(/)

Una guía para la protección CSRF en Spring Security

Última modificación: 6 de noviembre de 2018

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

Seguridad de primavera (https://www.baeldung.com/category/spring/spring-security/)

1. Información general

En este tutorial, discutiremos los ataques CSRF de falsificación de solicitudes entre sitios y cómo evitar que usen Spring Security.

Otras lecturas:

Protección CSRF con Spring MVC y Thymeleaf (https://www.baeldung.com/csrf-thymeleaf-with-spring-security)

Guía rápida y práctica para prevenir ataques CSRF con Spring Security, Spring MVC y Thymeleaf.

Leer más (https://www.baeldung.com/csrf-thymeleaf-with-spring-security) →

Configuración automática de Spring Boot Security (https://www.baeldung.com/spring-boot-security-autoconfiguration)

Una guía rápida y práctica para la configuración predeterminada de Spring Security de Spring Boot.

Leer más (https://www.baeldung.com/spring-boot-security-autoconfiguration) →

Introducción a la seguridad del método Spring (https://www.baeldung.com/spring-security-method-security)

Una guía de seguridad a nivel de método utilizando el marco de Spring Security.

Leer más (https://www.baeldung.com/spring-security-method-security) →

2. Dos ataques simples de CSRF

Existen múltiples formas de ataques CSRF: analicemos algunos de los más comunes.

2.1. OBTENER Ejemplos

Consideremos la siguiente solicitud *GET* utilizada por los usuarios registrados para transferir dinero a la cuenta bancaria específica *"1234"*:

```
GET http://bank.com/transfer?accountNo=1234&amount=100
```

Si el atacante quiere transferir dinero de la cuenta de una víctima a su propia cuenta, *"5678"*, debe hacer que la víctima active la solicitud:

```
GET http://bank.com/transfer?accountNo=5678&amount=1000
```

Hay varias formas de hacer que eso suceda:

- **Enlace:** el atacante puede convencer a la víctima de hacer clic en este enlace, por ejemplo, para ejecutar la transferencia:

```
<a href="http://bank.com/transfer?accountNo=5678&amount=1000 (http://bank.c
Show Kittens Pictures
</a>
```

- **Imagen:** el atacante puede usar una *etiqueta <img />* con la URL de destino como fuente de la imagen, por lo que el clic ni siquiera es necesario. La solicitud se ejecutará automáticamente cuando se cargue la página:

```
<img src="http://bank.com/transfer?accountNo=5678&amount=1000 (http://bank.
```

2.2. Ejemplo POST

Si la solicitud principal debe ser una solicitud POST, por ejemplo:

```
POST http://bank.com/transfer
accountNo=1234&amount=100
```

Entonces el atacante necesita que la víctima ejecute un similar:

```
POST http://bank.com/transfer
accountNo=5678&amount=1000
```

Ni *<a>* ni *<img />* funcionarán en este caso. El atacante necesitará un *<formulario>*, de la siguiente manera:

```
<form action="http://bank.com/transfer (http://bank.com/transfer)" method="
    <input type="hidden" name="accountNo" value="5678"/>
    <input type="hidden" name="amount" value="1000"/>
    <input type="submit" value="Show Kittens Pictures"/>
</form>
```

Sin embargo, el formulario se puede enviar automáticamente usando Javascript, de la siguiente manera:

```
<body onload="document.forms[0].submit()">
<form>
...
```

2.3. Simulación práctica

Ahora que entendemos cómo se ve un ataque CSRF, simulemos estos ejemplos dentro de una aplicación Spring.

Vamos a comenzar con una implementación de controlador simple : el *BankController* :

```
@Controller
public class BankController {
    private Logger logger = LoggerFactory.getLogger(getClass());

    @RequestMapping(value = "/transfer", method = RequestMethod.GET)
    @ResponseBody
    public String transfer(@RequestParam("accountNo") int accountNo,
      @RequestParam("amount") final int amount) {
        logger.info("Transfer to {}", accountNo);
        ...
    }

    @RequestMapping(value = "/transfer", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.OK)
    public void transfer2(@RequestParam("accountNo") int accountNo,
      @RequestParam("amount") final int amount) {
        logger.info("Transfer to {}", accountNo);
        ...
    }
}
```

Y también tengamos una página HTML básica que active la operación de transferencia bancaria:

```
<html>
<body>
    <h1>CSRF test on Origin</h1>
    <a href="transfer?accountNo=1234&amount=100">Transfer Money to John</a

    <form action="transfer" method="POST">
        <label>Account Number</label>
        <input name="accountNo" type="number"/>

        <label>Amount</label>
        <input name="amount" type="number"/>

        <input type="submit">
    </form>
</body>
</html>
```

Esta es la página de la aplicación principal, que se ejecuta en el dominio de origen.

Tenga en cuenta que hemos simulado tanto un *GET a* través de un enlace simple como un *POST a* través de un *<form>* simple .

Ahora, veamos cómo se vería **la página del atacante** :

```
<html>
<body>
    <a href="http://localhost:8080/transfer?accountNo=5678&amount=1000 (ht

    <img src="http://localhost:8080/transfer?accountNo=5678&amount=1000 (h

    <form action="http://localhost:8080/transfer (http://localhost:8080/tr
        <input name="accountNo" type="hidden" value="5678"/>
        <input name="amount" type="hidden" value="1000"/>
        <input type="submit" value="Show Kittens Picture">
    </form>
</body>
</html>
```

Esta página se ejecutará en un dominio diferente: el dominio del atacante.

Finalmente, ejecutemos las dos aplicaciones, la aplicación original y la atacante, localmente, y accedamos primero a la página original:

```
http://localhost:8081/spring-rest-full/csrfHome.html
```

Entonces, accedamos a la página del atacante:

```
http://localhost:8081/spring-security-rest/api/csrfAttacker.html
```

Al realizar un seguimiento de las solicitudes exactas que se originan en esta página del atacante, podremos detectar de inmediato la solicitud problemática, golpeando la aplicación original y autenticada por completo.

3. Configuración de seguridad de primavera

Para usar la protección CSRF de Spring Security, primero debemos asegurarnos de usar los métodos HTTP adecuados para cualquier cosa que modifique el estado (*PATCH* , *POST* , *PUT* y *DELETE,* no GET).

3.1. Configuracion Java

La protección CSRF está **habilitada de forma predeterminada** en la configuración de Java. Aún podemos desactivarlo si necesitamos:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
      .csrf().disable();
}
```

3.2. Configuración XML

En la configuración XML anterior (anterior a Spring Security 4), la protección CSRF estaba deshabilitada de manera predeterminada y podríamos habilitarla de la siguiente manera:

```
<http>
    ...
    <csrf />
</http>
```

A partir de **Spring Security 4.x** : la protección CSRF también está habilitada de manera predeterminada en la configuración XML; por supuesto, aún podemos desactivarlo si necesitamos:

```
<http>
    ...
    <csrf disabled="true"/>
</http>
```

3.3. Parámetros de forma extra

Finalmente, con la protección CSRF habilitada en el lado del servidor, también tendremos que incluir el token CSRF en nuestras solicitudes en el lado del cliente:

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

3.4. Usando JSON

No podemos enviar el token CSRF como parámetro si estamos usando JSON; en su lugar, podemos enviar el token dentro del encabezado.

Primero tendremos que incluir el token en nuestra página, y para eso podemos usar metaetiquetas:

```
<meta name="_csrf" content="${_csrf.token}"/>
<meta name="_csrf_header" content="${_csrf.headerName}"/>
```

Luego construiremos el encabezado:


```
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");

$(document).ajaxSend(function(e, xhr, options) {
    xhr.setRequestHeader(header, token);
});
```

4. Prueba CSRF deshabilitada

Con todo eso en su lugar, pasaremos a hacer algunas pruebas.

Primero intentemos enviar una solicitud POST simple cuando CSRF esté deshabilitado:

```
@ContextConfiguration(classes = { SecurityWithoutCsrfConfig.class, ...})
public class CsrfDisabledIntegrationTest extends CsrfAbstractIntegrationTe

    @Test
    public void givenNotAuth_whenAddFoo_thenUnauthorized() throws Exceptio
        mvc.perform(
          post("/foos").contentType(MediaType.APPLICATION_JSON)
                       .content(createFoo())
          ).andExpect(status().isUnauthorized());
    }

    @Test
    public void givenAuth_whenAddFoo_thenCreated() throws Exception {
        mvc.perform(
          post("/foos").contentType(MediaType.APPLICATION_JSON)
                       .content(createFoo())
                       .with(testUser())
        ).andExpect(status().isCreated());
    }
}
```

Como habrás notado, estamos usando una clase base para mantener la lógica auxiliar de prueba común : *CsrfAbstractIntegrationTest* :

```java
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
public class CsrfAbstractIntegrationTest {
    @Autowired
    private WebApplicationContext context;

    @Autowired
    private Filter springSecurityFilterChain;

    protected MockMvc mvc;

    @Before
    public void setup() {
        mvc = MockMvcBuilders.webAppContextSetup(context)
                             .addFilters(springSecurityFilterChain)
                             .build();
    }

    protected RequestPostProcessor testUser() {
        return user("user").password("userPass").roles("USER");
    }

    protected String createFoo() throws JsonProcessingException {
        return new ObjectMapper().writeValueAsString(new Foo(randomAlphabe
    }
}
```

Tenga en cuenta que, cuando el usuario tenía las credenciales de seguridad correctas, la solicitud se ejecutó con éxito, no se requirió información adicional.

Eso significa que el atacante puede simplemente usar cualquiera de los vectores de ataque discutidos anteriormente para comprometer fácilmente el sistema.

5. Prueba habilitada CSRF

Ahora, habilitemos la protección CSRF y veamos la diferencia:

```
@ContextConfiguration(classes = { SecurityWithCsrfConfig.class, ...})
public class CsrfEnabledIntegrationTest extends CsrfAbstractIntegrationTes

    @Test
    public void givenNoCsrf_whenAddFoo_thenForbidden() throws Exception {
        mvc.perform(
          post("/foos").contentType(MediaType.APPLICATION_JSON)
                       .content(createFoo())
                       .with(testUser())
          ).andExpect(status().isForbidden());
    }

    @Test
    public void givenCsrf_whenAddFoo_thenCreated() throws Exception {
        mvc.perform(
          post("/foos").contentType(MediaType.APPLICATION_JSON)
                       .content(createFoo())
                       .with(testUser()).with(csrf())
         ).andExpect(status().isCreated());
    }
}
```

Ahora, cómo esta prueba está utilizando una configuración de seguridad diferente, una que tiene habilitada la protección CSRF.

Ahora, la solicitud POST simplemente fallará si el token CSRF no está incluido, lo que por supuesto significa que los ataques anteriores ya no son una opción.

Finalmente, observe el método *csrf () (https://docs.spring.io/spring-security/site/docs/4.0.2.RELEASE/apidocs/org/springframework/security/test/web/servlet/request/SecurityMockMvcRequestPostProcessors.html#csrf())* en la prueba; esto crea un *RequestPostProcessor* que completará automáticamente un token CSRF válido en la solicitud con fines de prueba.

6. Conclusión

En este artículo, discutimos un par de ataques CSRF y cómo evitar que usen Spring Security.

La **implementación completa** de este tutorial se puede encontrar en el proyecto GitHub (https://github.com/eugenp/tutorials/tree/master/spring-security-mvc-custom) : este es un proyecto basado en Maven, por lo que debería ser fácil de importar y ejecutar tal como está.

Huésped

Sanket Mishra

Muchas gracias ... Esto fue realmente muy útil

+ 3 −

hace 3 años

¡Los comentarios están cerrados en este artículo!

CATEGORÍAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)