



# Web Application Security Patching Report

Product Name: British Airways

Product Version: v2.0

Test Completion: 25/05/2021

## **Application Developers**

Ee En Goh ([17202691](#))

Seán Conor McLoughlin ([10360547](#))

Svetoslav Nizhnichenkov ([17712081](#))

Olanipekun Akintola ([17734755](#))

---

# Table of Contents

---

<b>About this report</b>	2
<b>1 Administrator Functionality</b>	3
<b>2 Patches Done</b>	4
Insufficient Session Expiration	4
Improper Restriction of Excessive Authentication Attempts	9
Use of Hard-coded Password	14
Weak Password Requirements	17
Cleartext Transmission of Sensitive Information	20
Insufficient Logging	27
Missing Custom Error Page	30
Missing Encryption of Sensitive Data	32
Cleartext Storage of Sensitive Information	34
Improper Input Validation	40
Use of GET Request Method with Sensitive Query Strings	42
Failure to Restrict URL Access	44
<b>3 Extras</b>	46
MySQL to h2 migration	46

---

# About this report

---

## Executive Summary

Engineering Team: Team inSecurity

Number of days patching:

Test Start Date: 10/05/2021

Test End Date: 25/05/2021

## Project Information

Application Name: British Airways

Application Version: v2.0

Application Repository : [GitHub](#) Repository

[Trello Board](#)

Release Date: 05/03/2021

## Patching Summary

### Total Vulnerabilities Fixed

Severity	#Vulnerabilities
Critical	5
High	4
Medium	3
Low	0

---

# Chapter 1: Administrator Functionality

---

The administrator functionality is enabled through the use of the 'roles' table and the associated 'user\_roles' table, which applies the relevant role('MEMBER', 'GUEST', 'ADMIN') upon saving the User to the db. Access control to the endpoints is defined in *WebSecurityConfig* and with **@PreAuthorize** annotation in the controller. For example, the *"/reservations/username"* endpoint is accessible to both the specific User and any administrator. In *WebSecurityConfig.java* it is defined:

```
.antMatchers( ...antPatterns: "/user", "/getUserReservations").access( attribute: "hasAnyAuthority('ADMIN','MEMBER')")
```

and in *ReservationController.java*, the specific username is used to authorise:

```
@PreAuthorize("#username == authentication.name or hasAuthority('ADMIN')")
@GetMapping("/{username}")
```

To differentiate between the administrator and member when carrying out modifications on the member's account as an administrator, two **User** objects are instantiated. *sessionUser* is defined as the logged in **User** and *user* is the member having actions carried out upon them. If the logged in **User** is a member, then these are both the same. If an administrator is logged in, then the administrator is represented by *sessionUser*, and the member by *user*, where the **User** object is retrieved by searching the *UserRepository* for the username as given by the endpoint.

To display the relevant details in a given page, Thymeleaf takes the user\_role from the passed *sessionUser* modelAttribute and displays the content relevant to the **User** role:

```
<div th:each="entry: ${sessionUser.getRoles()}">
    <div th:if="${entry.name} == 'MEMBER'" class="dropdown-menu"
        aria-labelledby="member-drop-link">
```

---

## Chapter 2: Patches Done

---

### Critical : Insufficient Session Expiration [CWE-613]

#### Description

This vulnerability can be found in the web application that permits an attacker to reuse old session credentials or session IDs for authorization. In this application, there was no session invalidation control implemented, which results in an unexpired session cookie. Thus the cookie monitoring and blacklisting security control is implemented to patch this flaw.

#### Vulnerability location :

Any URL endpoint that can insert session cookie **JSESSIONID** in [version 1.0](#) (JWT in new patch)

#### Tasks included :

- Invalidate the cookie when a user logs out
- Fix an adequate session expiration time (15-30 minutes)



Figure 2.1: Vulnerable session cookie used in version 1.0

### Login

Username

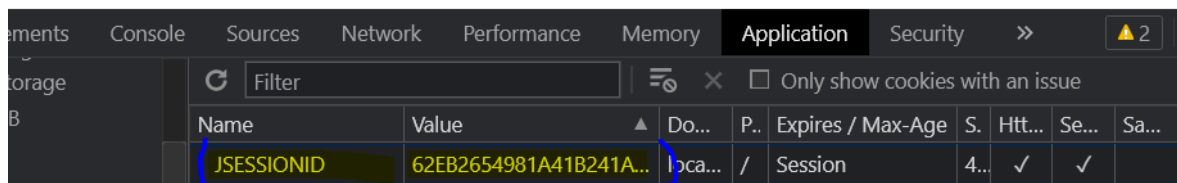


Figure 2.2: Session cookie is injectable in inspect element on any browser

### Security Control Implementation

1. In this patch, JSON Web Token (JWT) is used for both authentication and authorisation
2. When a successful login is done, a unique JWT token will be assigned to the user to get validated for authorisation on every single request. The related implementation can be found

in *JWTAuthenticationFilter* class **successfulAuthentication()** method (fig 2.3), that leads to the handling by *LoginSuccessfulHandler* class **onAuthenticationSuccess()** method.

```
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,
                                         FilterChain chain, Authentication auth) throws IOException {

    try {
        loginSuccessfulHandler.onAuthenticationSuccess(request, response, auth);
    } catch (ServletException e) {
        logger.error("Something wrong with AuthenticationSuccessHandler");
        e.printStackTrace();
    }
}
```

Figure 2.3: **successfulAuthentication()** method in *JWTAuthenticationFilter* class, that gets called when the given credential is correct

```
58         String token = JWT.create()
59             .withSubject(((ACUserDetails) auth.getPrincipal()).getUsername())
60             .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
61             .sign(HMAC512(SECRET.getBytes()));
62
63         response.addHeader(HEADER_STRING, s1: BEARER + token);
64
65         addCookie(token, response);
66
67         // Add to JWT token monitoring list
68         jwtTokenRepository.save(new JwtToken(userRepository.findByUsername(username), token));
```

Figure 2.4: **onAuthenticationSuccess()** method in *LoginSuccessfulHandler* class in charge of the creation of the JWT token and stores it to JWT\_TOKEN database for monitoring

3. Figure 2.5 shows how the new token is monitored with the database table **JWT\_TOKEN**, that the monitored token here is modelled with **JwtToken** class (fig 2.6) in the package *user.model* with associated JpaRepository named **JwtTokenRepository** in package *user.repository*
4. **JwtToken** class (fig 2.6) has 3 key fields, which are used in expire the used JWT token :
  - String *jwtToken* (line 17) that refers to the hash of the new JWT token
  - Date *expirationDate* (line 20) that refers to when this token will expire (see line 33 shows how it is calculated)
  - boolean *logout* (line 27) shows that if the user using this token has logged out
5. When a user logs out, the customized logout handler *CustomLogoutHandler* class will execute the overridden method **logout()** (fig 2.7) to find the **COOKIE\_NAME**, and set the boolean field **logout** to true. It means that this token has been used regardless of its expiration time, and thus it should not be used, ie. invalidated by adding to the blacklist.
6. In our security implementation, *JWTAuthorisationFilter* class method **doFilterInternal()** will be executed for every single request to role-dependent URLs done on the application. The class method **getAuthentication()** is used to authenticate the JWT token used in the request, with the following cases :
  - Codeblock in figure 2.8 catches the cases when expired or broken token value are used as the JWT token

- Codeblock in figure 2.9 catches the cases when the blacklisted JWT token is used, i.e. that the user that has used this token previously has logged out, or that this token is structurally valid but makes invalid claims
7. The token validation method used on line 114 in figure 2.8 is implemented in class **JwtTokenService**, with the following checks (see figure 2.10) :
- Line 93 checks if the associated username can be found in the userRepository, else it is an invalid username
  - Line 102 catches the NullPointerException that could be thrown from line 89 when the structurally correct JWT token used cannot be found in the jwtTokenRepository, thus it is an invalid token since it is not generated by this application.
  - Line 104 catches any unexpected error with the structurally correct JWT token
8. Any invalid JWT token detected by the security control mentioned above, will return **null** from method **getAuthentication()** mentioned in step 6. Thus, this invalid token will be removed from the response (figure 2.11), that will redirect the attacker to the login page (configured on *WebSecurityConfig*, see figure 2.12).

```

68 UsernamePasswordAuthenticationToken authentication = getAuthentication(request, token);
69
70 // Delete the invalid cookie from the response and the web browser
71 if(authentication == null) {
72     jwtCookie.setPath("/");
73     jwtCookie.setHttpOnly(true);
74     jwtCookie.setMaxAge(0);
75     response.addCookie(jwtCookie);
76

```

Figure 2.11: Delete invalid token from the response if the authentication token return null

```

122 .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementConfigurer<HttpSecurity>
123 .invalidSessionUrl(LOGIN_URL + "?expired");

```

Figure 2.12: Caption

## Why this security control is effective

This security control should be effective, since it stores any new JWT token created to the monitoring list when a user login successful, and invalidate the token right after the user logout successfully. Not only that, an attacker that uses any invalid or expired token will also be redirected back to the login page without any privilege escalation, with sufficient error handling implemented. Even if the user is forced to logout due to an expired token without handling by the LogoutHandler, it will still be okay since this implementation disallow the use of an expired token.

jdbc:h2:file:./data/fileDb

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM JWT\_TOKEN

SELECT \* FROM JWT\_TOKEN;

TOKEN_ID	EXPIRATION_DATE	TOKEN
1	2021-05-24 23:02:32.353	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMi

ATTEMPTS  
CONFIRMATION\_TOKEN  
CREDIT\_CARDS  
FLIGHTS  
GUESTS  
JWT\_TOKEN  
PASSENGERS

Figure 2.5: Database table JWT\_TOKEN that store the new JWT token for monitoring

```

16 @Column(name = "token")
17 private String jwtToken;
18
19 @Temporal(TemporalType.TIMESTAMP)
20 private Date expirationDate;
21
22 @OneToOne(targetEntity = User.class, fetch = FetchType.EAGER)
23 @JoinColumn(nullable = false, name = "user_id")
24 private User user;
25
26 @Column(name = "has_logout", nullable = false)
27 private boolean logout;
28
29 public JwtToken(){}
30
31 public JwtToken(User user, String token){
32     this.user = user;
33     this.expirationDate = new Date(new Date().getTime() + EXPIRATION_TIME);
34     this.jwtToken = token;
35     this.logout = false;
36 }

```

Figure 2.6: Definition of *JwtToken* class that model the monitored JWT token

```

30 if(cookies != null) {
31     for (Cookie cookie : cookies) {
32         if (cookie.getName().equals(COOKIE_NAME)) {
33             token = cookie.getValue();
34         }
35     }
36 }
37
38 // Invalidate the token used in this session before logout
39 // by adding it into a blacklist. Deletion of expired token
40 // can be implemented if time is efficient
41 JwtToken jwtToken = jwtTokenRepository.findByJwtToken(token);
42 jwtToken.setLogout(true);
43 jwtTokenRepository.save(jwtToken);

```

Figure 2.7: Set the boolean field **logout** to *true* when the user that is using the monitored JWT token has logged out



```

97     try {
98         username = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
99             .build()
100             .verify(token.replace(BEARER, replacement: "")) // Verify the token signature
101             .getSubject(); // Retrieve the username
102     } catch (TokenExpiredException error) {
103         log.warn("This request uses an expired JWT token, it will be deleted");
104         return null;
105     } catch (Exception e) {
106         log.warn("This request uses an invalid value for JWT token, it will be deleted");
107         return null;
108     }

```

Figure 2.8: `getAuthentication()` that catch the exceptions when invalid token (expired and broken token value) is used

```

114     if (jwtTokenService.isValidToken(token)) {
115
116         if (userDetailsService == null) {
117             userDetailsService = FilterUtil.loadUserDetailsService(request);
118         }
119         UserDetails userDetails = userDetailsService.loadUserByUsername(username);
120
121         return new UsernamePasswordAuthenticationToken(userDetails, userDetails.getPassword(), userDetails.getAuthorities());
122     } else {
123         log.warn(String.format("Invalid JWT token is used for authorisation : %s", token));
124     }
125
126     return null;
127 }

```

Figure 2.9: Check if the JWT token used is invalid, ie. correct username is used and the token is not blacklisted

```

88     try {
89         String username = getUsernameFromToken(token);
90
91         if (username == null) log.warn("Username not found in the given JWT token");
92
93         final boolean with_valid_username = userRepository.findByUsername(username) != null;
94         final boolean has_logout = isUserLogout(token);
95
96         if (!with_valid_username)
97             log.warn(String.format("Username <%s> found in the given JWT token is not registered", username));
98         if (has_logout) log.warn("This JWT token has been used by a logout user");
99
100         return with_valid_username && !has_logout;
101     } catch (NullPointerException error) {
102         log.warn("JWT token in valid structure with unregistered username is used, thus an invalid token");
103     } catch (Exception error) {
104         log.warn("JWT token in valid structure with suspicious/unexpected claims was used : " + token);
105         log.warn(error.getMessage());
106     }
107
108     return false;
109 }

```

Figure 2.10: Token validation method implementation in details

---

# Critical : Improper Restriction of Excessive Authentication Attempts [CWE-307]

## Description

This vulnerability occurs in software that does not implement sufficient measures to prevent multiple failed authentication attempts within in a short period, which makes it more susceptible to brute force attacks. In the first version of this application, the implementation allows brute-force and password guessing attacks to gain access to some of the existing user accounts. Hence, the login failure attempt counter is implemented as the security control, to limit of the consecutive failed attempts and 20 minutes lock time is assigned if it reaches or exceed the defined attempt limit.

### Vulnerability location :

Login page of this application, ie. /login URL

### Tasks included :

- Limit the number of consecutive failed authentication attempts to 3
- If an IP address performs 3 consecutive failed authentication attempts block it for a given amount of time (e.g., 20 mins)

## Security Control Implementation

1. This security control implementation is able to detect and block both *account-based* (correct username but incorrect password) and *IP-based* (both credentials are unregistered) brute-force attacks
2. After user logs in, the authentication has to go through the **JWTAuthenticationFilter** that validate if the login attempt is success with 3 overridden methods :
  - **attemptAuthentication()** that validate the credentials used in the login attempt
  - **successfulAuthentication()** that is called when the login attempt success, then it will pass the relevant information to the **LoginSuccessfulHandler** for further handling before allow the user to login successfully.
  - **unsuccessfulAuthentication()** that is called when the login attempt failed, then it will pass the relevant information to the **LoginFailureHandler** for further handling before deny the user login attempt.
3. Figure 2.13 shows how **attemptAuthentication()** handle certain exceptions during the authentication process. Line 67 is used to differentiate the type of brute-force attack prevention to be used. If `userDetails` on line 65 is *null*, it shows that the given username is not registered, and thus *UsernameNotFoundException* is passed to the **unsuccessfulAuthentication()** method, else it will check if the password matches the username in the database (line 70 to 73), which will lead to handling of possible account-based brute-force attack. Line 75 catches the *IpAddressLockedException*, which will happen when the failed login attempts were detected on certain IP address, and thus that IP address will be denied from login attempts for 20 minutes.

4. If the login attempt is a failure, **onAuthenticationFailure()** method (figure 2.14) from the class **LoginFailureHandler** will be called to handle the given exception. Line 62 decides which brute-force attack protection to be executed, *IP-based* (line 63) if the user get null from the search by username on the database, else it will be *account-based* attack prevention (line 67). The resulting exception will be added in the response, to differentiate the reason of authentication failure to the front-end (see the **login()** method of class **AuthenticationController** in package *user.controller*)

```
56  @Override
57  public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws
58
59      if (userDetailsService == null) {
60          userDetailsService = FilterUtil.loadUserDetailsService(request);
61      }
62
63      String username = request.getParameter("username");
64      try {
65          UserDetails userDetails = userDetailsService.loadUserByUsername(username);
66
67          if (userDetails == null)
68              unsuccessfulAuthentication(request, response, new UsernameNotFoundException("Username <" + username));
69          else
70              UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =
71                  new UsernamePasswordAuthenticationToken(userDetails, request.getParameter("password"),
72
73                  return authenticationManager.authenticate(usernamePasswordAuthenticationToken);
74
75      } catch (IpAddressLockedException error) {
76          unsuccessfulAuthentication(request, response, error);
77      }
78      return null;
79  }
```

Figure 2.13: Details of **attemptAuthentication()** how it handles the authentication validation

```
53  @Override
54  public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response,
55
56      String username = request.getParameter(USERNAME_PARAM);
57
58      // Load the User object to check if this user exist
59      User user = userRepository.findByUsername(username);
60
61      // Failed login attempt with unknown username from certain IP address
62      if (user == null)
63          failed = ipBasedFailureHandler(username, request.getParameter("password"), failed);
64
65      // Failed login attempt with certain registered username
66      else
67          failed = accountBasedFailureHandler(user, failed);
68
69      request.getSession().setAttribute(SECURITY_LAST_EXCEPTION, failed);
70
71      new DefaultRedirectStrategy().sendRedirect(request, response, FAILED_LOGIN_URL);
72  }
```

Figure 2.14: Overview process of **onAuthenticationFailure()** method from class **LoginFailureHandler**, of how it handles those authentication exceptions

5. The **ipBasedFailureHandler()** method handle exceptions when the IP address is currently locked (*IpAddressLockedException*), or unregistered username was used (*UsernameNotFoundException*). When latter case occurs, a new **AuthenticationFailureBadCredentialsEvent** will be created and listened by the **AuthenticationFailureListener** class to extract the IP address from the request and cached by the **LoginAttemptService** class.

- 
6. Figure 2.15 shows how the caching of IP address that fails the login attempt with [Google Guava](#). Line 27 explains that any cached IP address will stay in this *attemptCache* for 20 minutes with method `expireAfterWrite()`. **AuthenticationFailureListener** will extract the IP address, insert/find it from this cache and lock it when the consecutive failed attempts exceed 3 times (`MAX_ATTEMPT`). Cached IP addresses get invalidated when **AuthenticationSuccessListener** listens any successful login event from that IP address.
  7. The **accountBasedFailureHandler()** method in figure 2.14 uses the exception received to differentiate if the account is locked when an account-based attack is detected. If the attempt is still within the limits, the associated attempt object will be updated based on the given username, and when the update makes the attempt exceed the limit, that account will be locked and a **LockedException** will be thrown to generate relevant error message on the front-end (see line 149 to 154 on figure 2.16). If the returned exception is a **LockedException**, it means the account used in the authentication is currently locked, and remained until the lock time is over. In this case, it requires 2 consecutive login attempts with the correct credential, else the account will be locked again for the other 20 minutes.
  8. The modelling of a failed login attempt in an account-based attack prevention is based on the **Attempts** (see figure 2.17) class in package *user.model* and the associated repository **AttemptsRepository** in package *user.repository*. The lock time of an account that misses the correct password depends on the field `Date lastModified` in the unlocking logic mentioned in previous step.
  9. The authentication failure handling in **LoginFailureHandler** class ends up with a redirection to `/login?error=true` with the response object that contains the relevant exception. (see line 69 to 71 on figure 2.14), that will bring us to the *AuthenticationController* class methods **login()** and **getErrorMessage()** in package *user.controller*, that in charge of returning relevant hint message about the reason of authentication failure.
  10. Figure 2.18, 2.19 and 2.20 show how the **getErrorMessage()** method results with the exception type given.

## Why this security control is effective

This security control is effective because it sets a limit for failing the login attempts consecutively, and denies any login attempt from the locked account and IP address regardless of the correctness of the credentials once the limit is exceeded. Thus, it will decrease the efficiency of both types of brute-force attacks, making it is harder to gain access to the application.

```

16 @Service
17 public class LoginAttemptService {
18
19     private static final Logger logger = LoggerFactory.getLogger(LoginAttemptService.class);
20
21     private static final int MAX_ATTEMPT = 3;
22
23     private final LoadingCache<String, Integer> attemptsCache;
24
25     public LoginAttemptService() {
26         super();
27         attemptsCache = CacheBuilder.newBuilder().expireAfterWrite(duration: 20, TimeUnit.MINUTES) // Locked time for 20 minutes
28             .build(new CacheLoader<String, Integer>() {
29
30                 @Override
31                 public Integer load(final String key) {
32                     return 0;
33                 }
34             });
35     }

```

Figure 2.15: Caching the IP address that cause the login failure attempt

```

136 // New fail attempt
137 if (userAttempts.isEmpty()) {
138     Attempts attempts = new Attempts(username, new Date());
139     attemptsRepository.save(attempts);
140     logger.warn(String.format("%d times of failed login attempts by <%=>", attempts.getAttempts(), username));
141     failed = new BadCredentialsException("Invalid credentials. Please try again");
142 } else {
143     Attempts attempts = userAttempts.get();
144     attempts.setAttempts(attempts.getAttempts() + 1);
145     attempts.setLastModified(new Date());
146     attemptsRepository.save(attempts);
147     logger.warn(String.format("%d times of failed login attempts by <%=>", attempts.getAttempts(), username));
148
149     if (attempts.getAttempts() + 1 > ATTEMPTS_LIMIT) {
150         user.setAccountNonLocked(false);
151         userRepository.save(user);
152         logger.warn(String.format("Failed login attempts by <%=> exceeds the consecutive limits, thus lock the a
153         failed = new LockedException("Too many invalid attempts. Your account will be locked for 20 minutes");
154     } else
155         failed = new BadCredentialsException("Invalid credentials. Please try again");
156 }

```

Figure 2.16: accountBasedFailureWithinLimit() method in

```

10 public class Attempts {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private int id;
15     private String username;
16     private int attempts;
17     private Date lastModified;
18
19     public Attempts() {}
20
21     public Attempts(String username, Date lastModified) {
22         this.username = username;
23         this.lastModified = lastModified;
24         this.attempts = 1;
25     }

```

Figure 2.17: Attempt modelling for account-based brute-force attack

---

**Invalid credentials. Please try again**

## Login

Username

Username

Figure 2.18: Error message for login with correct username but incorrect password within attempt limit

**Too many invalid attempts. Your account will be locked for 20 minutes**

## Login

Username

Username

Figure 2.19: Error message for login with correct username but incorrect password exceed the attempt limit, which the account will be locked from login for the next 20 minutes

**You are locked from further login attempts for 20 minutes.**

## Login

Username

Username

Figure 2.20: Error message for login with unregistered credential exceed the attempt limit, which the IP address will be locked from login for the next 20 minutes

---

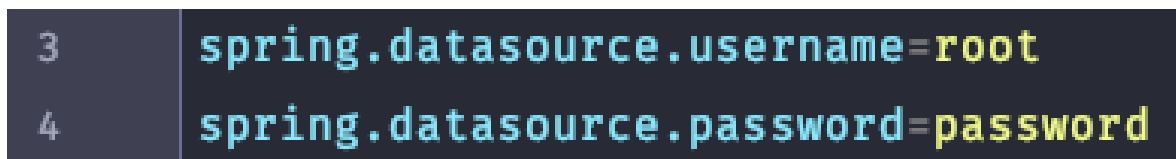
## Critical : Use of Hard-coded Password [CWE-259]

### Description

This web application was found to hard-coding the database credential onto the project configuration file, without any encryption. To rectify this, encryption of sensitive information on the configuration file is implemented to prevent the disclosure of sensitive data.

### Vulnerability location

- Code Location - Line 3-4 of *application.properties* file



```
3 spring.datasource.username=root
4 spring.datasource.password=password
```

Figure 2.21: Hard-coded database credential on configuration file without encryption

### Tasks included :

- Avoid hard-coded database credentials

### Security Control Implementation

1. [Jasypt](#) is imported as a Maven dependency with the following code added in *pom.xml* file

```
<dependency>
  <groupId>com.github.ulisesbocchio</groupId>
  <artifactId>jasypt-spring-boot</artifactId>
  <version>2.1.1</version>
</dependency>
```

2. After import this dependency, go to the corresponding maven library path on terminal (CLI) with the following command :

```
cd ~/.m2/repository/org/jasypt/jasypt/1.9.0
```

3. Run the following command on the terminal to generate the encoded output with the 3 required parameters

```
java -cp jasypt-1.9.0.jar org.jasypt.intf.cli.JasyptPBESStringEncryptionCLI
input="value" password=key algorithm=PBEWithMD5AndDES
```

- (a) **input** - Property value to be encrypted
- (b) **password** - The key for both encryption and decryption, which is required to run this application

- (c) **algorithm** - The encoding algorithm, that here we use a Password-Based version of DES Encryption algorithm, with the MD5 hash of the password as the encryption key

```
MINGW64 ~/m2/repository/org/jasypt/jasypt/1.9.0
$ java -cp jasypt-1.9.0.jar org.jasypt.intf.cli.JasyptPBEStringEncryptionCLI input="value" password=key algorithm=PBEwithMD5AndDES

----ENVIRONMENT-----
Runtime: Oracle Corporation Java HotSpot(TM) 64-Bit Server VM 14.0.2+12-46

----ARGUMENTS-----
input: value
password: key
algorithm: PBEwithMD5AndDES

----OUTPUT-----
09ZDmDxj1sn7EaJFxmSqew==
```

Figure 2.22: Encoded input as the output of Jasypt

4. Replace those sensitive information in *application.properties* config file with the output, by adding the prefix **ENC(** and suffix **)**

```
10 spring.datasource.username=ENC(1X4Pf7NqhgzugfmKdBBGHw==)
11 spring.datasource.password=ENC(0IJM1G6ejskVULoz0cxJxhthF17Q9FeB)
```

Figure 2.23: Add the output of Jasypt to application.properties (here the username is also encoded)

5. Jasypt reads all the properties from *application.properties* or any classpath properties that you passed, with value prefix **ENC(** and suffix **)**. Any configuration value in this file can be encoded with the same key to allow all of them to be decoded when the correct password given (Both the username and password are encoded in figure 2.23).
6. **@EnableEncryptableProperties** annotation is added into the configuration class (ie. line 29 of *WebSecurityConfig.java*), to tell Spring Boot to enable encryptable properties across the entire Spring Environment

```
27 @Configuration
28 @EnableWebSecurity
29 @EnableEncryptableProperties
30 @EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
31 public class WebSecurityConfig extends WebSecurityConfigurerAdapter{
```

Figure 2.24: Add the annotation to the configuration class

7. The argument **jasypt.encryptor.password** is required to run this application, with the encoding password used as the corresponding value. Full command to run the application is shown as below :

```
mvn -Djasypt.encryptor.password=key spring-boot:run
```

8. Reference : [Medium](#)



---

## Why this security control is effective

This solution allows the encryption of sensitive configuration information with Jasypt, which prevents the information disclosure of database credential in plaintext. It requires the attacker to figure out the secret key in order to run the application and have the access to the database. Also, the encoded database credential on the repository is just a placeholder, which is not used in production, hence the attacker has no way to access our production database besides an inefficient brute-force attack. In the web application pentesting report, cloud native secret manager was suggested which is a more secure option, with an alternative solution being Spring Cloud Vault, which is planned to be implemented in future.

---

## High : Weak Password Requirements [CWE-521]

### Description

There was no password policy established in BA application, which could makes the attempt of compromising a user account easier. Thus, NIST Special Publication [800-63B](#) is referenced to establish a new password policy. The following conditions must be satisfied to create a password on BA for accessing the associated account :

- At least 1 lowercase letter
- At least 1 uppercase letter
- At least 1 digit
- At least 1 special character
- At least 8 characters in length

Otherwise, the account registration and the changing of password will be rejected, by using input validation on server-side.

### Vulnerability location :

- Web Application Location - all password creation related pages
  - /register
  - /editPassword
  - /adminRegister (**new** page)
- Source Code Location (line in v1.0)
  - user/controller/UserController class - **createUser()** function (line 110)
  - user/controller/UserController class - **editPassword()** function (line 217)

### Tasks included :

- Establish password policy

### Security Control Implementation

1. The password strength visualizer functionality (see *password\_validation.js* in *resources* directory) is available on client-side for both pages mentioned above, to allow users recognize the strength of password in the corresponding input field currently.

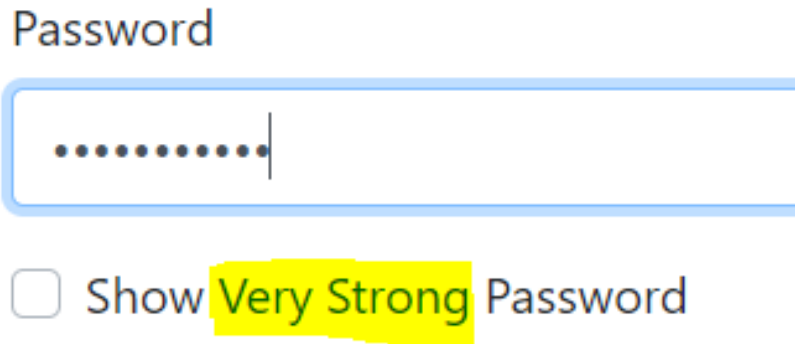


Figure 2.25: Password strength visualizer illustration

2. Checkbox for viewing the password in plaintext is also available, to allow the user to confirm their input password is following the password policy. (See `show_password.js` in *resources* directory)

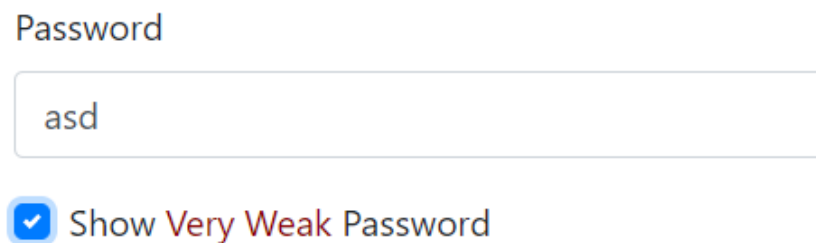


Figure 2.26: Show Password function illustration

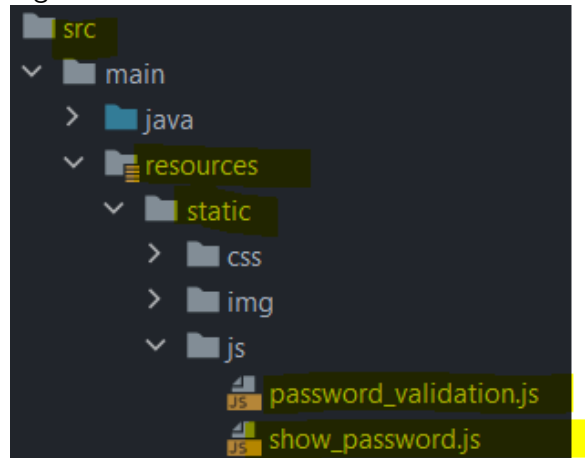


Figure 2.27: Path to these JavaScript source code

3. Standard password input validation has been implemented on client-side, ie. **required** attribute value on each HTML `<input>` element, including password-related field. It disallows the submission of empty value for necessary fields (See `<form>` elements in most HTML files)
4. Password input validation feature is also available on server-side, to ensure that the new user password is following the policy, else the POST request will be rejected with certain error messages on the relevant form. Example, figure 2.28 shows how the member account registration is rejected when the password in figure 2.26 was used (ie. *asd*)

Password

Password

☐ Show Password

**Invalid username or password.**

Figure 2.28: Password validation error message during registration

5. The definition of password policy can be found on line 11-37 of *password\_validation.js* for client-side, and line 18 of *RegexConstants.java* class for server-side (fig 2.29). The implementation of password validation on server-side is done on line 45-51 (fig 2.30) to validate password field in *User* object, and *validatePassword()* method on *UserValidator.java* class to validate new password for any account.

```
/**
 * (?=.*[a-z]) The string must contain at least 1 lowercase alphabetical character
 * (?=.*[A-Z]) The string must contain at least 1 uppercase alphabetical character
 * (?=.*[0-9]) The string must contain at least 1 numeric character
 * (?=.*[!@#$%^&*]) The string must contain at least one special character, but we are escaping reserved
 * (?=.{8,32}) The string must be eight characters or longer up to 32 characters
 */
public static final String PASSWORD_REGEX = "(?=.*[a-z])(?=.*\\d)(?=.*[A-Z])(?=.*[@#$%!*]).{8,32}";
```

Figure 2.29: Password policy definition on server-side

```
45 if ((!user.getPasswordConfirm().equals(user.getPassword())) ||
46     (!isValid(user.getPassword(), PASSWORD_REGEX)) ||
47     (user.getUsername().length() < 4 || user.getUsername().length() > 32) ||
48     (!isValid(user.getUsername())) ||
49     (userService.findByUsername(user.getUsername()) != null)
50 )
51     errors.rejectValue(s, "passwordConfirm", s1: "Diff.userForm.passwordConfirm");
```

Figure 2.30: Password validation implementation

## Why this security control is effective

The control implementation here should be effective since it allows the users to recognize their password strength during the creation when registering an account or update their password. Also, the checkbox that allows the showing of password in plaintext, lets the user to confirm their password, and make comparison to the password policy through the result of the password strength visualizer. Although limited client-side input validation were implemented, sufficient validations are done on server-side, to ensure every field are in expected regular expression (ie. regex). Thus, this security control should be able to prevent injection-based attacks such as SQL injection and Cross-Site Scripting.

# High : Cleartext Transmission of Sensitive Information [CWE-319]

## Description

BA is not using any encryption when transporting data. The BA web application communicates using HTTP. As a result, this makes it significantly easier for attackers to perform attacks such as MITM.

## Vulnerability location

```
POST /register HTTP/1.1
Host: localhost:8081
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.16; rv:84.0) Gecko/20100101 Firefox/84.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 153
Origin: http://localhost:8081
Connection: close
Referer: http://localhost:8081/register
Cookie: JSESSIONID=AE251C76652A4BEB71E73B05BA673B9F
Upgrade-Insecure-Requests: 1

name=Jane&surname=Doe&username=janed&password=p3G%2116t%21P6&passwordDuplicate=p3G%2116t%21P6
&phone=778-682-8212&email=janed%40gmail.com&address=New+York
```

Figure 2.31: MITM attack from Burp Suite

## Tasks included :

- Enforce the use of https.

## Security Control Implementation

1. Implement the following configuration in application.properties file which makes the spring boot application use SSL.

```
28 #SSL Key Info
29 security.require-ssl=true
30 server.ssl.key-store-type=PKCS12
31 server.ssl.key-store-password=password
32 server.ssl.key-store=src/main/resources/ba-insecurity-ssl-key.p12
```

2. Create a self-signed certificate by going to the terminal (CLI) and run the keytool command as below :

```
Microsoft Windows [Version 10.0.19041.985]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Wimself>keytool -genkeypair -alias selfsigned_localhost_sslserver -ext "SAN:c=DNS:localhost,IP:127.0.0.1" -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore ba2-insecurity-ssl-key.p12 -validity 3650
```

3. The self signed certificate is protected by password. Enter the password from the application.properties file and the other details as shown below. A self-signed certificate is generated afterwards.

```

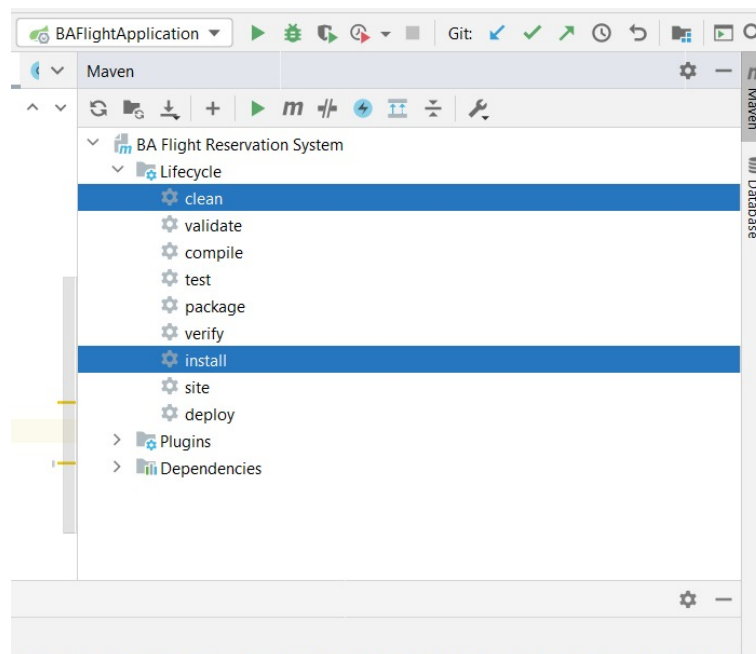
C:\Users\Himself>keytool -genkeypair -alias selfsigned_localhost_sslserver -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore ba-insecurity-ssl-key.p12 -validity 3650
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: BA inSecurity
What is the name of your organizational unit?
[Unknown]: inSecurity Assignment
What is the name of your organization?
[Unknown]: British Airways
What is the name of your City or Locality?
[Unknown]: Dublin
What is the name of your State or Province?
[Unknown]: County Dublin
What is the two-letter country code for this unit?
[Unknown]: IE
Is CN=BA inSecurity, OU=inSecurity Assignment, O=British Airways, L=Dublin, ST=County Dublin, C=IE correct?
[no]: y
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 3,650 days
for: CN=BA inSecurity, OU=inSecurity Assignment, O=British Airways, L=Dublin, ST=County Dublin, C=IE
C:\Users\Himself>

```

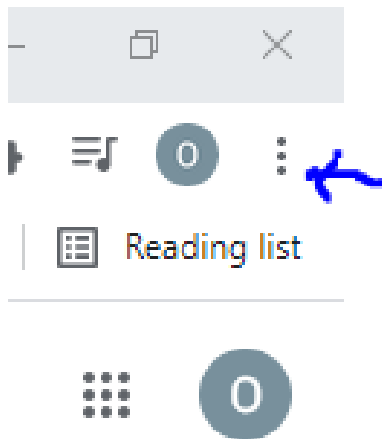
4. The PKS key is created and stored on your computer. Type name of the key in the search field in windows to locate key and move it to following directory from application.properties file, in ba-application source code.

src/main/resources

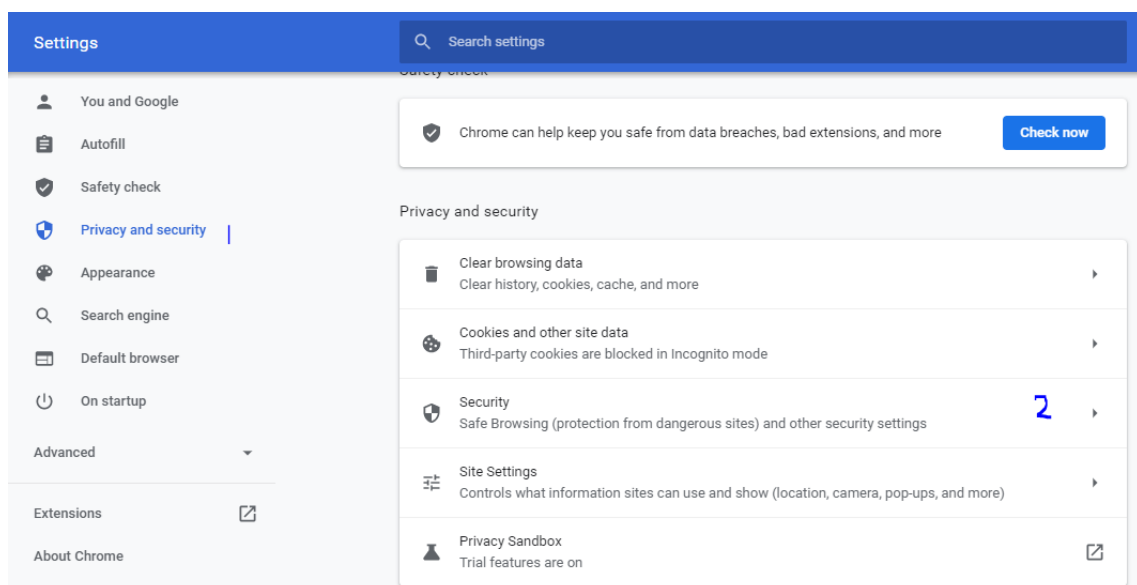
5. Clean the project by opening Maven projects panel, open the Lifecycle item. Run clean and install as seen below.



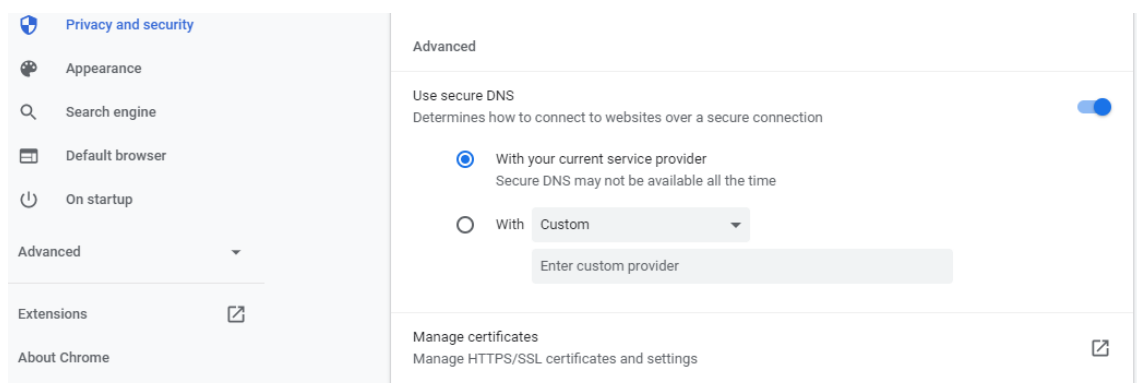
6. Open the browser(e.g. Google Chrome), click on the three dots on the top right-hand corner as seen below in screenshot.



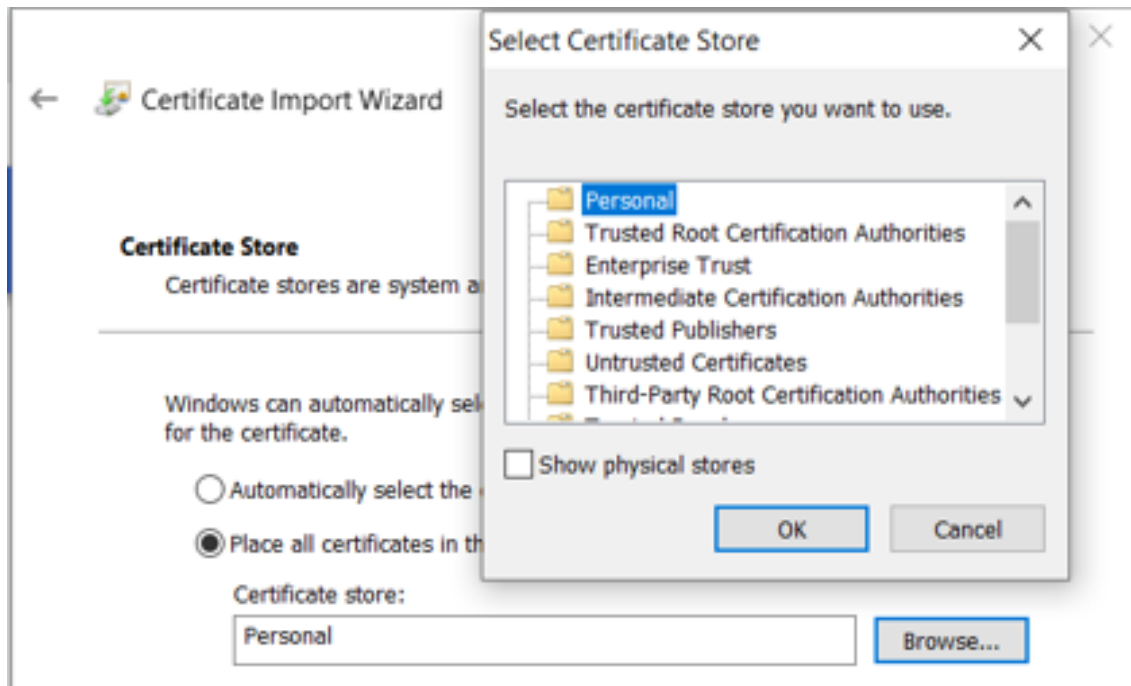
7. Click on "Settings", then click on "Privacy and security". This is then followed by clicking "Security".



8. The following page is displayed as seen below. Click on "Manage Certificates"



9. Certificate window is displayed. Click on "Trusted Root Certificate Authorities" at the top of window. You then click on "Imports".
10. Certificate Import Wizard is then displayed. Click "Next"
11. File to Import page is displayed. Click "Browse" to locate and import the certificate from your computer. Make sure the right file format of certificate is selected in the import box. Click "Next"
12. Enter the password from the configuration in application properties and Click Next
13. Click Browse and a Dialog Box is opened



14. Select Trusted Root Certification Authorities and Click "OK"
15. Click Next and the page below displays. Click Finish



## Completing the Certificate Import Wizard

The certificate will be imported after you click Finish.

You have specified the following settings:

Certificate Store Selected by User	Trusted Root Certification Authorities
Content	PFX
File Name	C:\Users\Himself\Desktop\ba-insecurity-ssl-key.p12

<  >

Finish Cancel

16. A warning dialog box is displayed that a certificate is about to be imported, Click "Yes"
17. A message indicating Certificate Import was successful is displayed.
18. Paste this in your chrome address bar:  
`chrome://flags/#allow-insecure-localhost`
19. Select "Enabled" next to highlighted text

Chrome | chrome://flags/#allow-insecure-localhost

Reset all

[#enable-google-srp-isolated-prerender-nsp](#)

●

**Allow invalid certificates for resources loaded from localhost.**

Allows requests to localhost over HTTPS even when an invalid certificate is presented. – Mac, Windows, Linux, Chrome OS, Android

[#allow-insecure-localhost](#)

Enabled ▼

20. A message is displayed stating "Your changes will take effect the next time you relaunch Google Chrome". Click "Relaunch" button.
21. The next few steps entails redirection of HTTP requests to HTTPS requests. Configure the Tomcat web server used on spring 2.x to enable HTTP traffic redirection to HTTPS inside a @Bean called ServletWebServerFactory as seen in screenshot.

```

@Bean
public ServletWebServerFactory servletContainer() {
    TomcatServletWebServerFactory tomcat = postProcessContext(context) -> {
        SecurityConstraint securityConstraint = new SecurityConstraint();
        securityConstraint.setUserConstraint("CONFIDENTIAL");
        SecurityCollection collection = new SecurityCollection();
        collection.addPattern("/*");
        securityConstraint.addCollection(collection);
        context.addConstraint(securityConstraint);
    };
    tomcat.addAdditionalTomcatConnectors(redirectConnector());
    return tomcat;
}

```

22. HTTP requests are set to port 8080 while HTTPS requests are set to port 8443. All HTTP requests are redirected to Tomcat's HTTPS port as seen in screenshot

```

private Connector redirectConnector() {
    Connector connector = new Connector( protocol: "org.apache.coyote.http11.Http11NioProtocol");
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    connector.setRedirectPort(8443); // Port 8443 in Apache Tomcat is used for running your s
    return connector;
}

```

23. The next few steps involves encryption of the password and key store type of the self signed certificate in the application.properties file using Jasypt. Add Jasypt maven dependency in the pom.xml file

```

<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot</artifactId>
    <version>2.1.1</version>
</dependency>

```

24. Run the following command on the terminal (CLI) to go to the corresponding maven library path

```
cd ~/.m2/repository/org/jasypt/jasypt/1.9.0
```

25. Run the following command on the terminal to generate the encoded output with the 3 required parameters

```
java -cp jasypt-1.9.0.jar org.jasypt.intf.cli.JasyptPBEStrEncryptionCLI
input="value" password=key algorithm=PBESWithMD5AndDES
```

- (a) **input** - password and key store type of the self-signed certificate to be encrypted
- (b) **password** - The key for both encryption and decryption, which is required to run this application
- (c) **algorithm** - The encoding algorithm, that here we use a Password-Based version of DES Encryption algorithm, with the MD5 hash of the password as the encryption key

26. The password and key store type for the ssl in application.properties file should be substituted with the output from previous step adding the prefix ENC(and suffix)

```
73 # The format/type used for the keystore. It could be set to JKS in case it is a JKS file
74 server.ssl.key-store-type=ENC(MErDLP2hkMEGmdb5JDoCww==)
75 # The path to the keystore containing the certificate
76 server.ssl.key-store=classpath:ba2-insecurity-ssl-key.p12
77 # The password used to generate the certificate
78 server.ssl.key-store-password=ENC(OIJM1G6ejskVULoz0cxJxhthF17Q9FeB)
```

Figure 2.32: HTTPS configuration in application.properties file

27. Jasypt reads all the properties from *application.properties* or any classpath properties that you passed, with value prefix **ENC(** and suffix **)**. Thus, any configuration value in this file can be encoded with the same key and got all of them decoded when the correct password given ( the password and key store type of the ssl certificate is encrypted).
28. **@EnableEncryptableProperties** annotation should be added to the Configuration class in WebSecurityConfig.java which makes the application to understand the encryptable properties across the entire Spring environment.

```
27 @Configuration
28 @EnableWebSecurity
29 @EnableEncryptableProperties
30 @EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
31 public class WebSecurityConfig extends WebSecurityConfigurerAdapter{
```

Figure 2.33: Add the annotation to the configuration class

29. The argument **jasypt.encryptor.password** is required to run this application, with the encoding password used as the corresponding value. Full command to run the application is shown as below :

```
mvn -Djasypt.encryptor.password=key spring-boot:run
```

## Why this security control is effective

The URL of the web application is insecure by default with HTTP meanwhile HTTPS provides Transport Layer Security. We chose Public Key Cryptographic Standards (PKCS12) to implement HTTPS because it is a much mature encryption which means it is widely supported. In addition, it is a language-neutral way to store encrypted private keys and certificates. The pentesters were able to carry out a man in the middle attack (MITM). With an HTTPS, it becomes impossible for an attacker to be able to eavesdrop because every sensitive data is being transmitted in a secure way by encryption between a web server and the browser.

---

## Medium : Insufficient Logging [[CWE-778](#)]

### Description

The application was logging minimal data, meaning when various security critical events were taking place, the software didn't make use of any recording mechanisms (e.g. printing out to console or storing into a file on the local disk).

**Vulnerability location:** There was no security critical event logging mechanism implemented in any of the classes but some occasional debugging print statements. This includes all the classes where user interaction is necessary such as **FlightController**, **ReservationController**, **AuthenticationController**, **CardController** and **UserController**

Tasks included:

- Perform appropriate logging using the log4j Java framework to record sensitive operations, such as logins, access to/modification of sensitive information (reservations, credit card information).

### Security Control Implementation

1. Excluded the default logger from **pom.xml**.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

2. Added log4j2 logger to **pom.xml**.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

3. Created a **LOGGER** object in each class that required logging by calling the **getLogger(<className.class>)** method provided by the **LoggerFactory** class.

```
private static final Logger LOGGER = LoggerFactory.getLogger(CardController.class);
```

4. Step 3 was only successful due to the importing of the following classes:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

5. Created a **log4j2.xml** file containing xml code that specifies the name of the file that will contain the log messages and the format in which the messages will be saved in.

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <configuration>
3
4     <Appenders>
5         <Console name="Console" target="SYSTEM_OUT" follow="true">
6             <PatternLayout>
7                 <pattern>%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss}} ${hostName}
8                 %highlight{${LOG_LEVEL_PATTERN:-%5p}}{FATAL=red blink, ERROR=red, WARN=yellow bold, INFO=green,
9                 DEBUG=green bold, TRACE=blue} %style{${sys:PID}}{magenta} [%15.15t] %style{%-40.40C{1.}}{cyan} :
10                %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>
11             </Console>
12
13         <File name="File" fileName="application.log">
14             <PatternLayout>
15                 <pattern>%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss}} ${LOG_LEVEL_PATTERN:-
16                %5p} ${sys:PID} [%15.15t] %-40.40C{1.} : %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>
17             </File>
18         </Appenders>
19
20     <Loggers>
21         <Root level="info">
22             <AppenderRef ref="Console"/>
23             <AppenderRef ref="File"/>
24         </Root>
25     </Loggers>
26 </configuration>
```

6. Disabled some of the default log messages coming from different packages related to the spring framework by using the **OFF** value in **application.properties**. Hence, we are only left with user interaction specific log messages that are of utmost importance to have on record.

```

31 # logging mechanism properties
32 spring.main.banner-mode=OFF
33 logging.level.sql=OFF
34 logging.level.apple=OFF
35 logging.level antlr=OFF
36 logging.level.web=OFF
37 logging.level.aj=OFF
38 logging.level.Class50=OFF
39 logging.level.com=OFF
40 logging.level.images=OFF
41 logging.level.java=OFF
42 logging.level.javassist=OFF
43 logging.level.javax=OFF
44 logging.level.jdk=OFF
45 logging.level.lombok=OFF
46 logging.level.net=OFF
47 logging.level.netscape=OFF
48 logging.level ognl=OFF
49 logging.level.org=OFF
50 logging.level.sun=OFF
51 logging.level.templates=OFF
52 logging.level.toolbarButtonGraphics=OFF
53 logging.level.org.springframework.boot=OFF
54 logging.level.org.springframework.beans=OFF
55 spring.main.log-startup-info=false

```

- Now, we are able to log messages anywhere in the code simply by creating a **LOGGER** object (as explained above) and using **LOGGER.<level>("<text>")** where level corresponds to one of **debug**, **info**, **error**, **warn**, **trace**.

```

LOGGER.warn("Unsuccessful attempt to add credit card details by a non-logged in user.");

```

```

LOGGER.info("Member credit card added for user <" + user.getUsername() + "> with the role of <" + userRoles + ">");

```

- Then, opening the file with the name we have specified in **log4j2.xml**, we can see log messages such as the following:

```

2021-May-19 19:54:53 INFO 31031 [nio-8443-exec-5] u.c.u.c.CardController : Called viewCreditCards(): by user <slav40> with the role of <MEMBER>
2021-May-19 19:56:18 INFO 31031 [nio-8443-exec-4] u.c.u.c.CardController : Member credit card added for user <slav40> with the role of <MEMBER>
2021-May-19 19:56:24 INFO 31031 [nio-8443-exec-1] u.c.u.c.CardController : Called viewCreditCards(): by user <slav40> with the role of <MEMBER>

```

## Why this security control is effective

This method is very effective as it allows developers to monitor what is happening with the application by being able to see log messages coming from critical events from a security perspective. As a result, malicious actions can be identified (e.g. a number of unsuccessful login attempts, etc.) or the smooth working of the application itself.

## Medium : Missing Custom Error Page [CWE-756]

### Description

The application was not returning custom error pages to the user. The in turn exposed sensitive information about the application which can potentially be used for a malicious attack by adversaries.

**Vulnerability location :** All the classes in the application that contain user interaction and methods that can return errors (e.g. **FlightController**, **ReservationController**, **CardController**, **UserController**, **AuthenticationController**).

**Tasks included:**

- Catch and manage all error pages in order to avoid disclosing sensitive information about the platform that you adopted to implement your web application.

### Security Control Implementation

1. Create a custom error page layout to be used throughout the application using JavaScript and HTML.

```
1 <!DOCTYPE HTML>
2
3 <html lang="en" xmlns:th="http://www.thymeleaf.org">
4
5 <head>
6     <meta charset="utf-8">
7     <meta http-equiv="X-UA-Compatible" content="IE=edge">
8     <title>BAFlight : Method Not Allowed</title>
9     <meta name="description" content="">
10    <meta name="viewport" content="width=device-width, initial-scale=1">
11
12    <div th:replace="fragments/header :: header-data"></div>
13    <link rel="stylesheet" href="/css/error.css">
14 </head>
15
16 <body>
17
18 <div id="error-content" class="container text-center align-middle">
19
20    <h1>405</h1>
21
22    <div class="container">
23        <p><em>HTTP 405 Method Not Allowed</em></p>
24        <p>Page or Server Not Found</p>
25
26        <div class="mt-5">
27            <p>: Return to :</p>
28            <a href="/" class="btn btn-light">Home</a>
29            <a href="javascript:history.back()" class="btn btn-primary">Back</a>
30        </div>
31    </div>
32 </div>
33
34 </body>
35
36 </html>
```

2. When an error occurs, the application uses a custom error page and appropriate messages to notify the user that something has gone wrong, however it doesn't reveal anything about

---

the application itself that can be used against it:

# 500

HTTP 500 Internal Server Error  
You are not allowed to access this page

: Return to :

[Home](#)

[Back](#)

# 401

HTTP 401 Unauthorized to access  
Unauthorized to access this page

: Return to :

[Home](#)

[Back](#)

3. In order to create new custom error pages, the same layout could be used with the only difference being the error code and the message to be returned to the user.

## Why this security control is effective

This security control is very effective due to the fact that it avoids exposing sensitive information and data of the application itself to the user, which in turn can be used to forge an attack against the application.



---

# High : Missing Encryption of Sensitive Data [CWE-311]

## Description

The application did not encrypt sensitive and important data as it was saved into the database. As a result, anyone who had access to the database could see plaintext sensitive information and use it to forge attacks against the application.

**Vulnerability location:** MySQL database used to store plaintext information about the credit card(s) a user is associated with.

**Tasks included:**

- Encrypt credit card and other sensitive information when storing this information in the database.

## Security Control Implementation

1. Created an **EncryptionService** class that uses the AES algorithm to encrypt and decrypt input strings. It uses the same pair of secret and salt keys in order to perform encryption and decryption. These keys are privately stored in the class.

```
public static String encrypt(String textToEncrypt) {
    try {
        byte[] iv = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
        IvParameterSpec ivspec = new IvParameterSpec(iv);

        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
       KeySpec spec = new PBEKeySpec(SECRET_KEY.toCharArray(), SALT.getBytes(), 65536, 256);
        SecretKey tmp = factory.generateSecret(spec);
        SecretKeySpec secretKey = new SecretKeySpec(tmp.getEncoded(), "AES");

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivspec);

        return Base64.getEncoder()
            .encodeToString(cipher.doFinal(textToEncrypt.getBytes(StandardCharsets.UTF_8)));
    } catch (Exception e) {
        LOGGER.error("Error while encrypting: " + e.toString());
    }
    return null;
}

public static String decrypt(String textToDecrypt) {
    try {
        byte[] iv = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
        IvParameterSpec ivspec = new IvParameterSpec(iv);

        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
        KeySpec spec = new PBEKeySpec(SECRET_KEY.toCharArray(), SALT.getBytes(), 65536, 256);
        SecretKey tmp = factory.generateSecret(spec);
        SecretKeySpec secretKey = new SecretKeySpec(tmp.getEncoded(), "AES");

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, secretKey, ivspec);

        return new String(cipher.doFinal(Base64.getDecoder().decode(textToDecrypt)));
    } catch (Exception e) {
        LOGGER.error("Error while decrypting: " + e.toString());
    }
    return null;
}
```

2. The service can now be used to encrypt/decrypt sensitive information. The following example shows how the service was used to encrypt and decrypt credit card details when adding a credit card to a member's account (which stores the details into the database) and viewing the credit card (via user-interaction on the front end side).

```

@PostMapping("/addMemberCreditCard")
public String addMemberCreditCard(String cardholder_name, String card_number, String card_type,
                                   int expiration_month, int expiration_year, String security_code, Model model, HttpServletRequest req) {
    User user = null;

    // encrypt card_number and security_code before saving it in the database
    String encryptedCardholderName = EncryptionService.encrypt(cardholder_name);
    String encryptedCardNumber = EncryptionService.encrypt(card_number);
    String encryptedCardType = EncryptionService.encrypt(card_type);
    String encryptedSecurityCode = EncryptionService.encrypt(security_code);

```

```

@PreAuthorize("#username == authentication.name")
@GetMapping("/viewCreditCards/{username}")
public String viewMemberCreditCards(@PathVariable(value = "username") String username, Model model, HttpServletRequest req) {
    User sessionUser = null;

    Principal userDetails = req.getUserPrincipal();
    if (userDetails != null) {
        sessionUser = userRepository.findByUsername(userDetails.getName());
        List<CreditCard> creditCards = sessionUser.getCredit_cards();

        for (CreditCard card : creditCards) {
            card.setCardholder_name(EncryptionService.decrypt(card.getCardholder_name()));
            card.setCard_number(EncryptionService.decrypt(card.getCard_number()));
            card.setType(EncryptionService.decrypt(card.getType()));
            card.setSecurity_code(EncryptionService.decrypt(card.getSecurity_code()));
        }
    }
}

```

- Then, logging into MySQL database, we can see the attributes for `card_number`, `cardholder_name`, `security_code` and `type` being encrypted.

```
mysql> select * from credit_cards;
```

id	card_number	cardholder_name	expiration_month	expiration_year	security_code	type	user_registrationId
11	0m5FNG8X00888977+c5dyd8d8cy8uak7Pd8q8	00T8p888u-j8k8u88888888	12	2008	818888888888888888888888	888888888888888888888888	9
12	6x78k8Qm/78814F7U7FD88877+WHZ8PzJl8y8Q8u8m	Al888888888888888888888888	1	2009	888888888888888888888888	888888888888888888888888	11

- On the front end side, the user sees the decrypted version of the credit card credentials.

Cardholder Name	Card Number	Card Type	Delete
nqika bojlova	1234 1234 6969 6969	Visa Debit	<button>Delete</button>

Add New Card
Back

## Why this security control is effective

This security control is effective because it essentially prevents adversaries from being able to view sensitive or important information in plaintext which would allow them to forge malicious attacks against the application and maybe impersonate a legit user.

---

## High : Cleartext Storage of Sensitive Information [CWE-312]

### Description

The application was storing user data (e.g. passwords) in the database as plaintext. As such, the data can be compromised and used by an attacker to forge attacks against the application.

**Vulnerability location :** MySQL database storing users' credentials with the password being one of them.

- Use BCrypt or a similar slow hashing functionality provided by SpringBoot to store your passwords.

### Security Control Implementation

1. Add the spring boot starter security dependency in the pom file.

```
67     <dependency>
68         <groupId>org.springframework.boot</groupId>
69         <artifactId>spring-boot-starter-security</artifactId>
70     </dependency>
```

2. Add **Role** and **User** models that will become tables in the database, along with their repository interfaces for auto-wiring.

```
7 @Entity
8 @Table(name = "role")
9 public class Role {
10     @Id
11     @GeneratedValue(strategy = GenerationType.IDENTITY)
12     private Long id;
13
14     private String name;
15
16     @ManyToMany(mappedBy = "roles")
17     private Set<User> users;
18
19     public Long getId() {
20         return id;
21     }
22
23     public void setId(Long id) {
24         this.id = id;
25     }
26
27     public String getName() {
28         return name;
29     }
30
31     public void setName(String name) {
32         this.name = name;
33     }
34
35     public Set<User> getUsers() {
36         return users;
37     }
38
39     public void setUsers(Set<User> users) {
40         this.users = users;
41     }
42 }
```

```

8 @Entity
9 @Table(name = "user")
10 public class User {
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Long id;
14
15     private String username;
16
17     private String password;
18
19     @Transient
20     private String passwordConfirm;
21
22     @ManyToMany
23     @JoinTable(name="user_roles",
24               joinColumns = @JoinColumn(name="user_id", referencedColumnName="id"),
25               inverseJoinColumns = @JoinColumn(name="role_id", referencedColumnName="id"))
26
27     private Set<Role> roles;
28
29     public Long getId() {
30         return id;
31     }
32
33     public void setId(Long id) {
34         this.id = id;
35     }
36
37     public String getUsername() {
38         return username;
39     }
40
41     public void setUsername(String username) {
42         this.username = username;
43     }
44
45     public String getPassword() {
46         return password;
47     }
48
49     public void setPassword(String password) {
50         this.password = password;
51     }
52
53     public String getPasswordConfirm() {
54         return passwordConfirm;
55     }
56
57     public void setPasswordConfirm(String passwordConfirm) {
58         this.passwordConfirm = passwordConfirm;
59     }
60
61     public Set<Role> getRoles() {
62         return this.roles;
63     }
64
65     public void setRoles(Set<Role> roles) {
66         this.roles = roles;
67     }
68 }

```

```

1 package demo.repository;
2
3 import demo.model.User;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface UserRepository extends JpaRepository<User, Long> {
7     User findByUsername(String username);
8 }

```

```

1 package demo.repository;
2
3 import demo.model.Role;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 public interface RoleRepository extends JpaRepository<Role, Long> {
7 }

```

3. Use **application.properties** to specify the name of the **.properties** file that will be stored inside the directory **resources**.

```

13 spring.messages.basename=validation

```

4. Provide a unique error message not to disclose sensitive/important information about the application by typing it in in the **validation.properties** file defined in the previous step.

```

2 Diff.userForm.passwordConfirm=Invalid username or password.

```

5. Validate the user details before adding them into the user database by adding the following code into the **UserController** class.

```

33 @PostMapping("/registration")
34 public String registration(@ModelAttribute("userForm") User userForm, BindingResult bindingResult) {
35     userValidator.validate(userForm, bindingResult);
36
37     if (bindingResult.hasErrors()) {
38         return "registration";
39     }
40     userService.save(userForm);
41
42     //securityService.autoLogin(userForm.getUsername(), userForm.getPasswordConfirm());
43     return "redirect:/welcome";
44 }

```

6. Add user validator to validate user's username and password by enforcing policies.

```

14 @Component
15 public class UserValidator implements Validator {
16     @Autowired
17     private UserService userService;
18
19     @Override
20     public boolean supports(Class<?> aClass) {
21         return User.class.equals(aClass);
22     }
23
24     @Override
25     public void validate(Object o, Errors errors) {
26         User user = (User) o;
27
28         //ValidationUtils.rejectIfEmptyOrWhitespace(errors, "username", "NotEmpty");
29
30         if ((user.getUsername().length() < 6 || user.getUsername().length() > 32) ||
31             (!isUserValid(user.getUsername())) ||
32             (userService.findByUsername(user.getUsername()) != null) ||
33             (!user.getPasswordConfirm().equals(user.getPassword())) ||
34             (!isStrong(user.getPassword())))
35         )
36             errors.rejectValue("passwordConfirm", "Diff.userForm.passwordConfirm");
37     }
38
39     private boolean isUserValid(String username){
40         return username.matches("^[a-z0-9_-]{6,32}$");
41     }
42
43     private static boolean isStrong(String password){
44         final String PASSWORD_PATTERN = "((?=.*[a-z])(?=.*\\d)(?=.*[A-Z])(?=.*[@#%!])).{8,32})";
45         Pattern pattern = Pattern.compile(PASSWORD_PATTERN);
46
47         Matcher matcher = pattern.matcher(password);
48
49         return matcher.matches();
50     }
51 }

```

7. Add password strength computation in the front end by using **JavaScript**.

```

1  function CheckPasswordStrength(password) {
2      var password_strength = document.getElementById("password_strength");
3
4      //TextBox left blank.
5      if (password.length == 0) {
6          password_strength.innerHTML = "";
7          return;
8      }
9
10     //Regular Expressions.
11     var regex = new Array();
12     regex.push("[A-Z]"); //Uppercase Alphabet.
13     regex.push("[a-z]"); //Lowercase Alphabet.
14     regex.push("[0-9]"); //Digit.
15     regex.push("[$@!%*#?&]"); //Special Character.
16     var passed = 0;
17
18     //Validate for each Regular Expression.
19     for (var i = 0; i < regex.length; i++) {
20         if (new RegExp(regex[i]).test(password)) {
21             passed++;
22         }
23     }
24
25     //Validate for length of Password.
26     if (passed > 2 && password.length > 8) {
27         passed++;
28     }
29
30     //Display status.
31     var color = "";
32     var strength = "";
33     switch (passed) {
34         case 0:
35             case 1:
36                 strength = "Very Weak";
37                 color = "darkred";
38                 break;
39             case 2:
40                 strength = "Weak";
41                 color = "red";
42                 break;
43             case 3:
44                 strength = "Fair";
45                 color = "darkorange";
46                 break;
47             case 4:
48                 strength = "Strong";
49                 color = "green";
50                 break;
51             case 5:
52                 strength = "Very Strong";
53                 color = "darkgreen";
54                 break;
55     }
56     password_strength.innerHTML = strength;
57     password_strength.style.color = color;
58 }

```

8. Configure user settings in **WebSecurityConfig** class.

```

27  @Override
28  protected void configure(HttpSecurity http) throws Exception {
29      http
30          .authorizeRequests()
31          .antMatchers("/resources/**", "/registration").permitAll()
32          .anyRequest().authenticated()
33          .and()
34          .formLogin()
35          .loginPage("/login")
36          .permitAll()
37          .and()
38          .logout()
39          .permitAll();
40  }

```

9. Add additional services to perform user authentication, update user details and retrieve user details, and their corresponding public interfaces.

```

15 @Service
16 public class SecurityServiceImpl implements SecurityService{
17     @Autowired
18     private AuthenticationManager authenticationManager;
19
20     @Autowired
21     private UserDetailsService userDetailsService;
22
23     private static final Logger logger = LoggerFactory.getLogger(SecurityServiceImpl.class);
24
25     @Override
26     public String findLoggedInUsername() {
27         Object userDetails = SecurityContextHolder.getContext().getAuthentication().getDetails();
28         if (userDetails instanceof UserDetails) {
29             return ((UserDetails)userDetails).getUsername();
30         }
31         return null;
32     }
33
34     @Override
35     public void autoLogin(String username, String password) {
36         UserDetails userDetails = userDetailsService.loadUserByUsername(username);
37         UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new
38             UsernamePasswordAuthenticationToken(userDetails, password, userDetails.getAuthorities());
39
40         authenticationManager.authenticate(usernamePasswordAuthenticationToken);
41
42         if (usernamePasswordAuthenticationToken.isAuthenticated()) {
43             SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
44             logger.debug(String.format("Auto login %s successfully!", username));
45         }
46     }
47 }

```

```

20 @Service
21 public class UserDetailsServiceImpl implements UserDetailsService{
22     @Autowired
23     private UserRepository userRepository;
24
25     @Override
26     @Transactional(readOnly = true)
27     public UserDetails loadUserByUsername(String username) {
28         User user = userRepository.findByUsername(username);
29         if (user == null) throw new UsernameNotFoundException(username);
30
31         Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
32         for (Role role : user.getRoles()) {
33             grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));
34         }
35
36         return new org.springframework.security.core.userdetails.User(user.getUsername(), user.
37             getPassword(), grantedAuthorities);
38     }
39 }

```

```

12 @Service
13 public class UserServiceImpl implements UserService {
14     @Autowired
15     private UserRepository userRepository;
16     @Autowired
17     private RoleRepository roleRepository;
18     /* @Autowired
19     private BCryptPasswordEncoder bCryptPasswordEncoder;*/
20
21     @Override
22     public void save(User user) {
23         // user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
24         user.setRoles(new HashSet<>(roleRepository.findAll()));
25         userRepository.save(user);
26     }
27
28     @Override
29     public User findByUsername(String username) {
30         return userRepository.findByUsername(username);
31     }
32 }

```

```

3 public interface SecurityService {
4     String findLoggedInUsername();
5
6     void autoLogin(String username, String password);
7 }

```

```

5 public interface UserService {
6     void save(User user);
7
8     User findByUsername(String username);
9 }

```

10. Built on top of that by providing better security and slow hashing by increasing the number of log rounds to 12.

```
@Override
public void save(User user) {
    bcryptPasswordEncoder = new BCryptPasswordEncoder( strength: 12);
    user.setPassword(bcryptPasswordEncoder.encode(user.getPassword()));
}
```

Passwords were originally encoded with 10 log rounds:

10	10	pragha	ahhha	kekoe@gmail.com	banko	\$2a\$10\$K0GgRL8P15K/c9T1S.8M0VxytHz7RnGOWATZuU0htnztGNG	8812345678	banko	banko
11	1324	slav		kekoe@gmail.com	slav	\$2a\$10\$hbDC7VvVv8GpD1j4uWV.gmR5v8S5mHf1aveH7J1eHaar8h1G	8899991234	slav	slav40

However now, passwords are encoded with 12 log rounds:

registrationid	address	email	name	password	phone	surname	username	
8	10	vanko street	slav40@gmail.com	slav	\$2a\$12\$X8DUpc1z2nsdxy18n/G682oQ0Fn3pJyEgMyR5E8QMcCaX.HFToD	8899991432	denisov	slav4o
9	15	nexus better	dekonarka@gmail.com	banana	\$2a\$12\$Y0zqYQhcS8BManG6bF/hkeavKvD13x5AyyVPI2Q6Sp.PiLiVMZV13n	8896969696	banana	banana

## Why this security control is effective

This security control is effective as it prevents the application from storing users' passwords in clear text in the database by using hashing and salting. This avoids the possibility of forging various malicious attack given that an adversary has access to the database.



---

## Moderate : Improper Input Validation [CWE-20]

### Description

The application was allowing unfiltered text to be entered when submitting a registration form, allowing attacks like XSS to be carried out.

#### Vulnerability location:

- Client Side: In the registration html pages, such as **register.html** and **registerCreditCard.html**
- Server Side: In the relevant @PostMapping methods in the Controllers, such as **/register** and **/addMemberCreditCard**

#### Tasks Include:

- Use input validation frameworks at the client and the server sides.

### Security Control Implementation

#### • Client Side

1. JavaScript functions were created which assesses the given values based on an appropriate regex.

```
function passengerSurnameValidator(name){  
  
    let surname_validity = document.getElementById( elementId: "surname_validity");  
    const nameRegex = /^[a-zA-Z]+(([', . -][a-zA-Z ])?[a-zA-Z]*)*$/  
  
    if(name.value.match(nameRegex)){  
        surname_validity.innerHTML = "";  
        return true;  
    }  
    else{  
        surname_validity.innerHTML = "Invalid Surname Given.";  
        surname_validity.style.color = "red";  
        return false;  
    }  
}
```

2. These functions were applied to the input fields of the HTML forms used to create data, such as register.html, registerCreditCard.html and passengerDetails.html.

```
<input id="surname" class="form-control" type="text" name="surname" placeholder="Surname" th:field="*{surname}"  
onchange="passengerSurnameValidator(document.memberForm.surname)" onkeydown="clearErrors()" required/>
```

3. This results in a form that cannot be submitted until all of the regex conditions are met, filtering out any unwanted/ potentially malicious characters.

---

## Register your account

First Name

Invalid Name Given.

Surname

Invalid Surname Given.

- **Server Side**

1. Custom Validator classes were created (located in the validator package) to assess the object created by Thymeleaf @Valid @ModelAttribute, returned by the HTML form.
2. If any errors in the field values are encountered by the relevant validator, then the client is returned to the form with relevant error messages.

```
@PostMapping("/{register}")
public String register(@Valid @ModelAttribute("userForm") User userForm, BindingResult bindingResult,
    Model model, HttpServletRequest req) throws ServletException {

    userValidator.validate(userForm, bindingResult);

    if (bindingResult.hasErrors()) {

        LOGGER.warn("Unable to register user with username <" + userForm.getUsername() + ">");
        return "register.html";
    }
}
```

3. The object will not be stored in the db until it passes the validator tests.

## Why this security control is effective

User input validation is provided by the JS methods in real-time, meaning that well-intentioned clients will be able to better informed as to how to correctly submit forms to the service. Malicious users will be unable to proceed with illegal and potentially malicious characters in their forms.

Should the JS scripts fail to load or be circumvented, further validation will be carried out server-side, which will also return appropriate error messages to the client. Should the regex being used be found to be inadequate, it is a simple matter of changing the values found in **filter/Regex-Constants**, allowing for an easily modifiable validation framework.

---

# Critical : Use of GET Request Method with Sensitive Query Strings [CWE-598]

## Description

BA has several RESTful API routes that allow an attacker to easily configure query strings to access information they should not have access to.

### Vulnerability location

- Client Side: In the html pages where update and deletion can be performed such as `viewAllFlights.html` and `updateReservation.html`
- Server Side: In the relevant `@GetMapping` methods such `/deleteFlight/{flightID}` in the `FlightController` and `/user/delete/{username}` in the `UserController`

### Tasks included :

- Do not allow modifications of sensitive data using GET requests.

## Security Control Implementation

- **Client Side:** A form was created whereby the corresponding method in the `UserController` was specified but not the inclusion of the username parameter. The username is included in the input tag with type **hidden**. The entire code is seen in the screenshot below



```
66 <form th:action="@{/user/delete}" method="post" class="horizontal-button">
67   <input type="hidden" id="username" name="username" th:value="${user.getUsername()}">
68   <button type="submit" class="btn btn-danger">Delete</button>
69 </form>
```

- **Server Side:** The corresponding method in the `UserController` included changing the method from `@GetMapping` to `@PostMapping` and removal of `{username}` parameter from method signature. The changes made in the method signature can be seen below



```
176 // Delete a registration record
177 @PreAuthorize("username == authentication.name or hasAuthority('ADMIN')")
178 @PostMapping("/user/delete")
179 public void deleteRegistration(@RequestParam String username, HttpServletRequest req, HttpServletResponse response) throws UserNotFoundException, IOException {
180     Principal userDetails = req.getUserPrincipal();
181     User sessionUser = userRepository.findByUsername(userDetails.getName());
182     User user = userRepository.findByUsername(username);
```

- This change made in this example ensures that the sensitive can be sent from the browser to the server without it being visible in the query string.
- Similar patterns were followed for updates or deletion throughout the application

## Why this security control is effective

This solution is effective because sensitive information related to flights, reservations and users can no longer be exposed through the URL. This would prevent a scenario where attackers could perform CRUD operations if those sensitive data were present in the query string. The data is

---

now being sent in the body of the HTTP and using this solution with Secure HTTP ensures that there is indeed secure transmission of data from browser to server.

If an attacker uses a web accelerator, using a POST request for update and delete scenarios ensures that the application does not execute links that are clicked by the accelerator unlike using a GET request.

---

# Critical: Failure to Restrict URL Access [CWE-817]

## Description

By not performing authorisation checks on who can access specific URLs, attackers can easily access endpoints that provide views and functionalities that compromise the service if used maliciously. The definition of User functions by endpoints associated with User IDs also allows for an attacker to easily brute force multiple endpoints in succession without prior knowledge of User details.

## Vulnerability location

- Pre-authorisation and endpoint naming in UserController, FlightController, ReservationController and CardController.
- Pattern matching in WebSecurityConfig.

## Tasks Include:

- Avoid using User IDs to access specific resources
- Implement appropriate access control to only allow the user associated with a specific account to modify his/her flights and credit card transactions. Also apply access control policies to regulate access to the urls that you have identified in your app.

## Security Control Implementation

- User Roles are defined in the Roles table in the db, and these roles are assigned to User accounts upon their creation. These roles are then used to verify the User's privileges to access specific endpoints.

```
@Override
public void adminSave(User user) {
    bCryptPasswordEncoder = new BCryptPasswordEncoder( strength: 12);
    user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
    user.setAccountNonLocked(true);
    Role userRole = roleRepository.findByName("ADMIN");
    Set<Role> roleSet = new HashSet<>();
    roleSet.add(userRole);
    user.setRoles(roleSet);
    userRepository.save(user);
}
```

- Endpoints have been renamed to direct based on a Member's username, not their User ID. This allows for authorisation to a User's username instead of their particular ID.
- In WebSecurityConfig, pattern matching to the specific endpoints defines which type of User will be able to access an endpoint.

```
http.cors().and().csrf().disable() httpSecurity
    .requiresChannel().anyRequest().requiresSecure() // Require HTTPS Requests
    .and()
    .authorizeRequests() ExpressionUrlAuthorizationConfigurer->ExpressionInterceptUrlRegistry
    .antMatchers( _andPatterns: "/error", "/resources/**", "/img/**", "/css/**", "/js/**", LOGIN_URL, "/register", "/", "/questRegister").permitAll()
    .antMatchers( _andPatterns: "/user", "/user/delete/", "/processMemberPayment").access( attribute: "hasAnyAuthority('ADMIN','MEMBER')")
    .antMatchers( _andPatterns: "/editProfile", "/editPassword", "/viewCreditCards").access( attribute: "hasAuthority('MEMBER')")
    .antMatchers( _andPatterns: "/admin", "/adminRegister", "/users", "/flights", "/reservations", "/deleteReservation", "/questRegister", "/adminRegister",
```

- 
- In the Controllers, the `@PreAuthorize` annotation is applied to methods where access should be restricted. Building on the pattern matching in `WebSecurityConfig`, the annotation can handle more complicated expressions, allowing for the access of endpoints by User's with either Admin clearance or being a Member with the matching username to the URL.

```
@PreAuthorize("#username == authentication.name or hasAuthority('ADMIN')")
@GetMapping("/{users/{username}")
@ResponseBody
public User getRegistrationByUsername(@PathVariable(value = "username") String username, HttpServletRequest req) throws UserNotFoundException {
```

## Why this security control is effective

By switching the endpoint variables to usernames instead of arbitrary User ID values, the efficacy of brute force attacks when User details are unknown is greatly diminished. An attacker cannot effectively iterate through usernames like they would IDs. By defining roles and pre-authorizing endpoints, URLs can be effectively whitelisted, further preventing them from being probed and attacked.

---

## Chapter 3: Extras

---

### MySQL to h2 migration

#### Description

The application had to migrate from MySQL database to the persistent h2 database to allow for admin user to be already in the system upon the starting of the application.

#### Implementation

1. Add the pom dependency.

```
104         <dependency>
105             <groupId>com.h2database</groupId>
106             <artifactId>h2</artifactId>
107             <scope>runtime</scope>
108             <version>1.4.200</version>
109         </dependency>
```

2. Enabled the use of **h2-console** by applying the proper settings within the **WebSecurity-Config** class.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    // enable h2 access via the h2-console
    http.authorizeRequests().antMatchers("/h2-console/**").permitAll()
        .and().csrf().ignoringAntMatchers("/h2-console/**")
        .and().headers().frameOptions().sameOrigin();
}
```

3. Configure the h2 database settings with **application.properties** and made sure it is *persistent* by storing it in a local file.

```
7 spring.h2.console.enabled=true
8 spring.datasource.url=jdbc:h2:file:./data/fileDb
9 spring.datasource.driverClassName=org.h2.Driver
10 spring.datasource.username=admin
11 spring.datasource.password=password
12 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

4. It can then be used by navigating to *localhost:8080/h2-console* and supplying the following username and password credentials: **admin** : **password**.

English
Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:file:./data/fileDb

User Name: admin

Password:

Connect Test Connection

5. Then we can navigate through it and look at the different tables and their contents.

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM USERS

REGISTRATIONID	ADDRESS	EMAIL	NAME	PASSWORD	PHONE	SURNAME	USERNAME	ACCOUNT_NON_LOCKED
1	10 john smith	johnsmith@gmail.com	john	\$2a\$12\$gUZWwHhHBLQDSyaszUOISfBzRJsGAGSLtmZx8aMtWwqVv6	089 1234 1234	smith	john_smith	TRUE
2	69 zuckerberg avenue	zuckerberg@gmail.com	mark	\$2a\$12\$eAKUUVOCy8VWVWgZEf69.5IcaHsJfGhP5RcpgSGWthM6LTly	089 971 5341	zuckerberg	zuckerberg	TRUE
3	3 HAROLD'S GRANGE ROAD	ag@gmail.com	Home	\$2a\$12\$5mUd8f3ZozD9KZzVgGZuZz9gHscuZDGCspRT8f9ZD8TEOQW	123	Goh	vincent	TRUE
35	15 unknown street	guest@gmail.com	unknown	\$2a\$12\$8dWfY4pCeydRgue.TCuZ7A.D0bRuHfY1qZ68Q1H8u4Pv2	085 889 1441	host	testguest	TRUE

Run

Run Selected

Auto complete

Clear

SQL statement:

SELECT \* FROM FLIGHTS

FLIGHTID	ARRIVAL_DATE_TIME	CANCEL_LIMIT_TIME	DEPARTURE_DATE_TIME	DESTINATION	SOURCE
1	2021-08-06 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Sofia, Bulgaria
2	2021-08-06 13:30:00	null	2021-08-06 11:30:00	Dublin, Ireland	Stockholm, Sweden
3	2021-08-05 10:30:00	null	2021-08-07 08:00:00	Dublin, Ireland	Oslo, Norway
4	2021-08-05 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Rio de Janeiro, Brazil
5	2021-08-05 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Bucharest, Romania
6	2021-08-05 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Manila, Philippines
7	2021-08-05 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Seoul, South Korea
8	2021-08-05 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Pyongyang, North Korea
9	2021-08-05 00:30:00	null	2021-08-05 20:30:00	Dublin, Ireland	Tokyo, Japan
10	2021-08-02 00:30:00	null	2021-08-01 20:30:00	Dublin, Ireland	Sofia, Bulgaria
11	2021-08-04 00:30:00	null	2021-08-03 20:30:00	Dublin, Ireland	Sofia, Bulgaria
12	2021-08-13 00:30:00	null	2021-08-12 20:30:00	Dublin, Ireland	Sofia, Bulgaria
13	2021-08-12 13:30:00	null	2021-08-12 11:30:00	Dublin, Ireland	Stockholm, Sweden
14	2021-08-12 10:30:00	null	2021-08-12 08:00:00	Dublin, Ireland	Oslo, Norway

## Why this security control is effective

Since the application will not allow admin registration, our team needed a persistent database that will contain an admin user out of the box ready to be used. These criteria were satisfied by the h2 database, hence a decision was made to migrate from MySQL to h2.