

Yifu Yang: 17204587, Mingqi Yang: 17208990, Tom Furlong: 18366263

Assignment 3

GitHub repository: https://github.com/YifuYANG/Runtime_Terror

Number of exposed vulnerabilities: 12

Number of Fixed vulnerabilities: 12

Point 1: Implement appropriate access control to only allow the user associated with a specific account to access his/her vaccination information

We already have that functionality done in the first assignment, a regular user can only see their own appointment while an admin can check and approve all the appointments as images shown below.

The first screenshot shows the 'Activity Management System' for user 'yifu'. It displays a table with vaccination appointment details:

First dose center	First dose date	First dose time	First dose status	Second dose center	Second dose date	Second dose time	Second dose status
UCD	2022-05-26	MORNING	PENDING	NA	NA	NA	NA

Below the table is an 'Exit and close' button.

The second screenshot shows the 'Activity Management System' for user 'mingqi'. It displays a similar table:

First dose center	First dose date	First dose time	First dose status	Second dose center	Second dose date	Second dose time	Second dose status
UCD	2022-05-15	MORNING	PENDING	NA	NA	NA	NA

Below the table is an 'Exit and close' button.

The third screenshot shows the 'Appointments Management System' for an admin user. It displays a message: '2 appointment(s) are waiting for confirmation:'. Below this is a table with appointment details:

Appointment ID	User ID	Dose N.O.	Brand	Vac Center	Status	Date	Action
5	46	1	PFIZER	UCD	PENDING	2022-05-15	Approve
6	45	1	PFIZER	UCD	PENDING	2022-05-26	Approve

Below the table is an 'Exit and close' button.

We use token to check user's identity:

1. Once the users visit the activity page, it sends their unique token to the controller
2. The token pool will check his/her identity
3. Then we use a helper function which find the user in database to check his/her privilege
4. If the current user is an ADMIN, then show all appointments amount all the users
5. If the current user is a CLIENT, then only show the appointments are relative to his/her account

```
@RestrictUserAccess(requiredLevel = UserLevel.ANY)
@GetMapping
public String activityPage(@RequestHeader("token") String token, Model model) throws CustomErrorException {
    try {
        Long userid = tokenPool.getUserIdByToken(token);
        String userName = userRepository.findById(userid).getFirst_name();
        List<ActivityForm> activities;
        if(!userIsAdmin(userid))
            activities = activityDao.findAllActivitiesByUserId(userid);
        else
            activities = activityDao.findAllActivities();
        model.addAttribute("user", userName);
        model.addAttribute("activities", activities);
        log.info("Activity page accessed, operator ID = " + tokenPool.getUserIdByToken(token));
        return "activity";
    } catch (Exception e){
        throw new CustomErrorException("some error happened");
    }
}

private boolean userIsAdmin(Long userId) throws CustomErrorException {
    try {
        return userRepository.findById(userId).get().getUserLevel() == UserLevel.ADMIN;
    } catch (Exception e){
        throw new CustomErrorException("some error happened");
    }
}
```

Our program uses the mechanism of interacting tokens to identify a user, the advantage of using a unique token to check a user's identity is that it will protect our web application from CSRF attack, as each time the controllers are expecting the incoming request holds a unique token as it's header (similar logic to synchronizer token pattern).

Point 2: Gain Admin Access with no checks - Critical CWE-657

Type: Insecure Design

The user is prompted if they want to register as an Admin or Standard User via a check box during the Account Creation step. The account is elevated to Admin privileges if the Admin checkbox is selected. Because any user may create an account with Admin level access and hence access all Admin features, this is a serious security and design flaw.

Steps for Security Control Adoption and Implementation:

1. Removed frontend option to allow users to register as admins

```
</div>
<!-- <div class="form-group"
    th:classappend="${#fields.hasErrors('userLevel')}? 'has-error':''">
    <label for="ADMIN" class="control-label">Administrator</label>
    <input type="radio" id="admin" class="form-radio" th:field="**{userLevel}"
        value="ADMIN" />
    <label for="CLIENT" class="control-label">Client</label>
    <input type="radio" id="client" class="form-radio" th:field="**{userLevel}"
        value="CLIENT" />
    <p class="error-message" th:each="error : ${#fields.errors('userLevel')}
        th:text="${error}">Validation error</p>
</div> -->
```

2. Set user level as CLIENT at the backend so even if attackers bypass the frontend validation and send a form of data directly to the backend, they still wouldn't be able to register as an administrator

```
59     // Create a new User
60     @PostMapping
61     public String RegisterUser(@ModelAttribute("newUser") User newUser, BindingResult result) {
62         newUser.setUserLevel(UserLevel.CLIENT);
63     }
```

We believe this adoption for security control is effective as we disabled the ability for a user to create an admin account in our web application through both the frontend or backend.

Point 3: No Strong Password Enforcement - High CWE-521

Type: Insecure Design

The password field only allows for six characters when creating an account. The suggested password length is eight characters. These characters can be any character, thus the password doesn't have to be complicated or include strong password qualities like symbols, capital letters, or digits.

Steps for Security Control Adoption and Implementation:

1. Force user to create an account with at least 8 digits password and check password strength at frontend

```
<input id="password" minlength="6" class="form-control" type="password"
<input id="password" minlength="8" class="form-control" type="password"
```

```
function checkPassword(){
    regex = /^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&-+=()])?(=\S+$).{8, 20}$/;
    if (info.password.match(regex)) {
        return true;
    } else {
        return false;
    }
}
```

2. Create a common string pool that loads a list of common string once the application starts, then create a helper function that loops through the list to check if the password contains any common string from the list

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class CommonStringPool implements CommandLineRunner {
    private List<String> list = new ArrayList<String>();

    public boolean ifContainCommonString(String password){
        for(String string : list){
            if(password.contains(string)){
                return true;
            }
        }
        return false;
    }

    @Override
    public void run(String... args) throws Exception {
        File file = new File("src/main/resources/10k-most-common.txt");
        Scanner input = new Scanner(file);

        while (input.hasNextLine()) {
            list.add(input.nextLine());
        }
    }
}
```

3. Create a password validator that will first check the strength of the password by matching it with the regex (8-20 length, an uppercase character, a number, and a special character must appear at least once), then combine and return a Boolean result of if the password length greater than 8, if the password is strong, whether it contains the user's real name, whether it contains empty space, and whether it contains a common string

```
private Boolean passwordValidator(String password, String lastname, String firstname){
    String regex = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&-+=(){}])(?=.*\\S+$).{8,20}$";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(password);
    return matcher.matches() && !password.contains(lastname) && !password.contains(firstname) && password.length()>8
        && !commonStringPool.ifContainCommonString(password);
}

if(!passwordValidator(newUser.getPassword(),newUser.getLast_name(),newUser.getFirst_name())){
    return "redirect:/register?passwordError";
}
```

We believe this adoption for security control is effective as we forced users to create an account with non-common string, non-empty space, does not contain real name, long, and strong password (non-common, non-empty space, does not contain user real name, 8-20 length, an uppercase character, a number and a special character must appear at least once) in our web application according to [Microsoft password policy recommendations](#).

Point 4 & 5: Brute Force Passwords - High CWE-307

Type: Insecure Design

During the user login step, our web application is vulnerable to Brute Forcing the password. This is feasible because the number of requests that can be sent when attempting to log in is not limited. A Cluster Bomb Attack, for example, can be used to accomplish this. This process can take a long time, but once done, it can compromise the security of the online application by allowing access to an admin or user account.

Steps for Security Control Adoption and Implementation:

1. Create a new model which generates a table to store the Ip address for unsuccessful login attempts.

```
@Data
@NoArgsConstructor
@Entity
@Table(name = "ip_logs")
public class Ip_logs {
    @Id
    @GeneratedValue
    private Long ipId;
    private String method;
    private String url;
    private String ip;
    private boolean accountLocked = false;
    private int failedAttempts = 1;
    private Date lock_time;
    private Date last_attempt;

    public Ip_logs(String method, String url, String ip) {
        this.method = method;
        this.url = url;
        this.ip = ip;
    }

    public Ip_logs(String method, String url, String ip, boolean accountLocked, int failedAttempts, Date lock_time, Date last_attempt) {
        this.method = method;
        this.url = url;
        this.ip = ip;
        this.accountLocked = accountLocked;
        this.failedAttempts = failedAttempts;
        this.lock_time = lock_time;
        this.last_attempt = last_attempt;
    }
}
```

2. Create AttemptLimitationDao class which increase failed attempts for a Ip address, reset failed attempts for a Ip address, lock a Ip address, and unlock a Ip address after 20 mins

```
public void increaseFailedAttempts(Ip_logs ip_logs){
    if(ipAddressRepository.findByIpAddress(ip_logs.getIp())!=null){
        int failAttempts = ip_logs.getFailedAttempts() + 1;
        ip_logs.setFailedAttempts(failAttempts);
    }
}

public void resetFailedAttempts(Ip_logs ip_logs) {
    ip_logs.setFailedAttempts(0);
    ipAddressRepository.save(ip_logs);
}
```

```
public void lock(Ip_logs ip_logs) {
    ip_logs.setAccountLocked(true);
    ip_logs.setLock_time(new Date());
    ipAddressRepository.save(ip_logs);
}

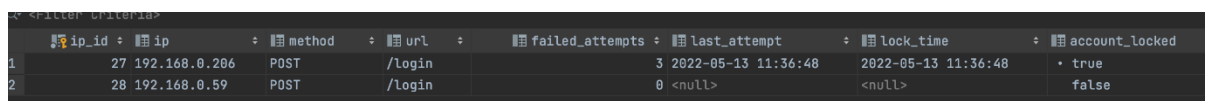
public void resetAttemptsAfterPeriodOfTime(Ip_logs ip_logs){
    if (ip_logs.getLast_attempt() != null) {
        long lastAttemptTimeInMillis = ip_logs.getLast_attempt().getTime();
        long currentTimeInMillis = System.currentTimeMillis();
        if (lastAttemptTimeInMillis + LOCK_DURATION < currentTimeInMillis) {
            ip_logs.setFailedAttempts(0);
            ipAddressRepository.save(ip_logs);
        }
    }
}
```

3. Inside the login controller, each time a user performs unsuccessful login, we will log his Ip address and increase its failed attempts accordingly, when an Ip address with failed attempts exceeded 3 we will record the lock_time and set the account_locked to true, this will lock the target Ip for 20 mins

```
} else if(ip_log!=null && !attemptLimitationDao.unlockWhenTimeExpired(ip_log)){
    map.put("status", "fail");
    map.put("msg", "Too many request, your ip has been banned for 20 mines");
} else {
    if(!encoder.matches(loginForm.getPassword(),user.getPassword())){
        if(ip_log==null) {
            ipAddressRepository.save(new Ip_logs(request.getMethod(),request.getRequestURI(),request.getRemoteAddr()));
        } else {
            if(!ip_log.isAccountLocked()){
                attemptLimitationDao.resetAttemptsAfterPeriodOfTime(ip_log);
                attemptLimitationDao.setLastAttempt(ip_log);
            }
            if(ip_log.getFailedAttempts()>2){
                attemptLimitationDao.lock(ip_log);
            } else {
                attemptLimitationDao.increaseFailedAttempts(ip_log);
            }
        }
        map.put("status", "fail");
        map.put("msg", "Wrong password.");
    }
}
```

We believe this adoption for security control is effective because instead of logging victim accounts that are being brute-forced, we log the attacker's Ip address so that when a victim's account is frozen, the attacker cannot change the target to brute force other accounts. When an attacker attacks multiple accounts at the same time, this will greatly extend the cracking time.

Below image gives an example of lock account:



	ip_id	ip	method	url	failed_attempts	last_attempt	lock_time	account_locked
1	27	192.168.0.206	POST	/login	3	2022-05-13 11:36:48	2022-05-13 11:36:48	true
2	28	192.168.0.59	POST	/login	0	<null>	<null>	false

Point 6: Outdated libraries - Medium CWE-1104

Type: Vulnerable and Outdated Components

Our application contains outdated libraries. The following is a list of them. These libraries have known vulnerabilities that could be exploited if the features with vulnerabilities are used in the future.

- Bootstrap v3.3.7 - <https://snyk.io/test/npm/bootstrap/3.3.7>
- JQuery v3.2.1 - <https://snyk.io/test/npm/jquery/3.2.1>

Steps for Security Control Adoption and Implementation:

1. Use maven command “mvn versions: display-dependency-updates” to check updates

```
org.webjars:bootstrap ..... 3.3.7 -> 5.1.3
org.webjars:jquery ..... 3.2.1 -> 3.6.0
```

2. Modify pom.xml to update the components

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>5.1.3</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.6.0</version>
</dependency>
```

3. Do additional front-end fine-tuning, component updates will affect the front-end UI

We believe this adoption for security control is effective as we replace the obsolete components to the latest version and we did additional front-end fine-tuning.

Point 7: Lack of logging - Low CWE-778

Type: Security Logging and Monitoring Failures

Our web application's logging features are incompatible. There is logging for any type of update or writing to the database in numerous controllers. There is no information logging for other controllers. This mismatch exposes the web app to potential exploitation without any warnings or logs being generated to identify the problem.

Steps for Security Control Adoption and Implementation:

1. Import slf4j package, then use log.info() to log all important actions
2. Log the user id when the token pool issues a token to the user
`log.info("Token issued to " + user.getUserId());`
3. Log the user id when removing his/her token from the token pool
`log.info("Token removed for user id = " + tokenPool.getUserIdByToken(token));`
4. Log the user id who accesses activity page
`log.info("Activity page accessed, operator ID = " + tokenPool.getUserIdByToken(token));`
5. Log the user id who accesses the admin page
`log.info("Admin page accessed, operator ID = " + tokenPool.getUserIdByToken(token));`
6. Log the update of an appointment and how updates it
`log.info("Update: appointment id = " + id + ", operator ID = " + tokenPool.getUserIdByToken(token));`
7. Log which user create what post
`log.info("User ID = " + userId + " created new post: " + content);`

We believe this adoption for security control is effective as we logged all important actions or the actions that potential attackers can use to attack our web application. Logging what each user did would give us some clues about who did what in where that harmed our application if our application was attacked.

Point 8: Improper Error Handling – Medium

Type: Insecure Design

Our web application did not handle the error messages properly, the errors were shown on white label error pages, which may contain information about the DB or the technology used to implement our web application.



Steps for Security Control Adoption and Implementation:

1. We create below Custom Error Exception classes which will create new custom error pages when some unpredictable error occurs

```
public class CustomErrorException extends Exception {
    public CustomErrorException(String msg){ super(msg); }
}

@ControllerAdvice
public class CustomErrorExceptionAdvice {

    @ExceptionHandler(value = CustomErrorException.class)
    public ModelAndView exceptionHandler(CustomErrorException e){
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg", e.getMessage());
        mv.setViewName("error");
        return mv;
    }
}
```

2. For access control, we throw error messages according to different situations, for example, when user does have the privileges to access the target API we say “access deny”

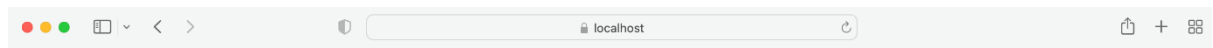
```
if(token == null || token.length() == 0) {
    log.warn("Absent token detected");
    throw new AuthenticationException("Access denied, please login first.");
}

if(!tokenPool.containsToken(token)) {
    log.warn("Invalid token detected -> " + token);
    throw new AuthenticationException("Access denied, invalid token >> " + token);
}

//If token is valid, we need to check whether user level satisfies required level
Long userId = tokenPool.getUserIdByToken(token);
//Then check if the token is expired
if(!tokenPool.validateTokenExpiry(token, LocalDateTime.now())) {
    log.warn("Expired token detected -> " + token);
    throw new AuthenticationException("Access denied, your token has been expired, please re-login.");
}

if(requiredLevel == UserLevel.ANY) {
    log.info("Token approved to execute " + method.getName());
    return joinPoint.proceed();
}

else if(userRepository.findById(userId).get().getUserLevel() != requiredLevel) {
    log.warn("Insufficient authorisation detected -> " + token);
    throw new AuthenticationException("Access denied, you have no privileges to access this content.");
}
```



Access denied, you have no privileges to access this content.

- For any other functions that will react with the database we use try and catch statement, if any unpredictable error relative to the DB occurs, the catch statement will just create a new page and throw a message instead of show users the actual error message

```
} catch (Exception e){  
    throw new CustomErrorException("some error happened");  
}
```



some error happened

- For registration we also used custom error alerts to show the user what data they entered in wrong.
- Firstly in the html we used the keyword 'required' in the forms inputs to make sure the user fills in all details, we also used the 'patterns' keyword to enforce the user to use correct patterns for data like PPS number, email, password. This forces the user to use correct data, we also check this on the server side.

```
<input id="PPS_number" minlength="8" maxlength="8" class="form-control"  
  th:field="*{PPS_number}" placeholder="1234567A" required pattern="\d{7}[a-zA-Z]{1}"  
  title="Follow the format of 7 digits and 1 letters at the end."/>
```

PPS Number

1231231|

Phone Number

0831395686

E-mail



Please match the requested format.
Follow the format of 7 digits and 1 letters at the end.

- On the server side we validate these inputs as well and return custom error alerts if needed. We do this for data validation and also checking for unique values which can only be done on the server side like if an email and a PPS number are unique (detail will be mentioned in point 10).

An account for this email exists already. Please try again!

Registration

First name

Last name

This PPS number is already registered.

Registration

Enter email in correct format. Please try again!

Registration

We believe this adoption for security control is effective as we implemented our own custom error pages and apply them to all functions that interact with the database, by doing so it will hide any information relative to our database or the technology that we used to implement the web application.

Point 9: No Anti CSRF tokens - Medium CWE-352

This point was ignored according to Liliana's clarification

Liliana Pasquale

May 9, 2022, 5:25 PM (5 days ago)



to me ▾

Hi Yifu,

sorry I am just getting back to you regarding yesterday's email.

Please, note that not everything that is mentioned in the report is 100% correct. The report should only serve as a general guideline.

What I am asking is to address the list of comments provided in the email.

Regarding your point 1: I understand that you use the users' tokens to regulate access control. However, you want to make sure that only the user associated with a specific account can access his/her vaccination information

Regarding your point 2: Please, disregard my request to protect against Cross-Site Request Forgery using a synchronizer token.

Regarding point 3, my request is different: I have asked to keep track of consecutive failed authentication attempts (independently of the time during which they occur). If an IP address performs 3 consecutive failed authentication attempts, block it for a given amount of time (e.g., 20 mins).

I hope this clarifies your questions.

Point 10: Improper Input Validation during registration - Medium CWE-20

Type: Insecure Design

Our web application lacks proper input validation during the registration step, for example, users may enter the wrong type of PPS number or repetitive PPS number to register an account.

Steps for Security Control Adoption and Implementation:

1. We create special characters filter to check user name and country name by matching them with the regex (if contains /, \, <, or >), if any inputs contain suspicious symbols, the backend will return an error message and determine the request

```
private Boolean specialCharacterFilter(String username){
    String regex = "[^\\/<>]*$";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(username);
    return matcher.matches();
}

if(!specialCharacterFilter(newUser.getFirst_name()) || !specialCharacterFilter(newUser.getLast_name())){
    return "redirect:/register?usernameError";
}

if(!specialCharacterFilter(newUser.getNationality())){
    return "redirect:/register?nationalityError";
}
```

2. We create PPS number validators on both side of our application to force users to input a valid PPS number by matching it with the regex (if PPS start with 7 digits and end with one letter, for example 1234567A) or a non-existing PPS number by checking it in database (repetitive PPS number)

```
<div class="form-group"
    th:classappend="{#fields.hasErrors('PPS_number')}? 'has-error':''">
    <label for="PPS_number" class="control-label">PPS Number</label>
    <input id="PPS_number" minlength="8" maxlength="8" class="form-control"
        th:field="{PPS_number}" placeholder="1234567A" required pattern="\d{7}[a-zA-Z]{1}"
        title="Follow the format of 7 digits and 1 letters at the end."/>
    <p class="error-message" th:each="error : ${#fields.errors('PPS_number')}"
        th:text="{error}">Validation error</p>
</div>

private Boolean ppsValidator(String ppsNumber){
    /** Seven Digits then one letters for PPSN*/
    String regex = "\\d{7}[a-zA-Z]{1}";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(ppsNumber);
    return matcher.matches();
}

private Boolean findByPPSN(String ppsNumber) {
    List<String> ppsn_list = findAllPPSN();
    Collections.reverse(ppsn_list);
    for (String ppsn:ppsn_list){
        if (ppsnEncoder.decrypt(ppsn).equals(ppsNumber)){
            return true;
        }
    }
    return false;
}
```

```
        if(!ppsValidator(newUser.getPPS_number())){
            return "redirect:/register?ppsnError";
        }
        if(findByPPSN(newUser.getPPS_number())) {
            return "redirect:/register?ppsnExistError";
        }
    }
```

3. We create an email validator on both client and server sides of our application to check if users inputs a valid email address by matching it with the regex (proper email structures)

```
<div class="form-group"
    th:classappend="${#fields.hasErrors('email')}? 'has-error':''">
    <label for="email" class="control-label">E-mail</label>
    <input type="email" id="email" class="form-control" th:field="*{email}" required
    pattern="^[\\w!#$%&'*/=?`{|}~^-]+(?:\\.[\\w!#$%&'*/=?`{|}~^-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}"
    title="Email must be in the format john@doe.com" placeholder="john@doe.com"/>
    <p class="error-message" th:each="error : ${#fields.errors('email')}""
        th:text="${error}">Validation error</p>
</div>

private Boolean emailValidator(String email){
    String regex = "^[\\w!#$%&'*/=?`{|}~^-]+(?:\\.[\\w!#$%&'*/=?`{|}~^-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(email);
    return matcher.matches();
}
```

```
        if(!emailValidator(newUser.getEmail())){
            return "redirect:/register?invalidEmail";
        }
    }
```

4. We create a phone number validator on both client and server sides of our application to force users to input a valid phone number by matching it with the regex (if the phone number is 10 digits which start with 08)

```
<div class="form-group"
    th:classappend="${#fields.hasErrors('phone_number')}? 'has-error':''">
    <label for="phone_number" class="control-label">Phone Number</label>
    <input id="phone_number" class="form-control" th:field="*{phone_number}"
    minlength="10" maxlength="10" placeholder="0831234567" pattern="^(08)[0-9]{8}$"
    required title="Phone number must be 10 digits and begin with 08"/>
    <p class="error-message" th:each="error : ${#fields.errors('phone_number')}""
        th:text="${error}">Validation error</p>
</div>
```

```
private Boolean phoneValidator(long phoneNumber){
    String regex = "^(8)[0-9]{8}$";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(String.valueOf(phoneNumber));
    return matcher.matches();
}
```

```
        if(!phoneValidator(newUser.getPhone_number())) {
            return "redirect:/register?phoneNumberError";
        }
    }
```

5. We create a date of birth validator on both client and server sides of our application to force users to input a valid date of birth by matching it with the regex (yyyy-mm-dd)

```
<div class="form-group"
  th:classappend="{#fields.hasErrors('date_of_birth')}? 'has-error':''">
  <label for="date_of_birth" class="control-label">Date Of Birth</label>
  <input id="date_of_birth" type="date" min="1920-01-01" max="2004-03-01"
    class="form-control" th:field="*{date_of_birth}" required/>
  <p class="error-message" th:each="error : ${#fields.errors('date_of_birth')}}"
    th:text="{error}">Validation error</p>
</div>

private Boolean dobValidator(String dob){
  String regex = "^\\d{4}-(0[1-9]|1[0-2])-(0[1-9]|1[2][0-9]|3[01])$";
  Pattern pattern = Pattern.compile(regex);
  Matcher matcher = pattern.matcher(String.valueOf(dob));
  return matcher.matches();
}

if(!dobValidator(newUser.getDate_of_birth())){
  return "redirect:/register?dobError";
}
```

6. We create a password validator to force users to input a strong password as mentioned in point 3 above

We believe this adoption for security control is effective as we try to validate all user input during the registration step on both client and server sides of our application, and also apply the idea of whitelisting as much as we can to our application. Instead of telling users what they cannot do, we tell them what they can do, for example, only 10 digits phone numbers which start with 08 are allowed.

Point 11: Content Security Policy (CSP) Not Implemented - Medium

Our app does not use Content Security Policy. There is no direct impact on our app if CSP is not implemented. If our website is vulnerable to an XSS attack, CSP can prevent the vulnerability from being exploited successfully. We will be missing out on an extra layer of protection if we do not deploy CSP.

Steps for Security Control Adoption and Implementation:

1. Add Content security policy header to all pages, for example, on the index page we set style-src and script-src to self, by doing so we restrict the script and style loading only from the same origin (domain + port)

```
src/main/resources/templates/index.html

@@ -6,6 +6,7 @@

<title>Vaccination Assistant</title>
<script type="text/javascript" th:src="@{/js/index.js}"></script>
<link rel="stylesheet" type="text/css" th:href="@{/css/index.css}">
<meta http-equiv="Content-Security-Policy" content="style-src 'self' https://cdn.jsdelivr.net script-src 'self' https://cdn.jsdelivr.net">
<!-- <link rel="stylesheet" type="text/css" th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap.min.css}" /> -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhF1dvKuhf1
<script src="https://cdn.jsdelivr.net/npm/vega@5.20.2/build/vega.min.js"></script>
```

We believe this adoption for security control is effective as we applied CSP on all our web pages, which create an extra security layer to protect our application from cross-site Scripting attack.

Point 12: Missing Encryption of Sensitive Data - Medium CWE – 311

Type: Insecure Design

Our web application leaks encryption for sensitive data such as PPS number, date of birth, and phone number, if the database was breached the attacker can extract that sensitive data directly.

Steps for Security Control Adoption and Implementation:

1. Encrypt sensitive information before storing them in the database including PPS number, date of birth, phone number, and password.
2. For the PPS number, Date Of Birth and Phone number we used the Jasypt library which has simplified but strong encryption for texts and numbers. It is bidirectional so we can decrypt the data if needed. For the password we use BCrypt which is a strong password encoder that is easily integrated with spring applications.
3. Using these encryptions ensures sensitive data cannot be leaked if the database is breached.

```
newUser.setPassword(passwordEncoder.encode(newUser.getPassword()));  
/**  
    PPS, data of birth, and phone number should be encoded before storing in to DB  
    */  
String dob = String.valueOf(newUser.getDate_of_birth());  
newUser.setDate_of_birth(dobEncoder.encrypt(dob));  
newUser.setPPS_number(ppsnEncoder.encrypt((newUser.getPPS_number())));  
BigInteger bigPhoneNumber = BigInteger.valueOf(newUser.getPhone_number());  
long phoneNum = phoneNumberEncoder.encrypt(bigPhoneNumber).longValue();  
newUser.setPhone_number(phoneNum);  
/** To decrypt use the decrypt() method that jasypt provides */
```

We believe this adoption for security control is effective as we used jasypt and BCrypt encoders to encode all the sensitive information before storing them into the database, which have simplified but strong encryptions for strings and numbers.