19. Falsificación de solicitud de sitios cruzados (CSRF)

Esta sección discute el soporte de falsificación de solicitudes de sitios cruzados (CSRF) de Spring Security .

19.1 Ataques CSRF

Antes de analizar cómo Spring Security puede proteger las aplicaciones de los ataques CSRF, explicaremos qué es un ataque CSRF. Echemos un vistazo a un ejemplo concreto para obtener una mejor comprensión.

Suponga que el sitio web de su banco proporciona un formulario que permite transferir dinero del usuario actualmente conectado a otra cuenta bancaria. Por ejemplo, la solicitud HTTP podría verse así:

```
POST / transferencia HTTP / 1.1
Anfitrión: bank.example.com
Cookie: JSESSIONID = randomid; Dominio = banco.ejemplo.com; Seguro; HttpOnly
Tipo de contenido: application / x-www-form-urlencoded

cantidad = 100.00 y enrutamiento Número = 1234 y cuenta = 9876
```

Ahora simule que se autentica en el sitio web de su banco y luego, sin cerrar sesión, visite un sitio web malvado. El malvado sitio web contiene una página HTML con la siguiente forma:

```
<form  action = "https://bank.example.com/transfer"  method = "post" >
<input  type = "hidden"
        name = "amount"
        value = "100.00" />
<input  type = "hidden"
        name = "routingNumber"
        value = "evilsRoutingNumber" />
<input  type = "hidden"
        name = "account"
        value = "evilsAccountNumber" />
<input  type = "submit"
        value= "¡Gana dinero!" />
</form>
```

Te gusta ganar dinero, así que haces clic en el botón enviar. En el proceso, ha transferido involuntariamente $ 100 a un usuario malintencionado. Esto sucede porque, si bien el sitio web malvado no puede ver sus cookies, las cookies asociadas con su banco aún se envían junto con la solicitud.

Peor aún, todo este proceso podría haberse automatizado utilizando JavaScript. Esto significa que ni siquiera necesitaba hacer clic en el botón. Entonces, ¿cómo nos protegemos de tales ataques?

19.2 Patrón de token de sincronizador

El problema es que la solicitud HTTP del sitio web del banco y la solicitud del sitio web malvado son exactamente las mismas. Esto significa que no hay forma de rechazar las solicitudes procedentes del sitio web maligno y permitir las solicitudes procedentes del sitio web del banco. Para protegernos de los ataques CSRF, debemos asegurarnos de que haya algo en la solicitud que el sitio maligno no pueda proporcionar.

Una solución es usar el patrón de token de sincronizador . Esta solución es para garantizar que cada solicitud requiere, además de nuestra cookie de sesión, un token generado aleatoriamente como parámetro HTTP. Cuando se envía una solicitud, el servidor debe buscar el valor esperado para el parámetro y compararlo con el valor real en la solicitud. Si los valores no coinciden, la solicitud debería fallar.

Podemos relajar las expectativas de que solo se requiera el token para cada solicitud HTTP que actualice el estado. Esto se puede hacer de manera segura ya que la misma política de origen garantiza que el sitio maligno no pueda leer la respuesta. Además, no queremos incluir el token aleatorio en HTTP GET, ya que esto puede hacer que los tokens se filtren.

Echemos un vistazo a cómo cambiaría nuestro ejemplo. Suponga que el token generado aleatoriamente está presente en un parámetro HTTP llamado _csrf. Por ejemplo, la solicitud de transferencia de dinero se vería así:

```
POST / transferencia HTTP / 1.1
Anfitrión: bank.example.com
Cookie: JSESSIONID = randomid; Dominio = banco.ejemplo.com; Seguro; HttpOnly
Tipo de contenido: application / x-www-form-urlencoded

cantidad = 100.00 y número de ruta = 1234 y cuenta = 9876 & _csrf = <seguro-seg
```

Notará que agregamos el parámetro _csrf con un valor aleatorio. Ahora el sitio web malvado no podrá adivinar el valor correcto para el parámetro _csrf (que debe proporcionarse explícitamente en el sitio web malvado) y la transferencia fallará cuando el servidor compare el token real con el token esperado.

19.3 Cuándo usar la protección CSRF

¿Cuándo debe usar la protección CSRF? Nuestra recomendación es utilizar la protección CSRF para cualquier solicitud que pueda ser procesada por un navegador por usuarios normales. Si solo está creando un servicio que es utilizado por clientes que no son navegadores, es probable que desee deshabilitar la protección CSRF.

19.3.1 Protección CSRF y JSON

Una pregunta común es "¿necesito proteger las solicitudes JSON hechas por javascript?" La respuesta corta es, depende. Sin embargo, debe tener mucho cuidado ya que hay vulnerabilidades CSRF que pueden afectar las solicitudes JSON. Por ejemplo, un usuario malintencionado puede crear un CSRF con JSON utilizando el siguiente formulario :

```
<form  action = "https://bank.example.com/transfer"  method = "post"  enctype =
<input  name = '{"cantidad": 100, "routingNumber": "evilsRoutingNumber", "accou
<input  type = "submit"
        value = "¡Gane dinero!" />
</form>
```

Esto producirá la siguiente estructura JSON

```
{ "cantidad" : 100 ,
 "número de ruta" : "evilsRoutingNumber" ,
 "cuenta" : "evilsAccountNumber" ,
 "ignore_me" : "= prueba"
}
```

Si una aplicación no validara el tipo de contenido, estaría expuesta a esta vulnerabilidad. Dependiendo de la configuración, una aplicación Spring MVC que valida el tipo de contenido aún podría explotarse actualizando el sufijo URL para terminar con ".json" como se muestra a continuación:

```
<form  action = "https://bank.example.com/transfer.json"  method = "post"  enc
<input  name = '{"cantidad": 100, "routingNumber": "evilsRoutingNumber "," acc
<input  type = " submit "
        value = " ¡Gane dinero! " />
</form>
```

19.3.2 CSRF y aplicaciones de navegador sin estado

¿Qué pasa si mi solicitud no tiene estado? Eso no significa necesariamente que estés protegido. De hecho, si un usuario no necesita realizar ninguna acción en el navegador web para una solicitud determinada, es probable que aún sea vulnerable a los ataques CSRF.

Por ejemplo, considere que una aplicación usa una cookie personalizada que contiene todo el estado para la autenticación en lugar del JSESSIONID. Cuando se realiza el ataque CSRF, la cookie personalizada se enviará con la solicitud de la misma manera que la cookie JSESSIONID se envió en nuestro ejemplo anterior.

Los usuarios que utilizan la autenticación básica también son vulnerables a los ataques CSRF, ya que el navegador incluirá automáticamente la contraseña del nombre de usuario en cualquier solicitud de la misma manera que se envió la cookie JSESSIONID en nuestro ejemplo anterior.

19.4 Uso de Spring Security CSRF Protection

¿Cuáles son los pasos necesarios para usar Spring Security para proteger nuestro sitio contra los ataques CSRF? Los pasos para usar la protección CSRF de Spring Security se detallan a continuación:

- Use verbos HTTP adecuados
- Configurar la protección CSRF
- Incluir el token CSRF

19.4.1 Usar verbos HTTP adecuados

El primer paso para protegerse contra los ataques CSRF es asegurarse de que su sitio web use los verbos HTTP adecuados. Específicamente, antes de que el soporte CSRF de Spring Security pueda ser útil, debe asegurarse de que su aplicación esté usando PATCH, POST, PUT y / o DELETE para cualquier cosa que modifique el estado.

Esto no es una limitación del soporte de Spring Security, sino un requisito general para la prevención adecuada de CSRF. La razón es que incluir información privada en un HTTP GET puede hacer que la información se filtre. Consulte RFC 2616 Sección 15.1.3 Codificación de información confidencial en URI para obtener orientación general sobre el uso de POST en lugar de GET para obtener información confidencial.

19.4.2 Configurar la protección CSRF

El siguiente paso es incluir la protección CSRF de Spring Security dentro de su aplicación. Algunos marcos manejan tokens CSRF inválidos al invalidar la sesión del usuario, pero esto causa sus propios problemas . Por el contrario, la protección CSRF de Spring Security

producirá un acceso HTTP 403 denegado. Esto se puede personalizar configurando AccessDeniedHandler para procesar de manera `InvalidCsrfTokenException` diferente.

A partir de Spring Security 4.0, la protección CSRF está habilitada de forma predeterminada con la configuración XML. Si desea deshabilitar la protección CSRF, la configuración XML correspondiente se puede ver a continuación.

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

La protección CSRF está habilitada de forma predeterminada con la configuración de Java. Si desea deshabilitar CSRF, la configuración de Java correspondiente se puede ver a continuación. Consulte el Javadoc de csrf () para obtener personalizaciones adicionales sobre cómo se configura la protección CSRF.

```
@EnableWebSecurity
 public  class WebSecurityConfig extiende
WebSecurityConfigurerAdapter {

        @ Override
         protegida  vacío configure (HttpSecurity http) lanza la excepción {
                http
                        .csrf (). disable ();
        }
}
```

19.4.3 Incluir el token CSRF

Envíos de formularios

El último paso es asegurarse de incluir el token CSRF en todos los métodos PATCH, POST, PUT y DELETE. Una forma de abordar esto es utilizar el `_csrf` atributo de solicitud para obtener el actual `CsrfToken`. A continuación se muestra un ejemplo de cómo hacer esto con un JSP:

```
<c: url  var = "logoutUrl"  value = "/ logout" />
<form  action = "$ {logoutUrl}"
        method = "post" >
<input  type = "submit"
        value = "Logout" />
<input  type = "hidden"
        name = "$ {_ csrf.parameterName}"
```

```
        value = "$ {_ csrf.token}" />
</form>
```

Un enfoque más fácil es usar la etiqueta csrfInput de la biblioteca de etiquetas Spring Security JSP.

Si está usando la `<form:form>` etiqueta Spring MVC o Thymeleaf 2.1+ y está usando `@EnableWebSecurity`, `CsrfToken` se incluye automáticamente para usted (usando el `CsrfRequestDataValueProcessor`).

Solicitudes Ajax y JSON

Si está utilizando JSON, entonces no es posible enviar el token CSRF dentro de un parámetro HTTP. En su lugar, puede enviar el token dentro de un encabezado HTTP. Un patrón típico sería incluir el token CSRF dentro de sus metaetiquetas. A continuación se muestra un ejemplo con un JSP:

```
<html>
<head>
        <meta  name = "_csrf"  content = "$ {_ csrf.token}" />
        <! - el nombre del encabezado predeterminado es X-CSRF-TOKEN ->
        <meta  name = "_csrf_header"  content = "$ {_ csrf.headerName}" />
        <! - ... ->
</head>
<! - ... ->
```

En lugar de crear manualmente las metaetiquetas, puede usar la etiqueta csrfMetaTags más simple de la biblioteca de etiquetas Spring Security JSP.

Luego puede incluir el token en todas sus solicitudes de Ajax. Si estaba usando jQuery, esto podría hacerse con lo siguiente:

```
$ ( function () {
 var token = $ ( "meta [name = '_ csrf']" ) .attr ( "content" );
 var header = $ ( "meta [name = '_ csrf_header']" ) .attr ( " contenido " );
$ (documento) .ajaxSend ( función (e, xhr, opciones) {
        xhr.setRequestHeader (encabezado, token);
});
});
```

Como alternativa a jQuery, recomendamos usar cujoJS's rest.js. El módulo rest.js proporciona soporte avanzado para trabajar con solicitudes y respuestas HTTP de manera RESTful. Una capacidad central es la capacidad de contextualizar el comportamiento de

adición del cliente HTTP según sea necesario mediante el encadenamiento de interceptores en el cliente.

```
cliente var = rest.chain (csrf, {
token: $ ( "meta [name = '_ csrf']" ) .attr ( "contenido" ),
nombre: $ ( "meta [nombre = '_ csrf_header']" ) .attr ( "contenido" )
});
```

El cliente configurado se puede compartir con cualquier componente de la aplicación que necesite realizar una solicitud al recurso protegido CSRF. Una diferencia significativa entre rest.js y jQuery es que solo las solicitudes realizadas con el cliente configurado contendrán el token CSRF, frente a jQuery, donde *todas las* solicitudes incluirán el token. La capacidad de determinar qué solicitudes reciben el token ayuda a evitar que se filtre el token CSRF a un tercero. Consulte la documentación de referencia de rest.js para obtener más información sobre rest.js.

CookieCsrfTokenRepository

Puede haber casos en los que los usuarios deseen conservar la `CsrfToken` cookie. Por defecto `CookieCsrfTokenRepository`, escribirá en una cookie llamada `XSRF-TOKEN` y la leerá desde un encabezado llamado `X-XSRF-TOKEN` o el parámetro HTTP `_csrf`. Estos valores predeterminados provienen de AngularJS

Puede configurar `CookieCsrfTokenRepository` en XML usando lo siguiente:

```
<http>
        <! - ... ->
        <csrf  token-repository-ref = "tokenRepository" />
</http>
<b: bean  id = "tokenRepository"
        class = "org.springframework.security.web. csrf.CookieCsrfTokenReposit
        p: cookieHttpOnly = " false " />
```

La muestra se establece explícitamente `cookieHttpOnly=false`. Esto es necesario para permitir que JavaScript (es decir, AngularJS) lo lea. Si no necesita la capacidad de leer la cookie con JavaScript directamente, se recomienda omitirla `cookieHttpOnly=false` para mejorar la seguridad.

Puede configurar `CookieCsrfTokenRepository` en Configuración Java utilizando:

```
@EnableWebSecurity
 public  class WebSecurityConfig extiende
                WebSecurityConfigurerAdapter {
```

```
        @ Override
         protegida  vacío configure (HttpSecurity http) lanza la excepción {
                http
                        .csrf ()
                                .csrfTokenRepository (CookieCsrfTokenRepository
        }
}
```

La muestra se establece explícitamente `cookieHttpOnly=false`. Esto es necesario para permitir que JavaScript (es decir, AngularJS) lo lea. Si no necesita la capacidad de leer la cookie con JavaScript directamente, se recomienda omitir `cookieHttpOnly=false` (en su `new CookieCsrfTokenRepository()` lugar) para mejorar la seguridad.

19.5 Advertencias CSRF

Hay algunas advertencias al implementar CSRF.

19.5.1 Tiempos de espera

Un problema es que el token CSRF esperado se almacena en la HttpSession, por lo que tan pronto como la HttpSession expire, su configuración `AccessDeniedHandler` recibirá una InvalidCsrfTokenException. Si está utilizando el predeterminado `AccessDeniedHandler`, el navegador obtendrá un HTTP 403 y mostrará un mensaje de error deficiente.

Uno podría preguntarse por qué lo esperado `CsrfToken` no se almacena en una cookie de forma predeterminada. Esto se debe a que existen vulnerabilidades conocidas en las que los encabezados (es decir, especificar las cookies) pueden ser configurados por otro dominio. Esta es la misma razón por la que Ruby on Rails ya no omite las comprobaciones CSRF cuando el encabezado X-Requested-With está presente . Consulte este hilo de webappsec.org para obtener detalles sobre cómo realizar el exploit. Otra desventaja es que al eliminar el estado (es decir, el tiempo de espera) pierde la capacidad de terminar por la fuerza el token si se ve comprometido.

Una manera simple de mitigar a un usuario activo que experimenta un tiempo de espera es tener un JavaScript que le permita al usuario saber que su sesión está por caducar. El usuario puede hacer clic en un botón para continuar y actualizar la sesión.

Alternativamente, especificar una personalizada le `AccessDeniedHandler` permite procesar la `InvalidCsrfTokenException` forma que desee. Para ver un ejemplo de cómo personalizar, `AccessDeniedHandler` consulte los enlaces proporcionados para la configuración de xml y Java .

Finalmente, la aplicación se puede configurar para usar CookieCsrfTokenRepository que no caducará. Como se mencionó anteriormente, esto no es tan seguro como usar una sesión, pero en muchos casos puede ser lo suficientemente bueno.

19.5.2 Iniciar sesión

Para protegerse contra la falsificación de solicitudes de inicio de sesión, el formulario de inicio de sesión también debe protegerse contra ataques CSRF. Como `CsrfToken` se almacena en HttpSession, esto significa que se creará una HttpSession tan pronto como `CsrfToken` se acceda al atributo token. Si bien esto suena mal en una arquitectura RESTful / sin estado, la realidad es que ese estado es necesario para implementar la seguridad práctica. Sin estado, no tenemos nada que podamos hacer si una ficha se ve comprometida. En términos prácticos, el token CSRF es bastante pequeño y debería tener un impacto insignificante en nuestra arquitectura.

Una técnica común para proteger el formulario de inicio de sesión es mediante el uso de una función de JavaScript para obtener un token CSRF válido antes del envío del formulario. Al hacer esto, no hay necesidad de pensar en los tiempos de espera de la sesión (discutidos en la sección anterior) porque la sesión se crea justo antes del envío del formulario (suponiendo que CookieCsrfTokenRepository no esté configurado en su lugar), por lo que el usuario puede permanecer en la página de inicio de sesión y envíe el nombre de usuario / contraseña cuando lo desee. Para lograr esto, puede aprovechar lo `CsrfTokenArgumentResolver` proporcionado por Spring Security y exponer un punto final como se describe aquí .

19.5.3 Cerrar sesión

Agregar CSRF actualizará LogoutFilter para usar solo HTTP POST. Esto garantiza que el cierre de sesión requiera un token CSRF y que un usuario malintencionado no pueda cerrar la sesión por la fuerza a sus usuarios.

Un enfoque es utilizar un formulario para cerrar sesión. Si realmente desea un enlace, puede usar JavaScript para que el enlace realice una POST (es decir, tal vez en un formulario oculto). Para los navegadores con JavaScript deshabilitado, opcionalmente puede hacer que el enlace lleve al usuario a una página de confirmación de cierre de sesión que realizará la POST.

Si realmente desea utilizar HTTP GET con el cierre de sesión, puede hacerlo, pero recuerde que esto generalmente no se recomienda. Por ejemplo, la siguiente configuración de Java realizará el cierre de sesión con la URL / el cierre de sesión se solicita con cualquier método HTTP:

```
@EnableWebSecurity
 public  class WebSecurityConfig extiende
WebSecurityConfigurerAdapter {

        @ Override
         protegida  vacío configure (HttpSecurity http) lanza la excepción {
                http
                        .cerrar sesión()
                                .logoutRequestMatcher ( nuevo AntPathRequestMa
        }
}
```

19.5.4 Multiparte (carga de archivos)

Hay dos opciones para usar la protección CSRF con datos multiparte / formulario. Cada opción tiene sus compensaciones.

- Colocación de MultipartFilter antes de Spring Security
- Incluir token CSRF en acción

Antes de integrar la protección CSRF de Spring Security con la carga de archivos de varias partes, asegúrese de que puede cargar sin la protección CSRF primero. Puede encontrar más información sobre el uso de formularios multiparte con Spring en la sección de soporte multiparte de Spring (carga de archivos) 17.10 de la referencia Spring y el javadoc MultipartFilter .

Colocación de MultipartFilter antes de Spring Security

La primera opción es asegurarse de que `MultipartFilter` se especifique antes del filtro Spring Security. Especificar `MultipartFilter` antes del filtro Spring Security significa que no hay autorización para invocar, lo `MultipartFilter` que significa que cualquiera puede colocar archivos temporales en su servidor. Sin embargo, solo los usuarios autorizados podrán enviar un archivo procesado por su aplicación. En general, este es el enfoque recomendado porque la carga de archivos temporales debería tener un impacto insignificante en la mayoría de los servidores.

Para garantizar que `MultipartFilter` se especifique antes del filtro Spring Security con la configuración de Java, los usuarios pueden anular antes de SpringSecurityFilterChain como se muestra a continuación:

```
 SecurityApplicationInitializer de clase pública extiende AbstractSecurityWebA

        @ Anular
         vacío protegido antes de  SpringSecurityFilterChain (ServletContext s
                insertFilters (servletContext, new MultipartFilter ());
        }
}
```

Para asegurarse de que `MultipartFilter` se especifica antes del filtro Spring Security con configuración XML, los usuarios pueden asegurarse de que el elemento <filter-mapping> del `MultipartFilter` se coloca antes de springSecurityFilterChain dentro del web.xml como se muestra a continuación:

```
<filter>
        <filter-name> MultipartFilter </filter-name>
        <filter-class> org.springframework.web.multipart.support.MultipartFilt
</filter>
<filter>
        <filter-name> springSecurityFilterChain </filter-name>
        <filter-class> org.springframework.web.filter.DelegatingFilterProxy </
</filter>
<filter-mapping>
        <filter-name> MultipartFilter </filter-name>
        <url- patrón> / * </url-pattern>
</filter-mapping>
<filter-mapping>
        <filter-name>springSecurityFilterChain </filter-name>
        <url-pattern>/ * </url-pattern>
</filter-mapping>
```

Incluir token CSRF en acción

Si permitir que usuarios no autorizados carguen archivos temporales no es aceptable, una alternativa es colocar `MultipartFilter` después del filtro Spring Security e incluir el CSRF como parámetro de consulta en el atributo de acción del formulario. A continuación se muestra un ejemplo con un jsp

```
<form  action = "./upload?${_csrf.parameterName}=${_csrf.token}"  method = "po
```

The disadvantage to this approach is that query parameters can be leaked. More genearlly, it is considered best practice to place sensitive data within the body or headers to ensure it is not leaked. Additional information can be found in RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's.

19.5.5 HiddenHttpMethodFilter

The HiddenHttpMethodFilter should be placed before the Spring Security filter. In general this is true, but it could have additional implications when protecting against CSRF attacks.

Note that the HiddenHttpMethodFilter only overrides the HTTP method on a POST, so this is actually unlikely to cause any real problems. However, it is still best practice to ensure it is placed before Spring Security's filters.

19.6 Overriding Defaults

Spring Security's goal is to provide defaults that protect your users from exploits. This does not mean that you are forced to accept all of its defaults.

For example, you can provide a custom CsrfTokenRepository to override the way in which the `CsrfToken` is stored.

You can also specify a custom RequestMatcher to determine which requests are protected by CSRF (i.e. perhaps you don't care if log out is exploited). In short, if Spring Security's CSRF protection doesn't behave exactly as you want it, you are able to customize the behavior. Refer to the Section 43.1.18, "<csrf>" documentation for details on how to make these customizations with XML and the `CsrfConfigurer` javadoc for details on how to make these customizations when using Java configuration.

Prev	Up	Next
18. Remember-Me Authentication	Home	20. CORS