Flaw report

Link: https://github.com/Cell9/cybersecurity

Installation:

- Clone the repo:
    - git clone
- In the cloned folder create jar-file:
    - mvn package
- Run the app:
    - java -jar target/cybersecuritybase-project-1.0-SNAPSHOT.jar
- App is can now be viewed at http://localhost:8080

Flaws:

Flaw 1: Sensitive data exposure

Description:
This application is currently storing all the passwords as plain text into the database, which would be extremely detrimental if the contents of the database were to be leaked or stolen somehow.

How to fix it:
Properly encrypting the passwords with a password encoder would be very helpful. To do this a password encoder should be added to the SecurityConfiguration.java. Start by defining the encoder by adding the following code:

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

In order to utilize the encoder correctly you must encrypt the passwords before storing them to the database. To do this, change the submit method under AccountController.java according to following code:

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String submit(@RequestParam String username, @RequestParam String password)
{
   Account a = new Account();
   a.setUsername(username);
   a.setPassword(passwordEncoder.encode(password));
   accountRepository.save(a);
   return "redirect:/login";
}
```

As the authentication must also be able to use the encoder, change the configureGlobal method under SecurityConfiguration.java according to following code:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

   auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
}
```

Flaw 2: Cross-site scripting (XSS)

Description:
If a user enters a scripts or any HTML code to the signup form. the code will be executed when the "Thank you"-page is rendered. Script e.g. *<script language="javascript" type="text/javascript">alert("You are vulnerable");</script>*.

How to fix it:
The app is currently rendering the done.html values using the *th:utext* which allows code to be executed. This can be prevented by changing them to *th:text,* which prevents execution of any unwanted code. Code should look like this after changes:

```
<ul>
     <li th:each="signup : ${signups}">
         <span th:utext="${signup.name}"></span>
         <span th:utext="${signup.address}"></span>
      </li>
</ul>
```

Flaw 3: Security misconfiguration

Description:
The application currently has the automatic CSRF checks manually disabled, which allows CSRF attacks. It means that, a signup could be forced by an element on another page if it has a suitable source URL.

How to fix it:
To fix this flaw, the command for manual disabling must be deleted from the configure method under SecurityConfiguration.java. To do this, remove the following code:

```
http.csrf().disable();
http.headers().frameOptions().sameOrigin();
```

Flaw 4: SQL injection

Description:
The app uses the input from the name field to show list of signups to the user. It does not sanitize the input and the query does not use any sort of secure methods. Instead it directly uses the input in the SQL query to fetch the list of signups. By using a suitable injection (e.g. `Rick' OR '1' = '1`), the entire contents of the database will be displayed.

How to fix it:
The way the app is currently fetching the signups via the name property is not very secure. If the signup is wanted to be shown on the done.html, it would be better to pass down the data from the parameters, e.g:

```
@RequestMapping(value = "/form", method = RequestMethod.POST)
public String submit(@RequestParam String name, @RequestParam String address, Model
model) {

    Signup s = new Signup(name, address);
    signupRepository.save(s);
    model.addAttribute("signup", s);
    return "done";

}
```

If it is necessary to fetch the information using the name parameter, the query should be made more secure. The way the app currently implements the custom

class SignupRepositoryFind.java to make a query with its own EntityManager makes it vulnerable to sql injections. This could be fixed by utilizing the Spring's native implementation for custom queries by using the more secure @Query notation:

```
@Query("SELECT s FROM Signup s where s.name = ?1")
String findByName(String name);
```

Flaw 5: Broken authentication

Description:
The app currently requires the user to authenticate in to access the signup form. There are however quite a few problems with it. First, the app is currently storing user passwords as plain text in the database as seen in flaw 1 above. Second, the app has no enforced password requirements such as minimum length or that it must contain numbers, capital letters or special characters. This makes it possible to use incredibly weak passwords such as 'password' or '1', which are useless. this compromises the security of the entire application and its users if the passwords are too weak and common.

How to fix it:
To fix the problem with minimum requirements, the passwords should be validated in submit under AccountController.java before creating the account. One way to do it would be:

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String submit(@RequestParam String username, @RequestParam String password,
Model model) {

    String requirement = "((?=.*[a-z])(?=.*\\\\d)(?=.*[A-Z])(?=.*[@#$%!]).{8,30})";
    Boolean passOk = password.matches(requirement);

    if (passOk) {
        accountRepository.save(new Account(username, password));
        return "redirect:/login";
    } else {
        model.addAttribute("error", "Password is too weak!");
        return "register";
    }
}
```

This checks for these criteria:

- 8-30 characters
- At least 1 lower and upper case character
- At least 1 special character(@#$%!) and digit

To show the error to user, add the following to register.html:

```
<span th:text="${error != null} ? ${error} : ""></span>
```