Spring Security

https://medium.com/@rasheed99/introduction-on-spring-security-architecture-eb5d7de75a4f

CSRF

CSRF (Cross-Site Request Forgery) protection is primarily designed to prevent malicious attackers from exploiting the trust that a site has in a user's browser. It achieves this by ensuring that requests originated from a different origin (e.g., a malicious site) cannot perform certain actions on behalf of an authenticated user without their knowledge or consent.

The reason CSRF protection typically focuses on preventing unsafe HTTP methods like POST and PUT is because these methods are often used to perform actions that modify data on the server. By tricking a user into submitting a forged request (e.g., via a maliciously crafted HTML link or form) that targets the vulnerable HTTP methods, an attacker could potentially execute unauthorized actions on behalf of the victim.

Here's why CSRF protection is less commonly applied to other HTTP methods:

1. **GET Requests**: CSRF attacks via GET requests are less common because GET requests are typically used for read-only operations and should not result in state changes on the server. However, it's still possible for attackers to abuse GET requests in certain scenarios, such as when they contain parameters that affect server state or trigger actions.

2. **DELETE Requests**: Similar to POST and PUT requests, DELETE requests are used to delete resources on the server. Without CSRF protection, an attacker could forge a DELETE request targeting a vulnerable endpoint and cause unintended data loss or damage.

3. **Safe Methods**: HTTP defines certain methods as "safe," meaning they should not have side effects on the server. These methods include GET, HEAD, and OPTIONS. Safe methods are generally considered less vulnerable to CSRF attacks because they are intended for informational or metadata retrieval purposes only.

4. **Idempotent Methods**: Idempotent methods are HTTP methods that produce the same result regardless of how many times they are called with the same input parameters. PUT and DELETE are idempotent methods, while POST is not. This property makes PUT and DELETE requests more suitable for CSRF protection because they can be safely retried without causing unintended side effects.

While CSRF protection is commonly applied to POST and PUT requests, it's

important to assess the security requirements of your application and consider whether additional protection measures, such as CSRF tokens or same-origin policy enforcement, are needed for other HTTP methods based on the specific risks and functionalities of your application.

Basically for better hold on the security filter chain we override by creating its bean in configuration file.
If we only give http.build() inside the bean then we can access anything, we add authorizeHttpRequests then..,

If we want to pop up a window to take user name and password for any request path then we have to enable basic auth
http.httpBasic(Customiser.withDefaults());

Spring 3.0 way:

WebSecurityConfigurerAdapter has been deprecated which provided us with overloaded methods.

Now we have to do by making beans

To define the authentication related stuff we have to define the bean of UserDetailService
Make a configuration class with @EnableWebSecurity annotation and @Configuration

In the bean we can define user as:

```
UserDetails admin =
User.withUserName(“Basant”).password(“pwd1”).roles(“admin”).build();
```

To use encryption for the password, we can make a bean of PasswordEncoder

```
public PasswordEncoder passwordEncoder() {
	return new BycrptPasswordEncoder();
}
```

We can use this bean in UserDetails bean method to encode the password

```
UserDetails admin =
User.withUserName(“Basant”).password(encoder.encode(“pwd1”)).roles(“admi
n”).build();
```

We can return from the bean new InMemoryUserDetailsManager(admin, ….

Other users);

This is known as authentication.

Now for authorisation:

```
Public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        return http.csrf.disable().authorizeHttpRequests().requestMatchers("/
product/welcome").permitAll()
                .and().authorizeHttpRequests().requestMatchers("/product/
**").authenticated().and().formLogin().and().build();
}
```

This was the way to authorise the endpoints for all the users.

Now to authorise the access of certain endpoints on the basis of role.

In the controller over the methods we can put

@PreAuthorize("hasAuthority('ROLE_ADMIN')') we have to prefix with ROLE

Now to enable method level authorisation, we have to define over the configuration class which is used to make the security beans
@EnableSecurityMethod()

Now this was in memory, but now we can do it for our stored users in database.

So for UserDetailsService bean we have to return our own UserDetailsService class.

Let say we made UserInfoUserDetailsService class implementing the UserDetailsService interface.

Now we have to implement a method that loads the user by name and return UserDetails

For that we can auto wire UserRepository and pass the found User to UserInfoUserDetails class which implements UserDetails interface.

This is in UserInfoUserDetailsService class

```
public UserDetails loadUserByUsername(String userName) throws
UserNameNotFoundException {
        Optional<UserInfo> userInfo = userRepository.findByName(userName);
```

```
        return userInfo.map(UserInfoUserDetails::new).orElseThrow(()->new
UsernameNotFoundException("User not found with name " + userName));
}

//here we have assumed that User has an attribute String role which contains
comma separated roles.
public class UserInfoUserDetails implements UserDetails {
        private String name;
        private String pwd;
        private List<GrantAuthorities> authorities;

        public UserInfoUserDetails(User userInfo) {
                name = userInfo.getUsername();
                pwd = userInfo.getPassword();
                authorities =
Arrays.stream(userInfo.getRoles().split(",")).map(SingleGrantedAuthrority::new)
.toList();
        }
        // for rest method we can return the name, password, authorities, and
make other return true.
}
```

To save the encrypted password we can before storing encrypt it by injecting the bean in controller or repository.

Now we have UserDetailsService to talk to the database.
So change the in memory to create UserDetailsService with your implementation of UserInfoUserDetailsService.

But now we need AuthenticationProvider to authenticate UserDetailsService and make its object and set to authentication object.

```
@Bean
public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider daoAuth = new DaoAuthenticationProvider();
        daoAuth.setUserDetailsService(userDetailsService());
        daoAuth.setPasswordEncoder(passwordEncoder());
        return daoAuth;
}
```

To show the message without login form that is send as JSON

At the backend, you need to configure your server to handle authentication and

authorization, and to respond with appropriate HTTP status codes or error messages when authentication or authorization failures occur. Here's how it might look like in a Spring Boot application with Spring Security:

1. **Configure Spring Security**:
   Configure Spring Security to handle authentication and authorization in your application. You can define security rules, authentication providers, and access control rules in your security configuration class.

   ```java
   import org.springframework.context.annotation.Configuration;
   import
org.springframework.security.config.annotation.web.configuration.EnableWebS
ecurity;
   import
org.springframework.security.config.annotation.web.configuration.WebSecurity
ConfigurerAdapter;

   @Configuration
   @EnableWebSecurity
   public class SecurityConfig extends WebSecurityConfigurerAdapter {

       @Override
       protected void configure(HttpSecurity http) throws Exception {
           http
               .authorizeRequests()
                   .antMatchers("/public/**").permitAll()
                   .anyRequest().authenticated()
                   .and()
               .formLogin()
                   .loginPage("/login")
                   .permitAll()
                   .and()
               .exceptionHandling()
                   .accessDeniedPage("/access-denied")
                   .and()
               .logout()
                   .permitAll();
       }
   }
   ```

2. **Handle Authentication Errors**:
   Define an endpoint to handle authentication errors and return an appropriate HTTP status code or error message.

   ```java
   ```

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class AuthenticationErrorHandler {

    @ExceptionHandler(AuthenticationException.class)
    @ResponseStatus(HttpStatus.UNAUTHORIZED)
    public String handleAuthenticationException(AuthenticationException ex) {
        return ex.getMessage(); // Return error message
    }
}
```

3. **Handle Authorization Errors**:
   Similarly, define an endpoint to handle authorization errors and return an appropriate HTTP status code or error message.

```java
@RestControllerAdvice
public class AuthorizationErrorHandler {

    @ExceptionHandler(AccessDeniedException.class)
    @ResponseStatus(HttpStatus.FORBIDDEN)
    public String handleAccessDeniedException(AccessDeniedException ex) {
        return ex.getMessage(); // Return error message
    }
}
```

In these examples:
- The `SecurityConfig` class configures Spring Security to handle authentication and authorization rules.
- The `AuthenticationErrorHandler` and `AuthorizationErrorHandler` classes handle exceptions thrown during authentication and authorization, respectively, and return appropriate error messages or status codes.

By configuring Spring Security and implementing custom error handlers, your backend application can effectively handle authentication and authorization failures and respond with meaningful error messages or status codes, which can be consumed by your frontend application to perform appropriate actions such as displaying error pages or messages to the user.