# CSRF Attack in a Spring Boot Service

## Introduction

A Cross-Site Request Forgery (CSRF) attack in a Spring Boot service occurs when a malicious website tricks a user's browser into making unauthorized requests to a different website where the user is authenticated. The attack exploits the trust a web application has in a user's browser.

## How It Works

1. A user logs into a Spring Boot application (e.g., an online banking app) and their browser stores a session cookie for authentication.  
2. The user visits a malicious website while still logged into the banking app.  
3. The malicious site contains hidden forms or scripts that make unauthorized requests to the banking app, using the user's session cookie.  
4. The banking app, seeing a valid session cookie, processes the request as if it were made by the legitimate user.

## Real-Life Example

Imagine you have a Spring Boot application providing an endpoint to transfer money:

@PostMapping("/transfer")  
public String transferMoney(@RequestParam String toAccount, @RequestParam double amount) {  
 // Code to transfer money  
 return "Transfer successful";  
}

### Scenario

1. \*\*User logs into the app\*\*: Alice logs into her online banking app and has a valid session cookie.  
2. \*\*Malicious attack\*\*: Alice visits a malicious website controlled by an attacker. The site contains the following HTML:

<form action="https://banking-app.com/transfer" method="POST">  
 <input type="hidden" name="toAccount" value="attacker\_account">  
 <input type="hidden" name="amount" value="1000">  
</form>  
<script>  
 document.forms[0].submit();  
</script>

3. \*\*Browser sends request\*\*: When Alice visits the malicious site, her browser automatically submits the form to the banking app because it includes her session cookie.  
4. \*\*Money is transferred\*\*: The banking app processes the request, assuming it’s legitimate, and transfers $1000 to the attacker's account.

## Prevention in Spring Boot

To prevent CSRF attacks, you can enable and configure CSRF protection in your Spring Security configuration. By default, Spring Security includes CSRF protection for state-changing requests (e.g., POST, PUT, DELETE).

Here’s an example:  
  
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .csrf() // Enable CSRF protection  
 .and()  
 .authorizeRequests()  
 .antMatchers("/public/\*\*").permitAll()  
 .anyRequest().authenticated();  
 }  
}

## How CSRF Tokens Work

Spring Security uses CSRF tokens to ensure that state-changing requests are intentional:  
1. The server includes a unique CSRF token in the HTML forms it serves to users.  
2. When the user submits a form, the token is sent back to the server.  
3. The server validates the token to ensure the request originated from the legitimate source.

For example, in a Thymeleaf template, you can include the CSRF token like this:

<form action="/transfer" method="POST">  
 <input type="hidden" name="\_csrf" th:value="${\_csrf.token}" />  
 <input type="text" name="toAccount" placeholder="Recipient account" />  
 <input type="text" name="amount" placeholder="Amount" />  
 <button type="submit">Transfer</button>  
</form>

If a request lacks a valid CSRF token, Spring Security rejects it, thereby mitigating CSRF attacks.