Hemtenta Spring Security

Hot 1: SQL Injection *(20 poäng totalt)*

G-frågor *(12 poäng)*

1.1 Hotet och er implementation *(8p)*

- **Förklara hur SQL injection fungerar mot REST APIs** ***(3p)***

  SVAR:

  SQL injection i ett REST API sker när en användares indata skickas direkt in i ett SQL statement utan att säkra den genom t.ex. validering eller prepared statements. Angripare kan då skriva sin input med avsikt att köra egen SQL kod som kan läsa, ändra och radera data som denna användare inte har rätt till.

  > **Exempel:**
  >
  > Om API:t frågar efter ett namn och någon skickar `"name": 'Jerry' OR '1'='1'` , kan frågan bli:
  >
  > ```
  > SELECT * FROM users WHERE name = 'Jerry' OR '1'='1';
  > ```
  >
  > vilket kommer visa alla användare.

- **Visa kodexempel från era JPA Repository-operationer som skyddar mot SQL injection** ***(5p)***

  SVAR:

  ```
  @Query("SELECT b FROM Book b JOIN Author a ON b.authorId = a.id WHERE " +
          "LOWER(b.title) LIKE LOWER(CONCAT('%', :searchQuery, '%')) OR " +
          "LOWER(CONCAT(a.firstName, ' ', a.lastName)) LIKE " +
          "LOWER(CONCAT('%', :searchQuery, '%'))")
  List<Book> searchBooksByTitleOrAuthor(@Param("searchQuery") String
  searchQuery);
  ```

Parametisering i min Query här skyddar mot SQL injection genom att JPA binder användarens input (`:searchQuery`) som en parameter istället för att direkt placera den i SQL-strängen. Detta gör så att eventuella specialtecken som `'` , `;` eller SQL-kommandon inte tolkas som kod utan som en vanlig sträng. Detta förhindrar att angripare kan manipulera queryn för att köra oönskade SQL-operationer.

1.2 Skyddsmekanismen *(4p)*

- **Förklara hur JPA/Hibernate förhindrar SQL injection med prepared statements**

  SVAR:

  JPA/Hibernate använder prepared statements för att skilja SQL-kod från användarinput. Värden som kopplas med `:param` skickas separat till databasen och behandlas alltid som data, inte kod. Specialtecken som `'` eller `;` kan därför inte manipulera queryn. Så även om någon skulle skriva `"; DROP TABLE Books;"` i min sökfunktion för books så skulle det tolkas som att man söker efter en bok som heter så, inte som skadlig SQL kod.

Hot 2: Trasig autentisering *(30 poäng totalt)*

G-frågor *(18 poäng)*

2.1 Hotet och lösenordssäkerhet *(10p)*

- **Beskriv autentiseringsproblem som kan drabba REST APIs *(4p)***

  SVAR:

  Autentiseringsproblem i REST APIs uppstår när användare kan komma åt data eller funktioner utan korrekt kontroll av identitet. Vanliga brister är svaga lösenord, dålig hantering av tokens eller att API:t saknar skydd mot brute force-attacker. Vid brute force-problem provar angripare systematiskt alla möjliga lösenord. Sådana brister kan leda till att angripare tar över konton eller får åtkomst till känslig information. För att skydda API:t bör man använda HTTPS och begränsa antalet inloggningsförsök, till exempel genom att blockera användaren i en viss tid efter tre felaktiga försök.

- **Visa kodexempel på BCrypt-hashing och lösenordsvalidering *(6p)***

  SVAR:

  I `/register` i min authController encodar jag password för nya användare såhär:

```
User user = new User();
user.setFirstName("user");
user.setLastName("userson");
user.setEmail(signUpRequest.getUsername());
// passwordEncoder.encode() använder BCrypt för att skapa ett säkert
hashat lösenord
user.setPassword(passwordEncoder.encode(signUpRequest.getPassword()));
user.setRole("USER");
userRepository.save(user);
```

Sedan i `/login` använder jag Spring Securitys AuthenticationManager för att validera användarens inloggning:

```
// jämför automatiskt det inmatade lösenordet mot det hashade lösenordet i
databasen
Authentication authentication = authenticationManager.authenticate(
    new UsernamePasswordAuthenticationToken(
        loginRequest.getUsername(),
        loginRequest.getPassword()
    )
);
```

2.2 Session-baserad autentisering *(8p)*

- **Förklara hur sessions fungerar för ert API och vad som händer vid inloggning**

  SVAR:

  Mitt API använder stateless autentisering med JWT istället för traditionella serverbaserade sessioner. Det innebär att servern inte lagrar någon information om inloggade användare mellan anrop. När en användare loggar in, genereras en JWT (JSON Web Token) som innehåller användarens identitet, roller och ett utgångsdatum (i mitt fall cirka 16 minuter framåt). Denna token skickas tillbaka till klienten.

  Vid varje efterföljande API-anrop skickar klienten med token i Authorization-headern. Servern validerar token genom att kontrollera signaturen och utgångstiden för att säkerställa att anropet kommer från en autentiserad användare.

VG-frågor *(12 poäng)*

2.3 JWT implementation *(12p)*

- **Visa kod för JWT-generering och validering *(8p)***

  SVAR:

  Token genereras med användarnamn, utgångstid och signeras med hemlig nyckel:

```
String jwt = Jwts.builder()
        .setSubject(userDetails.getUsername())
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() +
jwtExpirationMs))
        .signWith(SignatureAlgorithm.HS512, jwtSecret)
        .compact();
```

  Validering sker genom att försöka parsa token med samma hemliga nyckel:

```
boolean valid = Jwts.parser()
        .setSigningKey(jwtSecret)
        .parseClaimsJws(token)
        .getBody() != null;
```

  Användarnamn kan hämtas från token för autentisering:

```
String username = Jwts.parser()
        .setSigningKey(jwtSecret)
        .parseClaimsJws(token)
        .getBody()
        .getSubject();
```

- **Förklara fördelarna med stateless authentication för REST APIs *(4p)***

  SVAR:

  Eftersom all info om användaren ligger i token behöver servern inte hålla koll på någon session. Varje server kan kolla token själv, vilket gör det lätt att lägga till fler servrar (load balancing) och skala systemet. Det blir också säkrare eftersom man slipper vissa session-attacker och mer tåligt om en server skulle krascha.

Hot 3: Trasig auktorisering *(30 poäng totalt)*

G-frågor *(18 poäng)*

3.1 Hotet och rollbaserad säkerhet *(12p)*

- **Beskriv privilege escalation i APIs med exempel *(4p)***

  SVAR:

  Privilege escalation i APIs uppstår när en användare får högre åtkomst än de borde ha, t.ex. genom att manipulera request-data. Till exempel kan en vanlig användare ändra sin roll i en JSON-body från:

  ```
  {
      "username": "Jerry",
      "role": "user"
  }
  ```

  till:

  ```
  {
      "username": "Jerry",
      "role": "admin"
  }
  ```

  Om API:t inte kontrollerar detta, kan användaren få administratörsrättigheter och komma åt funktioner eller data de egentligen inte ska ha tillgång till.

- **Visa er Spring Security konfiguration för endpoint-skydd baserat på roller *(8p)***

  SVAR:

  ```
  http.authorizeHttpRequests(auth -> auth
      // Endpoints som bara admin får använda
      .requestMatchers(HttpMethod.POST, "/api/books/**").hasRole("ADMIN")
      .requestMatchers(HttpMethod.POST, "/api/authors/**").hasRole("ADMIN")
      .requestMatchers(HttpMethod.PUT,
  "/api/loans/{id}/extend**").hasRole("ADMIN")
      .requestMatchers("/api/users/**").hasRole("ADMIN")
      // Loans endpoints som både USER och ADMIN får använda
      .requestMatchers("/api/loans/**").hasAnyRole("USER", "ADMIN")
      // Resten är öppet
      .anyRequest().permitAll()
  );
  ```

Den här konfigurationen skyddar mot att vanliga användare försöker bli admin genom att ändra sin roll i requesten. Spring Security kollar alltid användarens riktiga roll från JWT, och alla POST/PUT-endpoints som kräver admin är låsta. Även om en vanlig användare försöker ändra sin roll i JSON-body går det alltså inte. På så sätt kan ingen fuska sig till högre privilegier.

3.2 HTTP-statuskoder *(6p)*

- **Förklara skillnaden mellan 401 och 403, ge exempel från ert API**

  SVAR:

  401 Unauthorized: Användaren är inte inloggad eller skickar ingen giltig JWT. T.ex: `POST /api/books` utan token → 401.

  403 Forbidden: Användaren är inloggad men har inte rätt roll. T.ex: vanlig användare utan admin roll försöker `POST /api/books` → 403.

VG-frågor *(12 poäng)*

3.3 Metodsäkerhet *(8p)*

- **Visa @PreAuthorize-annotationer från era service-klasser**

3.4 Objektbaserad säkerhet *(4p)*

- **Hur implementerar ni att User A inte kan komma åt User B:s data?**

Hot 4: Exponering av känslig data *(12 poäng totalt)*

G-frågor *(8 poäng)*

4.1 Dataskydd *(8p)*

- **Förklara vilka typer av känslig data som kan läcka via APIs** *(2p)*

  SVAR:

  Känslig data som kan läcka via API kan vara t.ex. personnummer, kreditkortsnummer, lösenord och annan konfidentiell information. Det är därför mycket viktigt att API:t skyddar känslig information så det inte läcker ut till personer som inte har rätt till denna informationen.

- **Visa hur ni använder DTOs för att filtrera känslig data från API-responses** *(6p)*

  SVAR:

  Jag använder UserDTO för att skicka användardata utan lösenord till klienten. Alla endpoints (`getAllUsers` , `getUserByEmail` , `addUser`) returnerar UserDTO istället för User. På så sätt exponeras aldrig känslig information som lösenord.

```
// UserDTO filtrerar bort lösenord
public class UserDTO {
    private Long id;
    private String name;
    private String email;
    private LocalDateTime registrationDate;

    public UserDTO(Long id, String firstName, String lastName,
                   String email, LocalDateTime registrationDate) {
        this.id = id;
        this.name = firstName + " " + lastName;
        this.email = email;
        this.registrationDate = registrationDate;
    }
}

// Användning av DTO i controller
List<UserDTO> users = userService.getAllUsers();
return ResponseEntity.ok(users);
```

VG-frågor *(4 poäng)*

4.2 Avancerad datakryptering *(4p)*

- **Visa implementation av kryptering av känslig data utöver lösenord**

Hot 5: Säkerhetskonfigurationsfel *(8 poäng totalt)*

G-frågor *(4 poäng)*

5.1 Spring Security konfiguration *(4p)*

- **Visa er SecurityConfiguration-klass och förklara vilka endpoints som är öppna vs skyddade**

  SVAR:

  Alla POST- och PUT-endpoints är skyddade med ADMIN-roll, förutom `/api/loans/{id}/extend` som även kräver admin. Övriga loans-endpoints kan användas av både USER och ADMIN. Alla GET-metoder är öppna eftersom de endast läser data och inte kan ändra något. På så sätt styrs åtkomsten på servernivå via Spring Security och JWT, oberoende av vad klienten skickar i requesten.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http,
                                       AuthEntryPointJwt authEntryPointJwt,
                                       AuthAccessDeniedHandler
authAccessDeniedHandler)
                                       throws Exception {
    http
        .cors().and()
        .csrf().disable()
        .exceptionHandling(ex -> ex
            .authenticationEntryPoint(authEntryPointJwt)   // 401
            .accessDeniedHandler(authAccessDeniedHandler)  // 403
        )
        .sessionManagement(sm -> sm
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            // ADMIN-only endpoints
            .requestMatchers(HttpMethod.POST,
"/api/books/**").hasRole("ADMIN")
            .requestMatchers(HttpMethod.POST,
"/api/authors/**").hasRole("ADMIN")
            .requestMatchers(HttpMethod.PUT, "/api/loans/{id}/extend**")
                .hasRole("ADMIN")
            .requestMatchers("/api/users/**").hasRole("ADMIN")
```

```
            // USER or ADMIN for all other /api/loans/**
            .requestMatchers("/api/loans/**").hasAnyRole("USER", "ADMIN")

            // Everything else is open
            .anyRequest().permitAll()
        );

    http.authenticationProvider(authenticationProvider());
    http.addFilterBefore(authenticationJwtTokenFilter(),
                         UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

VG-frågor *(4 poäng)*

5.2 Avancerad konfiguration *(4p)*

- **Visa implementation av antingen säkra HTTP-headers ELLER rate limiting (välj ett)**