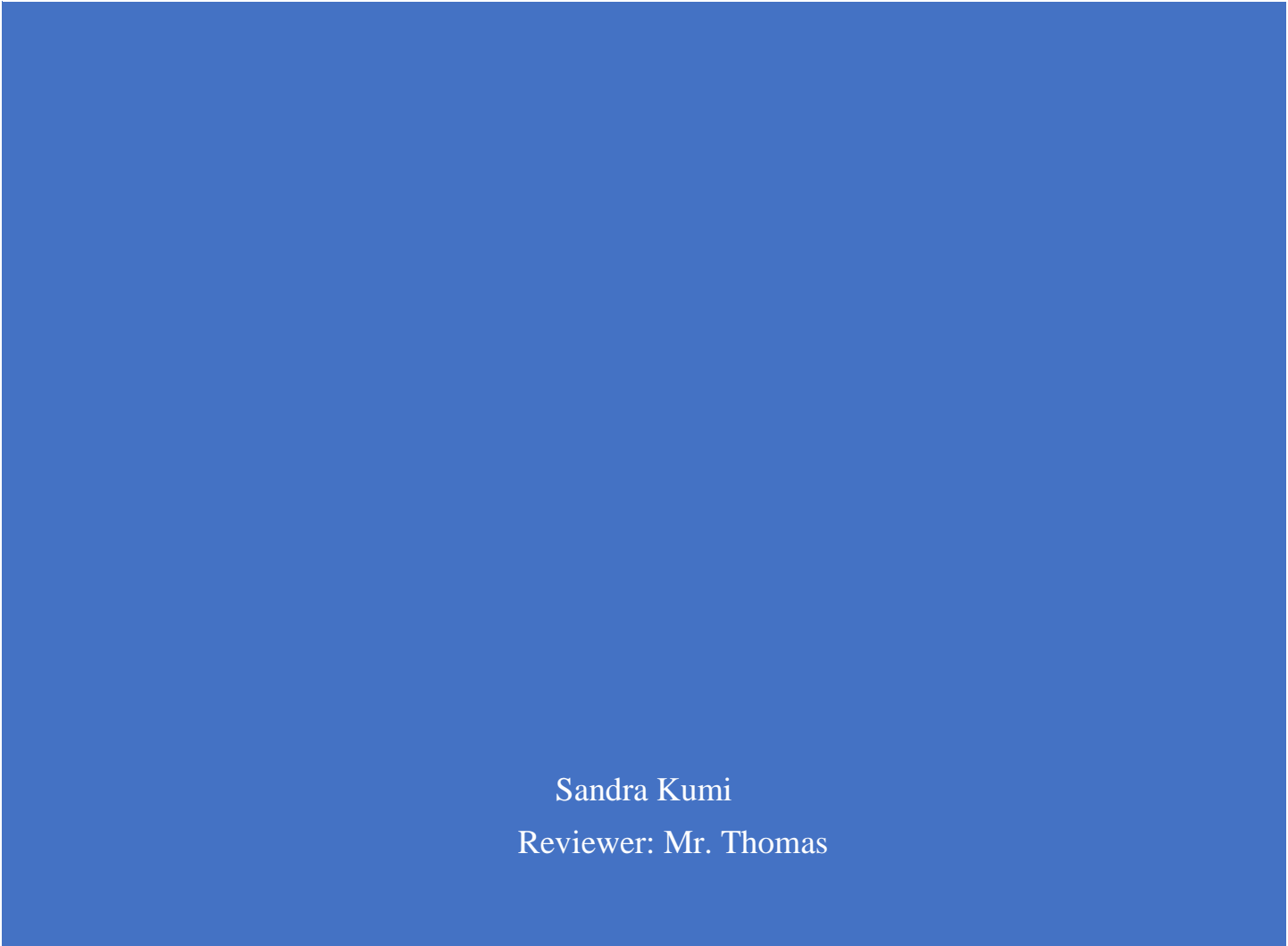




AUTHENTICATION



Sandra Kumi
Reviewer: Mr. Thomas

Authentication, Authorization, and PKI in a Spring Boot Application

1. Introduction

In modern web applications, security is paramount. This document provides a detailed overview of the authentication and authorization mechanisms implemented in our Spring Boot application, as well as the Public Key Infrastructure (PKI) setup for secure communication. These mechanisms ensure that only authorized users can access certain resources, and that all communication between the client and server is secure.

2. Authentication

2.1 What is Authentication?

Authentication is the process of verifying the identity of a user or system. It ensures that the entity attempting to access the application is who they claim to be. In our application, we implement two types of authentication: Basic Authentication and Token-Based Authentication using JSON Web Tokens (JWT).

2.2 Types of Authentication Implemented

2.2.1 Basic Authentication

- **Description:** Basic authentication involves sending the username and password in the HTTP header encoded in Base64. Although simple, it is not recommended for secure applications as it exposes credentials in transit unless the connection is secured with HTTPS.
- **Implementation:** Basic authentication is not directly implemented in our application, but understanding it is crucial as a foundation. The `UsernamePasswordAuthenticationFilter` in Spring Security would handle this if enabled.

// Basic Authentication can be enabled by adding the following in the security configuration:

```
http.httpBasic();
```

However, we rely on more secure authentication mechanisms, particularly JWT, for our application.

2.2.2 Token-Based Authentication (JWT)

- **Description:** Token-based authentication uses JSON Web Tokens (JWT) to securely exchange information between the client and server. Upon successful login, the server generates a token that the client must include in the Authorization header for subsequent requests. This token is stateless and contains encoded information about the user.
- **Implementation:**
 - **JWT Generation:** Implemented in the JwtUtil class, the token is generated after successful user authentication. The token includes claims like username and roles, and it is signed with a secret key.

```
public String generateToken(UserDetails userDetails) {  
    Map<String, Object> claims = new HashMap<>();  
    return createToken(claims, userDetails.getUsername());  
}
```

```
private String createToken(Map<String, Object> claims, String subject) {  
    return Jwts.builder()  
        .setClaims(claims)  
        .setSubject(subject)  
        .setIssuedAt(new Date(System.currentTimeMillis()))  
        .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY *  
1000))  
        .signWith(SignatureAlgorithm.HS512, secret)  
        .compact();  
}
```

- **User Login:** The UserService class handles login requests, verifies credentials, and returns a JWT.

```

public String loginUser(String username, String password) throws AuthenticationException {
    // Authenticate the user and generate a JWT token

    Authentication authentication = new UsernamePasswordAuthenticationToken(userDetails,
password, userDetails.getAuthorities());

    SecurityContextHolder.getContext().setAuthentication(authentication);

    return jwtUtil.generateToken(userDetails);
}

```

- **Token Validation:** The JwtRequestFilter class intercepts each request, extracts the JWT, and validates it before the request reaches the controller.

@Override

```

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain chain)

```

```

    throws ServletException, IOException {
    final String requestTokenHeader = request.getHeader("Authorization");

```

```

    String username = null;

```

```

    String jwtToken = null;

```

```

    // JWT token is in the form "Bearer token". Remove Bearer word and get only the token.

```

```

    if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {
        jwtToken = requestTokenHeader.substring(7);
        try {
            username = jwtUtil.getUsernameFromToken(jwtToken);
        } catch (IllegalArgumentException | ExpiredJwtException | MalformedJwtException e) {
            logger.error("Unable to get JWT Token or JWT Token has expired");
        }
    }
}

```

```

// Once we get the token, validate it.
if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
    UserDetails userDetails = this.userService.loadUserByUsername(username);

    // If token is valid, set authentication
    if (jwtUtil.validateToken(jwtToken, userDetails)) {
        UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =
            new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());

        usernamePasswordAuthenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
    }
}
chain.doFilter(request, response);
}

```

2.3 Authentication Flow

1. **User Login:** The user sends login credentials (username and password) via a POST request to the `/api/public/login` endpoint.
2. **Credential Verification:** The `UserService` verifies the credentials against stored user data using `UserDetailsService`.
3. **Token Generation:** Upon successful verification, a JWT is generated using the `JwtUtil` class and returned to the user.
4. **Token Usage:** The user includes the JWT in the Authorization header as `Bearer <token>` for all subsequent requests to access protected resources.

3. Authorization

3.1 What is Authorization?

Authorization is the process of determining what resources and actions an authenticated user is allowed to access. It controls access based on the user's role and permissions. In our application, authorization is implemented using Role-Based Access Control (RBAC).

3.2 Role-Based Access Control (RBAC)

- **Description:** RBAC restricts access to resources based on the user's role within the application. Roles such as USER and ADMIN define what actions a user can perform.
- **Implementation:**
 - **Roles Defined:** Roles are assigned to users during registration. For instance, a user can be assigned either a USER or ADMIN role.

```
public User registerUser(UserRequest userRequest) {  
    String encodedPassword = passwordEncoder.encode(userRequest.getPassword());  
    User user = new User();  
    user.setUsername(userRequest.getUsername());  
    user.setPassword(encodedPassword);  
    user.setRole(userRequest.getRole()); // Assigns role based on user request  
    userRepository.save(user);  
    return user;  
}
```

- **Access Control:** The SecurityConfig class defines access control for various API endpoints using the authorizeHttpRequests method. Endpoints are secured based on user roles.

@Bean

```
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
    http.csrf(AbstractHttpConfigurer::disable)  
        .authorizeHttpRequests(authorize -> authorize
```

```

        .requestMatchers("/api/authenticate", "/api/public/**").permitAll()
        .requestMatchers("/api/admin/**").hasAuthority("ADMIN")
        .requestMatchers("/api/private/**").hasAnyAuthority("ADMIN", "USER")
        .anyRequest().authenticated()
    )
    .sessionManagement(session -> session
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
    );

http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

return http.build();
}

```

- **Example:** Only users with the ADMIN role can access /api/admin/** endpoints, while both USER and ADMIN roles can access /api/private/** endpoints.

3.3 Authorization Flow

1. **Role Assignment:** Roles (USER, ADMIN) are assigned to users during registration.
2. **Resource Request:** The user attempts to access a protected resource by sending a request with a valid JWT.
3. **Role Verification:** The system checks the user's role encoded in the JWT and determines if access to the requested resource is permitted.
4. **Access Granted/Denied:** Based on role verification, the system either grants or denies access to the requested resource. For example, an ADMIN user can access /api/admin/**, while a USER cannot.

4. Public Key Infrastructure (PKI)

4.1 What is PKI?

Public Key Infrastructure (PKI) is a framework for managing digital keys and certificates that enable secure communication and data exchange over the internet. PKI relies on the use of a pair of keys: a public key and a private key. This system helps in establishing trust between entities communicating over an insecure network.

4.2 PKI Components

- **Public Key:** Used for encryption. It is openly shared and can be distributed freely. It is included in the digital certificate.
- **Private Key:** Used for decryption. It is kept secret and known only to the owner. It is used to sign messages or decrypt data encrypted with the public key.
- **Certificates:** Digital documents that bind a public key with an entity's identity, issued by a trusted Certificate Authority (CA). The certificate ensures that the public key belongs to the claimed owner.

4.3 PKI in the Application

4.3.1 HTTPS Configuration

- **Description:** HTTPS (HyperText Transfer Protocol Secure) uses SSL/TLS to encrypt communication between the client and server. This encryption is achieved through PKI, where SSL certificates containing the public key are exchanged to establish a secure session.
- **Implementation:**
 - **Key Generation:** Use tools like OpenSSL or Java's keytool to generate a key pair (public and private keys) and a self-signed certificate. This certificate is used by the server to establish secure communication.

bash

```
keytool -genkeypair -alias myserverkey -keyalg RSA -keysize 2048 -validity 365 -storetype PKCS12 -keystore keystore.p12 -storepass password
```

- **Spring Boot Configuration:** Configure the application to use the generated keys and certificates for HTTPS.

yaml

server:

ssl:

key-store: classpath:keystore.p12

key-store-password: password

key-store-type: PKCS12

key-alias: myserverkey

4.3.2 Secure Communication Flow

1. **Client Request:** The client initiates a secure connection to the server using HTTPS by sending a request to `https://yourdomain.com`.
2. **Certificate Exchange:** The server responds by sending its SSL/TLS certificate to the client. This certificate contains the server's public key.
3. **Verification:** The client verifies the certificate against trusted CAs. If the certificate is valid and trusted, the client continues with the connection.
4. **Session Establishment:** A secure session is established using asymmetric encryption (public/private keys). The session is then used to securely exchange data.
5. **Data Transmission:** All subsequent data exchange between the client and server is encrypted using symmetric encryption keys derived from the initial exchange, ensuring confidentiality and integrity.

5. Conclusion

The Spring Boot application implements robust authentication and authorization mechanisms to protect sensitive resources. JWT-based authentication ensures secure and stateless user sessions, while role-based access control enforces fine-grained permissions. Additionally, the use of PKI ensures that communication between the client and server is encrypted and secure through HTTPS. Together, these mechanisms provide a comprehensive security framework for our application.

6. References

- [Spring Security Documentation](#)
- JSON Web Tokens (JWT) Official Documentation
- [Public Key Infrastructure \(PKI\) Overview](#)
- [HTTPS and SSL/TLS Configuration in Spring Boot](#)