Cyber Security Project

Project can be found here :
https://github.com/ConstantKrieg/cybersecurityproject

It’s a normal Spring-boot application that can be opened from NetBeans and used from there.

THERE ARE TWO BRANCHES. MASTER IS THE ONE WITH FLAWS AND BRANCH 'CORRECT' HAS SOME KIND OF FIX TO THEM.

So for those who don't know branches can be changed with git checkout command. Easiest would be to first launch the application in the master branch and then reading through my flaws. In the correct branch you will be able to see fixes I've made so switch to that branch after exploring the app with flaws.

Branches can be changed from the project root like this:
git checkout master
git checkout correct

Step-by-step instructions using command line:

1. git clone https://github.com/ConstantKrieg/cybersecurityproject
2. cd CyberSecurityProject
3. mvn clean install
4. java -jar target/CyberSecurityProject-1.0-SNAPSHOT.jar

Application will now run on localhost:8080

I have created a simple web application where users can create an account, log in and add some tasks with name and some details which the application will then store in to a database. There are some security flaws in my application which I will now go through.

1. Broken Authentication and Session Management

Now passwords are not sent over an encrypted connection in account creation and in logging in. The javascript snippet just takes the credentials from the form and sends them to the back-end REST interface as they are. So for example someone who is listening the server where this application is running with for example WireShark, would see every password and username pair when users are logging in or registering. This can be seen by pressing F12 on Google Chrome before logging in and then choosing the network tab. Now when you press the log in button you'll see the password as plain text in the request that is sent to the backend.

This would need to be fixed with encryption and implementing HTTPS as the application layer protocol. In Spring this would mean that in SecurityConfiguration-class there would be an autowired config that assigns an encrypter to the UserDetailsService that the SecurityConfiguration is using.
Now from command line :

And you will be shown the fixed version.
This has been done by adding this to SecurityConfiguration:

```
@ Autowired
private AccountService accountService;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception
{    auth.userDetailsService(userDetailsService).passwordEncoder(accService.passwordEncoder());
}
```

and in the AccountService:

```
public String encodePassword(String password) {
      return passwordEncoder().encode(password);
   }

@Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }
```

In accountController there is now the encryption step before saving as you can see

To get HTTPS you'll need to apply for an SSL certificate for your server.

2. Insecure Direct Object References

This can be seen after logging in. When an user logs in the server will redirect it to its own page. The identification is done using the user's database id which can be seen in the URI. When the page loads javascript will then read the URI, split it into parts and find the id. It then makes a REST-call with that id to the backend. Because this is all client side it can be altered from the browser and another id may be entered in the URI field which leads to you seeing someone else's tasks.

This can be fixed by implementing a session that will have the current user stored and only that users data would be accessible. In controller these changes should be made.

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();

      Account acc = accountRepo.findByUsername(auth.getName());

      if (acc.getId().longValue() == id) {
         return "userPage";
      }
      return "redirect:/users/" + acc.getId();
```

3. Missing function level access control

The NoteController which is a REST-controller that handles only JSON formatted notes back and forth with the client does not have any access control. This means that it will save every task sent to the backend to database. Like said in previous flaw, the id is taken by the javascript from the URI so it's easy to change the id in the browser. This will then result to saving your task to someone else or you being able to retrieve someone else's tasks. This can be seen by logging in with your account, changing the id in the URI to something else and if another user's page is shown then adding a note from that page.

This could be fixed by again implementing a session which has the current user that is logged in stored. When and note is added it would be then confirmed in the backend that the user currently logged in is the same one that the tasks accountId refers to. This would also need to be implemented in the GET-functionality where the tasks are retrieved by their accountId.

This is fixed in the correct-branch the same way as the previous one. So by adding a check

4.Security misconfiguration

The most blatant one is when trying to save two users with the same username. After trying to add the second one a warning will show that this username already exists. This would be fine however when opening the the console with F12 and choosing the network tab you can see that the response is having a message like :

{"timestamp":1514557693240,"status":500,"error":"Internal Server Error","exception":"org.springframework.dao.DataIntegrityViolationException","message":"could not execute statement; SQL [n/a]; constraint [\"UK_GEX1LMAQPG0IR5G1F5EFTYAA1_INDEX_E ON PUBLIC.ACCOUNT(USERNAME) VALUES ('username', 1)\"; SQL statement: insert into account (id, password, username) values (null, ?, ?) [23505-193]]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement","path":"/users/reg"}

This shows the table in the database and the unique constraint key it has generated. Of course in this application there isn't much anyone could do with that but in general these kinds of flaws should be avoided

This could be done by modifying addAccount like this

```
try {
        accountRepo.save(account);
    } catch(ConstraintViolationException e) {
        return ResponseEntity.status(500).body("Username " + account.getUsername() + " already exists");;
    }
```

Which has been done in the correct-branch

5.Sensitive Data Exposure

There aren't much more to this than that passwords are stored in the database as clear text. Because of this every time a user registers or logs in the server will be handling clear text passwords. While it's especially dangerous over the web it shouldn't happen even internally in the server. Even if they would be hard to get access into storing sensitive data like this might lead in to trouble in the future when more advanced hacking methods are created .

This can be fixed the same way than in flaw #1 which was to implement encryption. Wisest would be to use some proven encryption library that always encrypts every password before handling them. This way even if an attacker got access to our database users passwords would be safe.

Fix to this was already shown in the first flaw. Adding encryption before saving