LINK: < <https://github.com/Henrikhi/NotSoSecureApplication>>

The whole project can be easily cloned or downloaded as a ZIP from github.

FLAW 1:

Broken authentication:

The application has a default admin user, with the username “admin” and password “admin”. This is a security flaw, because these types of default accounts should not be used. The password is far too simple, which makes it possible to log in to the application after fuzzing the username and password. And even worse, the account has admin authorities! So, with a simple fuzzing or just trying to log in with username “admin” and password “admin” gives the attacker all the admin rights in the website. This allows the attacker to go for example in the secret path “/adminpath”, which may contain some very important stuff that is only supposed to be accessed by admins.

This is a simple problem, and it has a simple solution. The admin’s password should be far more complicated that just “admin”. With a proper password, fuzzing becomes ineffective and the “/adminpath” path will stay untouched for a bit longer.

FLAW 2:

Sensitive Data Exposure

All data stored in the database is in plain text. This means that even the passwords are stored in plain text, without any type of encryption. This is a bad habit and makes the application vulnerable. All sensitive data should be always stored encrypted.

In this application, only two things are needed to make the passwords encrypted. First of all, the SecurityConfiguration class should have a PasswordEncoder Bean, and the authorization in the UserDetailsService should use that passwordEncoder. Second of all, when making a new account to the application, the Account’s password should be encoded with the PasswordEncoder before saving the account in the repository.

FLAW 3:

Broken Access Control

Attacker can access sites he is not supposed to access with simply modifying the URL. In the application, going to the “notes/{noteId}” site required the account to be logged in. However, this does not stop the account from peeking into other accounts notes as well! Every note in the database has its own unique id, but unfortunately the ids are integers starting from 1. Because of this, the attacker can easily access any notes he wants, with simply modifying the URL.

First of all, when trying to access a website “notes/{noteId}, the controller should check if the logged in user is allowed to access that note. This could be done in the controller by checking the authorized user now, and checking if that user matches user of the note. If the accounts match, the note belongs to the user and he is allowed to access the note. If the accounts do not match, the note should not be visible to the account. Also, the ids of the notes should be changed to for example UUID:s. This would make it harder to even access the site which controls the note.

FLAW 4:

Cross-Site Scripting (XSS)

The application is vulnerable for Cross-Site Scripting attacks. For example, the content of the notes is not checked in any way, and the attacker can insert malicious HTML or JavaScript code in the note. If the application allowed to for example send notes to different users, the receiver would fall victim in a Cross-Site Scripting attack when opening the note. In this way the attacked would be able to for example steal the session id of the victim and hijack his session.

There are several ways to prevent Cross-Site Scripting. There are several libraries, frameworks and other classes made to prevent Cross-Site Scripting which could be implemented and used in the application. Also, other ways to manually prevent Cross-Site Scripting would be to escape and validate user input and by sanitizing data. For example, when creating the note and the content, it would be possible to check if the content for example contains String “<SCRIPT>” and if so, to refuse to publish that note. Of course this would only prevent the Cross-Site Scripting attacks so far, and more robust actions should be taken, like enabling a Content Security Policy.

FLAW 5:

Using Components with Known Vulnerabilities

The application uses some old components which are known to have security vulnerabilities. This was noticed by running the Dependency-Check and studying the results. Of course, old and compromised plugins, dependencies or other components should not be used, because only one vulnerable component puts the whole application in risk. In the dependency check I ran, the application contained 18 vulnerable dependencies and 94 vulnerabilities in total. Actually this was kind of a shock for me, because I did not expect to find this many vulnerabilities in such a simple application.

The simplest way to tackle having components with vulnerabilities is to simply run different dependency tests regularly and checking if any compromised components are found. If expired components are found, they should be replaced with safer versions. Notice that not always the newest version is the safest, as sometimes new versions may contain some vulnerabilities that the older versions did not have. That is why the newest known safe and stable version should be chosen.