**1. Cross-Site Scripting (XSS)**

- XSS allows an attacker to inject malicious JavaScript into a vulnerable web page. This code then gets executed in the victim's browser when they visit that page.

- To test for stored XSS, I submitted a comment containing `<script>alert('XSS')</script>` on a blog post. This was stored in the backend database.

- When other users visited that post, the script tag got rendered as-is instead of being encoded properly. This caused the alert popup to trigger in their browsers.

- The payload I used was simple, but attackers could craft more harmful XSS payloads to steal session cookies, deface websites, redirect to phishing pages etc.

- Proper output encoding like HTML entity encoding prevents the browser from executing the script. Encoding untrusted input before outputting it in pages mitigates XSS.

**2. SQL Injection**

- SQLi allows an attacker to interfere with backend SQL queries by injecting new conditions or commands.

- In the login form, I entered `' OR '1'='1` in the username field. This condition always evaluates to true.

- The original query was modified from `SELECT \* FROM users WHERE username = 'input' AND password = 'input'` to `SELECT \* FROM users WHERE username = '' OR '1'='1' AND password = 'input'`.

- Since 1=1 is always true, the query doesn't check for valid credentials and lets me log in as any user.

- Using SQLi an attacker can grab password hashes, extract database contents, delete/modify data and more.

- Input validation and parameterized queries prevent SQLi by separating data from commands.

**3. Local File Inclusion**

- LFI allows reading arbitrary files on the server by manipulating path values.

- I appended `../../../etc/passwd` to the PDF link to break out of the web root directory.

- This caused the password file to be included in the response instead of the intended PDF file.

- An attacker can leverage LFI to view source code, config files with passwords/keys, system logs etc.

- Validating file paths and restricting access prevents attackers from accessing unauthorized files.

Here are some more details on the default credentials and missing access control vulnerabilities:

**4.Default Credentials**

- Many applications use default or common credentials like 'admin/admin' during development. These often remain unchanged when the application gets deployed.

- I tried accessing the /manager/html path in the app, which is the Tomcat manager interface.

- This prompted me for credentials. I entered the default 'tomcat/tomcat' credentials which worked.

- With access to the manager console, an attacker can stop/start web apps, deploy malicious WAR files, access application config and more.

- Some other common default credentials are root/root, admin/blank-password, etc. Attackers have lists to try out on apps.

- It's critical to change default credentials before deploying applications. Regular credential rotation also helps mitigate this.

**5.Missing Access Control**

- The application had some unauthorized access to privileged functions.

- I was able to directly access the /admin section without logging in to the app first.

- Normal users should not have been able to access this admin functionality.

- This allowed me to view and edit sensitive configuration and data.

- Proper access control checks should be implemented on all restricted functionality like admin panels, account settings etc.

- The app should check if a user is logged in and authorized before granting access to privileged functions.