTML or JavaScript entered by a user is rendered as plain text instead of being executed by the browser. This prevents injection of scripts.After the fix, the application is no longer vulnerable to stored XSS attacks.

FLAW 3: A6:2017-Security Misconfiguration

Source link:

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

Screenshots: Flaw-3-Before.jpg, Flaw-3-After.jpg

Description:

Security Misconfiguration occurs when an application is deployed with insecure settings/ development features still enabled. In this project, the issue was that `DEBUG = True` was left enabled in the `settings.py` file. While debug mode is turned on, Django displays detailed error pages whenever something goes wrong These pages include sensitive technical information like file paths and configuration details. While this is useful during development, it should not be exposed in production. An attacker could use this information to better understand the application's structure and potentially identify weaknesses to exploit.

Fix:

To fix this issue, I changed the setting from `DEBUG = True` to `DEBUG = False` in the `settings.py` file. With debug mode disabled, Django no longer shows detailed error messages to users. Instead, it displays a generic error page that does not reveal internal system information.

FLAW 4: 3 Sensitive Data Exposure

Source link:

https://github.com/NikiPOU/Cyber-Security-Project-1/blog/main/blog/models.py#L2

https://github.com/NikiPOU/Cyber-Security-Project-1/blog/main/blog/models.py#L4

https://github.com/NikiPOU/Cyber-Security-Project-1/blog/main/blog/views.py#L70

https://github.com/NikiPOU/Cyber-Security-Project-1/blog/main/blog/views.py#L87

Screenshots: Flaw-4-Before.jpg, Flaw-4-After.jpg

Description:

Sensitive Data Exposure occurs when an application does not properly protect confidential information. In this project, user passwords were originally stored in plain text inside the database. So, if a user signed up with a password like "123", the exact string was saved directly in the database without any encryption or hashing. Storing passwords in plain text is a serious security flaw because if the database is ever leaked or accessed by an attacker, all user accounts would immediately be compromised. Additionally, since many users reuse passwords across different websites, this could lead to further problems.

Fix:

To fix this issue, I implemented Django's built-in password hashing instead of storing passwords as plain text. Now, when a user creates an account, the password is automatically converted into a secure hash before being stored in the database, and the actual password is never saved directly. During login, Django compares the entered password with the stored hash instead of comparing plain text values. This significantly improves security and protects user credentials even if the database is exposed.

FLAW 5: A9-Using Components with Known Vulnerabilities

Source link:

https://github.com/NikiPOU/Cyber-Security-Project-1/requirements.txt#L2

Screenshots: Flaw-5-Before.jpg, Flaw-5-After.jpg

Description:

This vulnerability occurs when an application relies on outdated libraries or frameworks that contain publicly known security issues. In this project, I demonstrated this flaw by intentionally using `Django==2.2.0`, which is an outdated and unsupported version with multiple documented CVEs. Since vulnerability databases are publicly available, attackers can easily identify outdated software and exploit known weaknesses. Running unsupported or outdated dependencies increases the overall risk of the application being compromised.

Fix:

To fix this issue, I updated the dependency in the `requirements.txt` file to its latest stable and supported version. After upgrading Django, the application now runs on maintained versions that receive regular security updates and patches.

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. Only from five flaws can you see how something small can turn into a big problem. Fixing them was not too complicated. Hopefully, after doing this, the blog app is at least a bit safer and won't get easily hacked. My main takeaway is to keep the application up to date, double-check my settings, and add small changes to the app overtime, preventing vulnerabilities in the long run.