Cyber security project I

Project repository:
https://github.com/w4ldo/cybersecuritybase-project

This cyber security project work is a Spring-boot application and uses the provided skeleton build from the course project repository. It can be cloned and run on Netbeans for example.

The app has two users: user1 (password: user1) and user2 (password: user2)

Identifying the flaws

1. Security misconfiguration

steps to reproduce:
1. Log in (user1;user1 or user2;user2)
2. View site information (just left of URL depending on browser)
3. View cookies
4. From cookies in use open the localhost folder and see JSESSIONID
5. Notice how content is only 2 bytes instead of regular 16

The first flaw is a blatant misconfiguration in the CyberSecurityBaseProjectApplication.java.
The customize method calls-

cntxt.getManager().getSessionIdGenerator().setSessionIdLength(2);

-to set sessionIdLenght to 2 which can be easily guessed by the attacker and poses a potential security flaw. Fix would be to get rid of this configuration as the Spring-boot default settings are set to 16 bytes, which would be much harder to crack.

2. Broken access control

steps to reproduce:
1. Log in as user1 (password: user1)
2. Add title, content and press submit
3. A new note is created and appears on your list of notes
4. Press the “back” button to return to login screen
5. Log in as user2 (password: user2)
6. Type http://localhost:8080/notes/1 on your address bar
7. You can now see a note created by another user

to fix this the NoteController should use authentication to check the current user before returning the view

```
if (note.getAccount().equals(accountRepository.findByUsername(authentication.getName()))) {
        return “note”;
}
```

3. Sensitive Data Exposure

Right now the registered users passwords are stored in clear text. Every time a user logs in the servers are handling clear text passwords. If there would be breach on the servers all the users passwords would be compromised. Instead of saving passwords in clear text they should be encrypted when the account is created. To fix this would include a passwordEncoder in the CustomUserDetailService:

```
@Autowired
   private PasswordEncoder passwordEncoder;

   @PostConstruct
   public void init() {
      Account account = new Account();
      account.setUsername("user1");
      account.setPassword(passwordEncoder.encode("user1"));
      accountRepository.save(account);

      account = new Account();
      account.setUsername("user2");
      account.setPassword(passwordEncoder.encode("user2"));
      accountRepository.save(account);
   }
```

and SecurityConfiguration:

```
@Autowired
   private UserDetailsService userDetailsService;

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

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

In addition to this the server should be applied a SSL certificate to get HTTPS as the application layer protocol so that passwords are safe from anyone listening to the server.

4. Cross-Site Scripting (XSS)

steps to reproduce:
1. Log in (user1;user1 or user2;user2)
2. Type something in the title field
3. Type <script>alert(document.cookie);</script> in the content field
4. Press submit
5. Click on the title in the list of notes
6. You will be directed to localhost:8080/notes/{id}
7. Alert will pop up displaying cookies used by this site. (Alert will sometimes appear blank)

Due to broken access control another user could stumble on malicious script added by another user. To fix this instead of displaying notes as @ResponseBody they could be passed to the model and returned in a template where thymeleaf can parse the note in desired format:

in NoteController:

```
@RequestMapping(value = "/notes/{id}", method = RequestMethod.GET)
    public String loadNote(Model model, @PathVariable Long id) {
        model.addAttribute("note", noteRepository.getOne(id));
        return "note";
}
```

And in the note.html:

```
<p th:text="${note.title}"><p>
<p th:text="${note.content}"><p>
```

No input from user should ever be displayed unsanitized.

5. Broken authentication

Steps to reproduce:
1. Log in (user1;user1 or user2;user2)
2. Add title, content and press submit
3. A new note appears on the list of notes
4. Press “back” or go to localhost:8080/login to return to the login screen
5. To ensure you are logged out use fake credentials to try to log in
6. Use for example Postman (https://www.getpostman.com) to send a HTTP POST request to localhost:8080/delete
7. Log in again with the same user you used to create the note
8. Notice how the all the notes are gone

In conclusion an unregistered user can bypass authentication and send request to the server resulting in the database being cleared of all entries. To combat this stricter rules must be enforced in the SecurityConfiguration. Specifically the-

```
.antMatchers("/delete").permitAll()
```

-should be removed from the http.authorizeRequests() in the configure method to ensure no unauthenticated users gain access.

Bonus. Insufficient session management and role validation

Steps to reproduce:
1. Log in (user1;user1 or user2;user2)
2. Add title, content and press submit
3. A new note appears on the list of notes
4. Open the developer console in the browser (f12 in chrome)
5. Inspect the “DELETE” button in the elements tab
6. Right-click the input and select “edit as HTML”
7. Highlight the “disabled=”true”” and erase it with backspace
8. Press ctrl+enter
9. Click on the now enabled DELETE button
10. All notes are gone

Simply hiding or disabling unwanted functions in HTML is no way to control access rights. Component access should be validated on the server rather than on the client as front-side validations can be easily bypassed. Users should be established roles to control their access and session should track current users privileges.