

1. Create a Spring Boot Web Application

Start by creating a Spring Boot application with the required dependencies for **Spring Security** and **JUnit** for testing.

Add Dependencies in `pom.xml`

```
<dependencies>
  <!-- Spring Boot Web Dependency -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Security Dependency -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <!-- Spring Boot Starter Test for JUnit -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <!-- CSRF and XSS dependencies (they are handled by Spring Security,
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
  </dependency>
</dependencies>
```

2. Enable CSRF, XSS Protection, and Basic Authentication

In `SecurityConfig.java`, configure **CSRF protection**, **XSS prevention**, and **HTTP Basic Authentication**.

```

package com.example.security.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().enable() // Enable CSRF protection
            .and()
            .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/*").permitAll() // Allow GET
            .antMatchers(HttpMethod.POST, "/*").authenticated() // POST
            .anyRequest().authenticated()
            .and()
            .httpBasic() // Enable HTTP Basic Authentication
            .and()
            .addFilterBefore(new XssFilter(), UsernamePasswordAuthenticationFilter.class);

    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("admin").password(passwordEncoder().encode("admin"))
            .and()
            .withUser("user").password(passwordEncoder().encode("user"));
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

```

```
}
```

```
}
```

3. XSS Filter Implementation

To prevent **Cross-Site Scripting (XSS)**, implement a filter that will sanitize any potential XSS attacks in user input.

```
package com.example.security.config;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebFilter("/*")
public class XssFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        // Sanitize user input to avoid XSS (e.g., removing dangerous HTML tags)
        String sanitizedInput = httpRequest.getParameterMap().toString();

        httpRequest.setAttribute("sanitizedInput", sanitizedInput);

        // Continue with the filter chain
        chain.doFilter(request, response);
    }
}
```

```
    @Override
    public void destroy() {
    }
}
```

4. Sample Controller with CSRF and XSS Vulnerability

```
package com.example.security.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SecurityController {

    // CSRF Vulnerable endpoint (by default Spring Boot handles CSRF)
    @PostMapping("/submit")
    public String submitForm(@RequestParam String userInput) {
        return "Received: " + userInput;
    }

    // Sample endpoint for GET request (safe, open access)
    @GetMapping("/")
    public String index() {
        return "Welcome to the secure application!";
    }
}
```

5. JUnit Test for CSRF and XSS

Create JUnit tests to check if **CSRF** protection and **XSS** vulnerability are handled.

```
package com.example.security;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;
```

```

import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import com.example.security.controller.SecurityController;
import static org.springframework.test.web.servlet.request.MockMvcRequest
import static org.springframework.test.web.servlet.result.MockMvcResultMa

@SpringBootTest
public class SecurityTests {

    private MockMvc mockMvc;

    @Test
    public void testCsrfProtection() throws Exception {
        mockMvc = MockMvcBuilders.standaloneSetup(new SecurityController()

        mockMvc.perform(post("/submit").param("userInput", "<script>alert
            .andExpect(status().isForbidden()); // CSRF should block this
    }

    @Test
    public void testXssPrevention() throws Exception {
        mockMvc = MockMvcBuilders.standaloneSetup(new SecurityController()

        mockMvc.perform(post("/submit").param("userInput", "<script>alert
            .andExpect(status().isOk())
            .andExpect(content().string("Received: &lt;script&gt;alert('>
    }
}

```

6. HTTP Basic Authentication Test

```

package com.example.security;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.security.test.context.support.WithMockUser;
import static org.springframework.test.web.servlet.request.MockMvcRequest
import static org.springframework.test.web.servlet.result.MockMvcResultMa

@SpringBootTest
public class BasicAuthTests {

    @Test

```

```

    @WithMockUser(username = "admin", password = "admin")
    public void testBasicAuth() throws Exception {
        mockMvc.perform(get("/"))
            .andExpect(status().isOk())
            .andExpect(content().string("Welcome to the secure applicatio
    })

    @Test
    public void testUnauthorizedAccess() throws Exception {
        mockMvc.perform(get("/"))
            .andExpect(status().isUnauthorized()); // Should be blocked
    }
}

```

7. Enable CSRF Token in HTML Forms

To complete the CSRF configuration, ensure that your HTML forms include the CSRF token for all **POST** requests:

```

<form action="/submit" method="POST">
    <input type="text" name="userInput" />
    <input type="submit" value="Submit" />
</form>

```

Spring Boot automatically includes the CSRF token in the form as long as you include `csrf()` in your `SecurityConfig.java`.

8. Test the Application

- **Run the application** using `mvn spring-boot:run`.
- **Perform security testing** by submitting **malicious input** for **XSS** and **CSRF** via the JUnit tests and observe the behavior of the system.