Risk Analysis

Name: Tony Jiang

Semester: 6

Contents

Introduction

This document outlines the security risks that can occur in the music trivia web-based game and evaluates whether they are covered in the project and how they would be addressed. Using the top 10 OWASP also help define some security risks for this project.

Risk identification

These risks are in no particular order.

The “impact” is how long it would take to implement it in the project. Where 1 is low and 5 is high.

The “likelihood” is how likely the security risk can happen, where 1 is low and 5 is high.

Risk ID	Risk Description	Impact(1-5)	Likelihood(1-5)	Implemented
1	Unauthorized access to the song management section of the website.	4	2	**Yes**
2	Storage of user passwords in the database without encryption.	1	1	**Yes**
3	Unauthenticated user access to user information	4	3	**Yes**
4	Inject malicious SQL queries through input fields to manipulate the database.	2	3	**Some**
5	Inject malicious scripts into web pages viewed by other users.	3	3	**No**
6	Attackers can trick users into performing unwanted actions on the web app, such as changing account settings.	2	4	**No**

7	Users accessing data or functions they should not have access to by manipulating URLs or parameters.	2	4	**Some**
8	Exposing sensitive functionalities or data through APIs without proper security.	4	4	**Yes**
9	Users finding ways to cheat or manipulate game outcomes to gain unfair advantages.	5	5	**No**
10	Sensitive data being exposed through error messages or debug information.	2	2	**No**

Mitigation plan

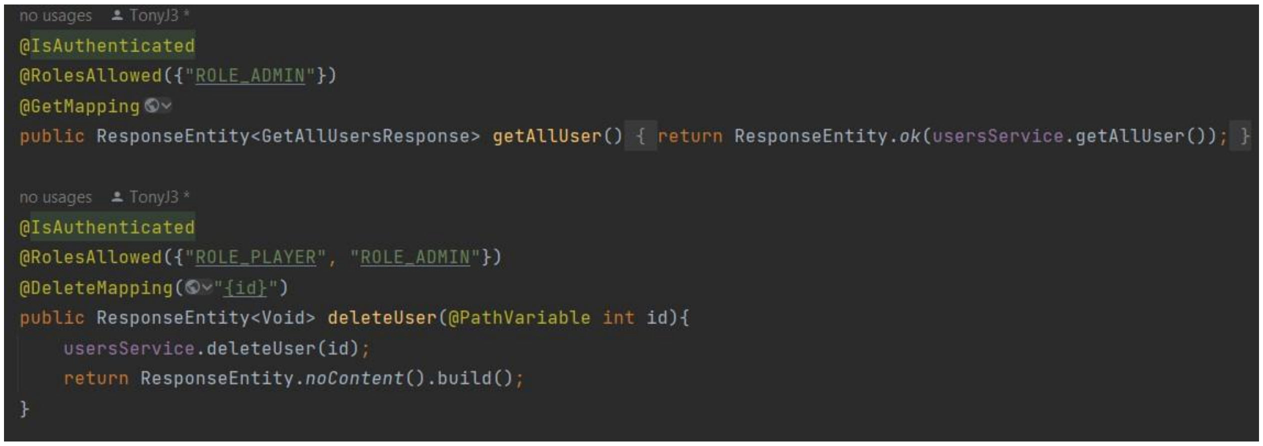
This section outlines how the identified security problems will be addressed. It includes implementation strategies or solutions to solve the problem. These measures are to be taken into consideration.

1. Unauthorized access to the song management section of the website.

Mitigation: Implement user roles.

Evidence:

Each function in the controller is associated with a specific user role that determines who can access that function. If a function is not assigned a role, it can be accessed by anyone.



```
no usages   TonyJ3 *
@IsAuthenticated
@RolesAllowed({"ROLE_ADMIN"})
@GetMapping
public ResponseEntity<GetAllUsersResponse> getAllUser() { return ResponseEntity.ok(usersService.getAllUser()); }

no usages   TonyJ3 *
@IsAuthenticated
@RolesAllowed({"ROLE_PLAYER", "ROLE_ADMIN"})
@DeleteMapping("{id}")
public ResponseEntity<Void> deleteUser(@PathVariable int id){
    usersService.deleteUser(id);
    return ResponseEntity.noContent().build();
}
```

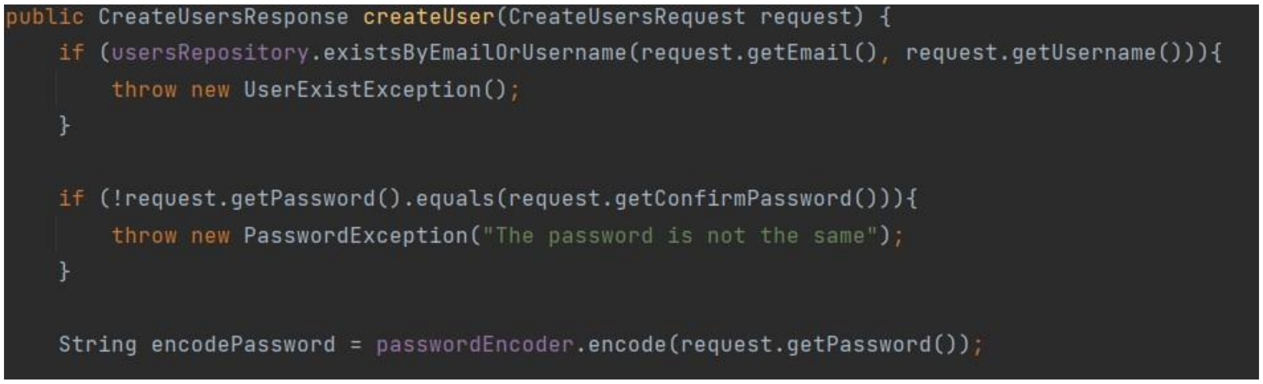
Validate: The validation will be done by using an automated security tool called OWASP Zap to find any security vulnerability.

2. Storage of user passwords in the database without encryption.

Mitigation: Implement password hashing.

Evidence:

I have implemented the password hashing when you sign up in the application. I use the bcrypt algorithm hashing. Bcrypt is wildly use that doesn't use a lot of memory and performance to hash and it's one of the moderate hashing securities.



```
public CreateUsersResponse createUser(CreateUsersRequest request) {
    if (usersRepository.existsByEmailOrUsername(request.getEmail(), request.getUsername())){
        throw new UserExistException();
    }

    if (!request.getPassword().equals(request.getConfirmPassword())){
        throw new PasswordException("The password is not the same");
    }

    String encodePassword = passwordEncoder.encode(request.getPassword());
```

Validate: The validation will be done by use an automated security tool call OWASP Zap to scan if any plaintext passwords are not transmitted or stored.

3. Unauthenticated user access to user information

Mitigation: Implement multi-factor authentication.

Evidence:

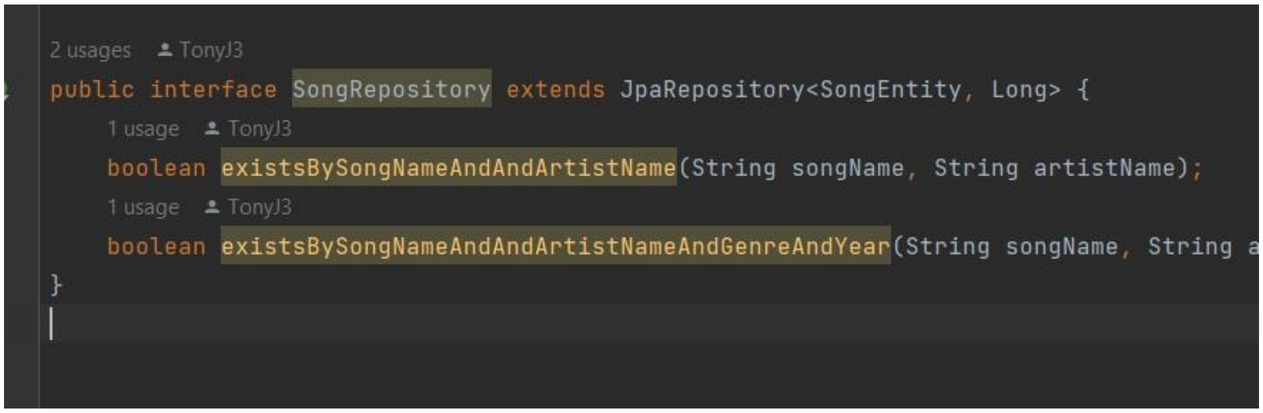
Validate: Use an automated security tools to scan for vulnerabilities that could allow unauthenticated access.

4. Inject malicious SQL queries through input fields to manipulate the database.

Mitigation: Spring Data JPA, which generate parameterized SQL queries and provide protection against SQL injection attacks.

Evidence:

I use Spring Data JPA naming conventions and parameter binding to prevent SQL injection attacks.



```
2 usages   TonyJ3
public interface SongRepository extends JpaRepository<SongEntity, Long> {
    1 usage   TonyJ3
    boolean existsBySongNameAndAndArtistName(String songName, String artistName);
    1 usage   TonyJ3
    boolean existsBySongNameAndAndArtistNameAndGenreAndYear(String songName, String a
}
```

Validate: Use automated security tools to scan for SQL injection vulnerabilities.

5. Inject malicious scripts into web pages viewed by other users.

Mitigation: Validate all user input field, especially data that is displayed on web pages, to ensure it does not contain any executable code.

Evidence: Not implemented.

Validate: Use automated security tool called XSS testing frameworks to scan the application for XSS vulnerabilities.

6. Attackers can trick users into performing unwanted actions on the web app, such as changing account settings.

Mitigation: Require users to provide password verification to change user’s information.

Evidence: Not implemented.

Validate: Use automated security tool called OWASP Zap to simulate various attack scenarios and assess the effectiveness of security controls.

7. Users accessing data or functions they should not have access to by manipulating URLs or parameters.

Mitigation: Implement roll-based access control to define roles and permissions for users, allowing them to access only the data and functions relevant to their roles.

Evidence: Not implemented.

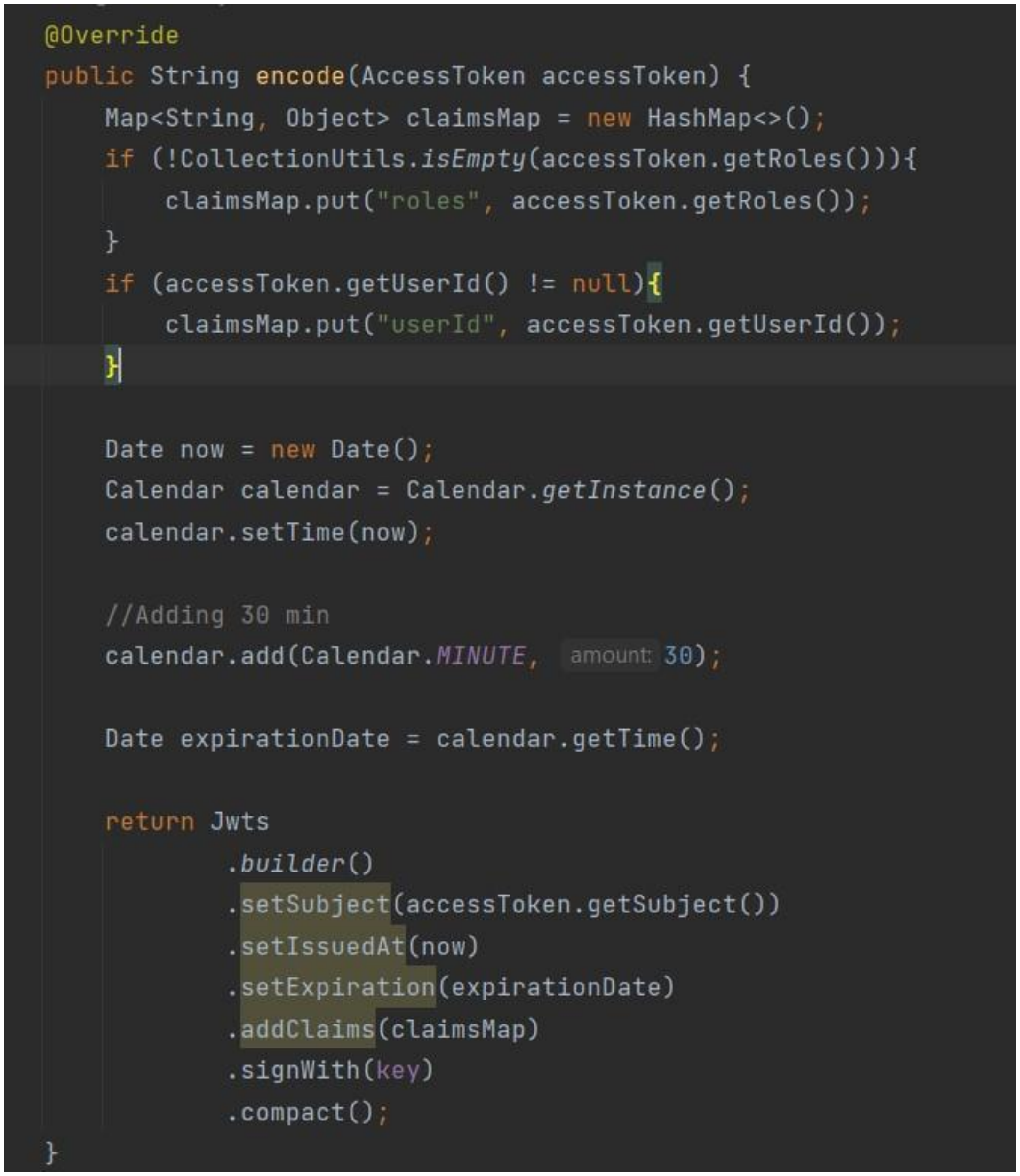
Validate: Use automated tool called OWASP ZAP to simulate various attack scenarios and assess the effectiveness of access controls.

8. Exposing sensitive functionalities or data through APIs without proper security.

Mitigation: Implement strong authentication mechanisms such as OAuth 2.0 or JWT (JSON Web Tokens) to ensure that only authenticated and authorized users can access sensitive APIs.

Evidence:

I implemented JWT access tokens to authenticate and authorize users to access sensitive APIs.



```java
@Override
public String encode(AccessToken accessToken) {
    Map<String, Object> claimsMap = new HashMap<>();
    if (!CollectionUtils.isEmpty(accessToken.getRoles())){
        claimsMap.put("roles", accessToken.getRoles());
    }
    if (accessToken.getUserId() != null){
        claimsMap.put("userId", accessToken.getUserId());
    }

    Date now = new Date();
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(now);

    //Adding 30 min
    calendar.add(Calendar.MINUTE,  amount: 30);

    Date expirationDate = calendar.getTime();

    return Jwts
            .builder()
            .setSubject(accessToken.getSubject())
            .setIssuedAt(now)
            .setExpiration(expirationDate)
            .addClaims(claimsMap)
            .signWith(key)
            .compact();
}
```

Validate: Use automated tool called OWASP ZAP to simulate various attack scenarios and assess the effectiveness of security controls.

9. Users finding ways to cheat or manipulate game outcomes to gain unfair advantages.

Mitigation: Implement anti-cheat measures such as cheat detection algorithms, integrity checks, and anti-tamper mechanisms to detect and prevent cheating attempts.

Evidence: Not implemented.

Validate: Use bots or scripts to automate player actions and verify that anti-cheat measures are functioning as intended.

10. Sensitive data being exposed through error messages or debug information.

Mitigation: Catch exceptions and errors, log them with minimal details, and display a generic error message to users.

Evidence: Not implemented.

Validate: Use automated tool called OWASP ZAP to scan for error messages that expose sensitive data and assess the effectiveness of security controls.

Conclusion

These are the risks that I have looked into and consider for the music trivia web-based game, along with how to mitigate them, provided evidence of the implementation, and outlined how they would be validated. More risks will be added and examined.