DGTIC UNAM
DIRECCIÓN GENERAL DE CÓMPUTO Y DE TECNOLOGÍAS DE INFORMACIÓN Y COMUNICACIÓN

REDEC UNAM

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
DIRECCIÓN GENERAL DE CÓMPUTO Y DE TECNOLOGÍAS
DE INFORMACIÓN Y COMUNICACIÓN

DIPLOMADO

Desarrollo de sistemas con tecnología Java

Módulo 11

CORS y CSRF

Mtro. Alfonso Gregorio Rivero Duarte

1. CORS

1.1 Introducción a CORS

¿Qué es CORS?

El intercambio de recursos de origen cruzado (CORS, por sus siglas en inglés), es un mecanismo basado en cabeceras HTTP que permite a un servidor indicar cualquier dominio, esquema o puerto con un origen (en-US) distinto del suyo desde el que un navegador debería permitir la carga de recursos.

CORS también se basa en un mecanismo por el cual los navegadores realizan una solicitud de "verificación previa" al servidor que aloja el recurso de origen cruzado, con el fin de comprobar que el servidor permitirá la solicitud real.

En esa comprobación previa, el navegador envía cabeceras que indican el método HTTP y las cabeceras que se utilizarán en la solicitud real.

¿Qué es CORS?

Un ejemplo de solicitud de origen cruzado: el código JavaScript del front-end servido desde

https://domain-a.com

utiliza XMLHttpRequest para realizar una solicitud a

https://domain-b.com/data.json

Por razones de seguridad, los navegadores restringen las peticiones HTTP de origen cruzado iniciadas desde scripts. Por ejemplo, XMLHttpRequest y la API Fetch siguen la Política Same-origin. Esto significa que una aplicación web que utilice esas API solo puede solicitar recursos del mismo origen desde el que se cargó la aplicación, a menos que la respuesta de otros orígenes incluya las cabeceras CORS adecuadas.

Aclaración!

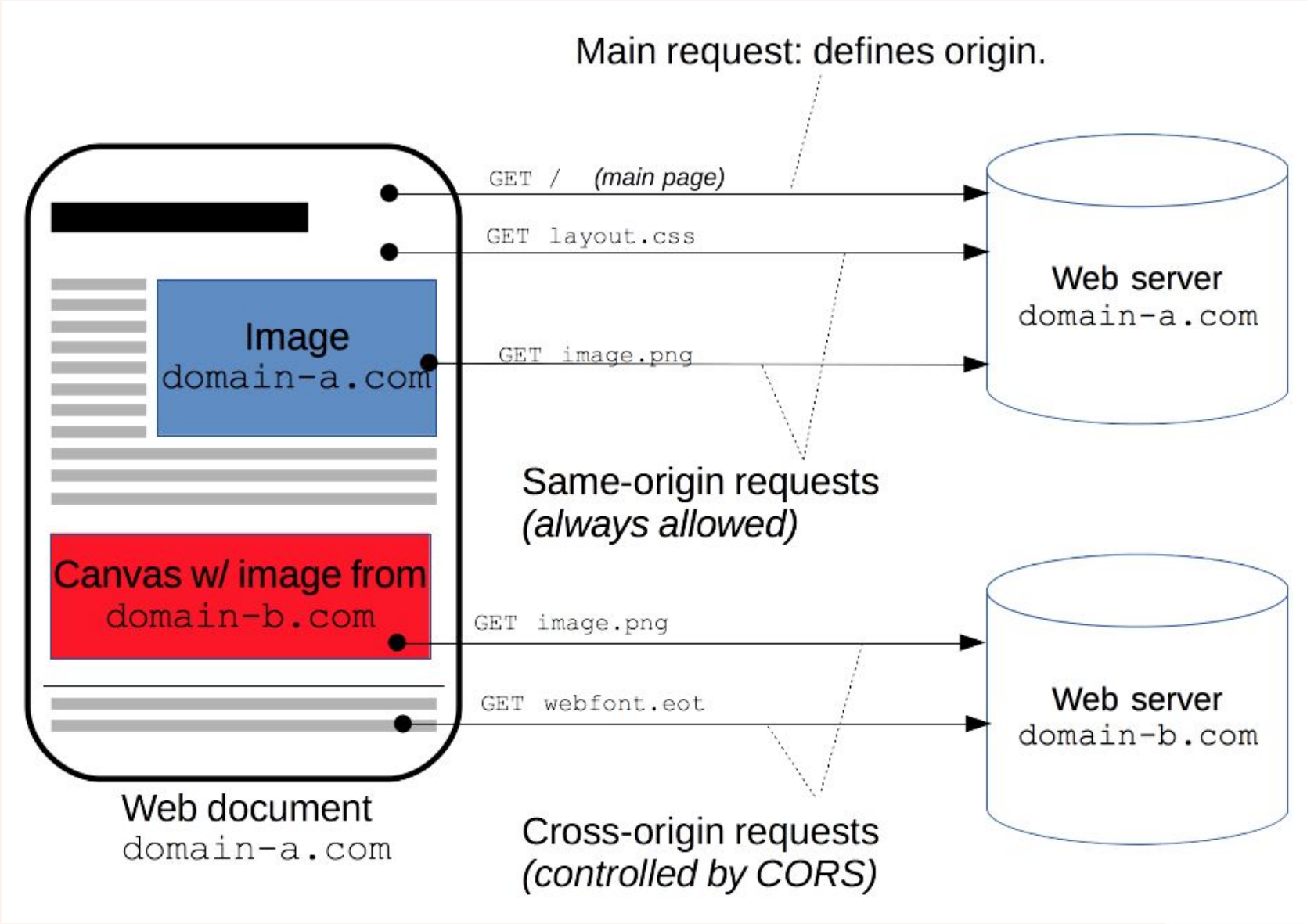
Esto es muy común cuando desarrollamos aplicaciones tipo REST donde tenemos el backend y frontend separados

Por ejemplo

1. tenemos el back desarrollado en spring
2. tenemos el front desarrollado en react / angular / vue
3. comunicamos con una capa intermedia en typescript (BFF)

Aquí, consumimos los endpoints del back dentro de TypeScript y ahí tenemos que configurar CORS del lado del front.

Pero, en el back, igual hay que configurarlo, puesto que lo más seguro es que el front viva en un lugar diferente al back!


Main request: defines origin.
GET / (main page)
GET layout.css
GET image.png
Image
domain-a.com
Web server
domain-a.com
Same-origin requests
(always allowed)
Canvas w/ image from
domain-b.com
GET image.png
GET webfont.eot
Web server
domain-b.com
Web document
domain-a.com
Cross-origin requests
(controlled by CORS)

Cabeceras

Existen diferentes cabeceras o CORS headers y cada una aborda un aspecto distinto. Ya hemos mencionado dos cabeceras importantes para identificar orígenes seguros y métodos permitidos, pero hay más:

Cabeceras

- **Access-Control-Allow-Origin:** ¿qué origen está permitido?
- **Access-Control-Allow-Credentials:** ¿también se aceptan solicitudes cuando el modo de credenciales es incluir (include)?
- **Access-Control-Allow-Headers:** ¿qué cabeceras pueden utilizarse?
- **Access-Control-Allow-Methods:** ¿qué métodos de petición HTTP están permitidos?
- **Access-Control-Expose-Headers:** ¿qué cabeceras pueden mostrarse?
- **Access-Control-Max-Age:** ¿cuándo pierde su validez la solicitud preflight?
- **Access-Control-Request-Headers:** ¿qué header HTTP se indica en la solicitud preflight?
- **Access-Control-Request-Method:** ¿qué método de petición HTTP se indica en la solicitud preflight?
- **Origin:** ¿de qué origen proviene la solicitud?

Cabeceras

CORS es un protocolo que habilita scripts corriendo en un navegador para interactuar con recursos de diferente orígen. Por ejemplo, si un UI desea realizar un llamado a un API a un dominio diferente, será bloqueado de realizar el llamado debido a CORS. Es una especificación de W3C implementada para casi todos los navegadores.

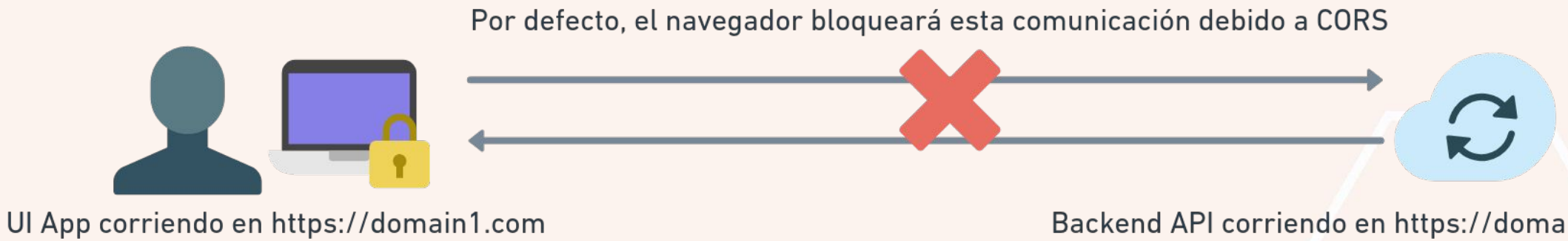
Entonces CORS no es un ataque o incidencia de seguridad, sino una protección default provista por navegadores para parar la compartición de datos / comunicaciones entre diferentes orígenes.

Cabeceras

"otros orígenes" significa que la URL que se accede es diferente de la ubicación de donde se ejecuta inicialmente el JavaScript, teniendo:

- un diferente esquema (HTTP o HTTPS)
- un dominio diferente
- un diferente puerto

Cabeceras



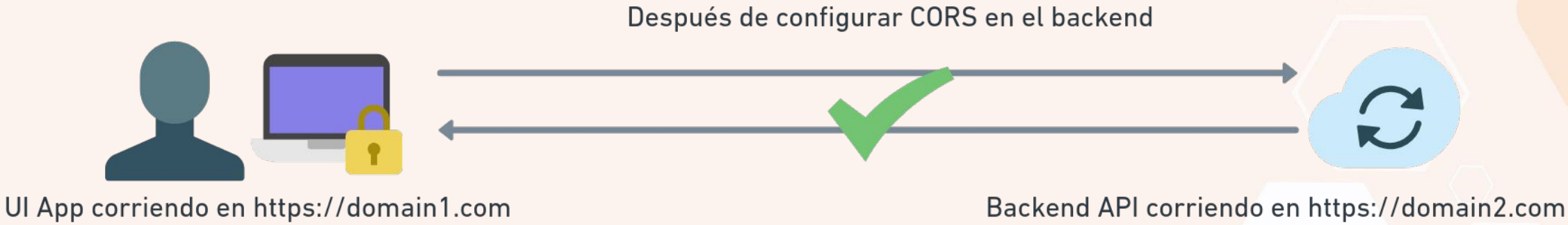
Si tenemos un escenario válido, donde una App Web (UI) deployado en un servidor, trata de comunicarse con un servicio REST deployado en otro servidor, entonces este tipo de comunicaciones las podemos habilitar con la ayuda de la anotación **`@CrossOrigin`**.

Cabeceras

`@CrossOrigin` permite a los clientes de cualquier dominio consumir el API.

La anotación **`@CrossOrigin`** puede estar mencionada hasta arriba de cada clase o método (endpoint):

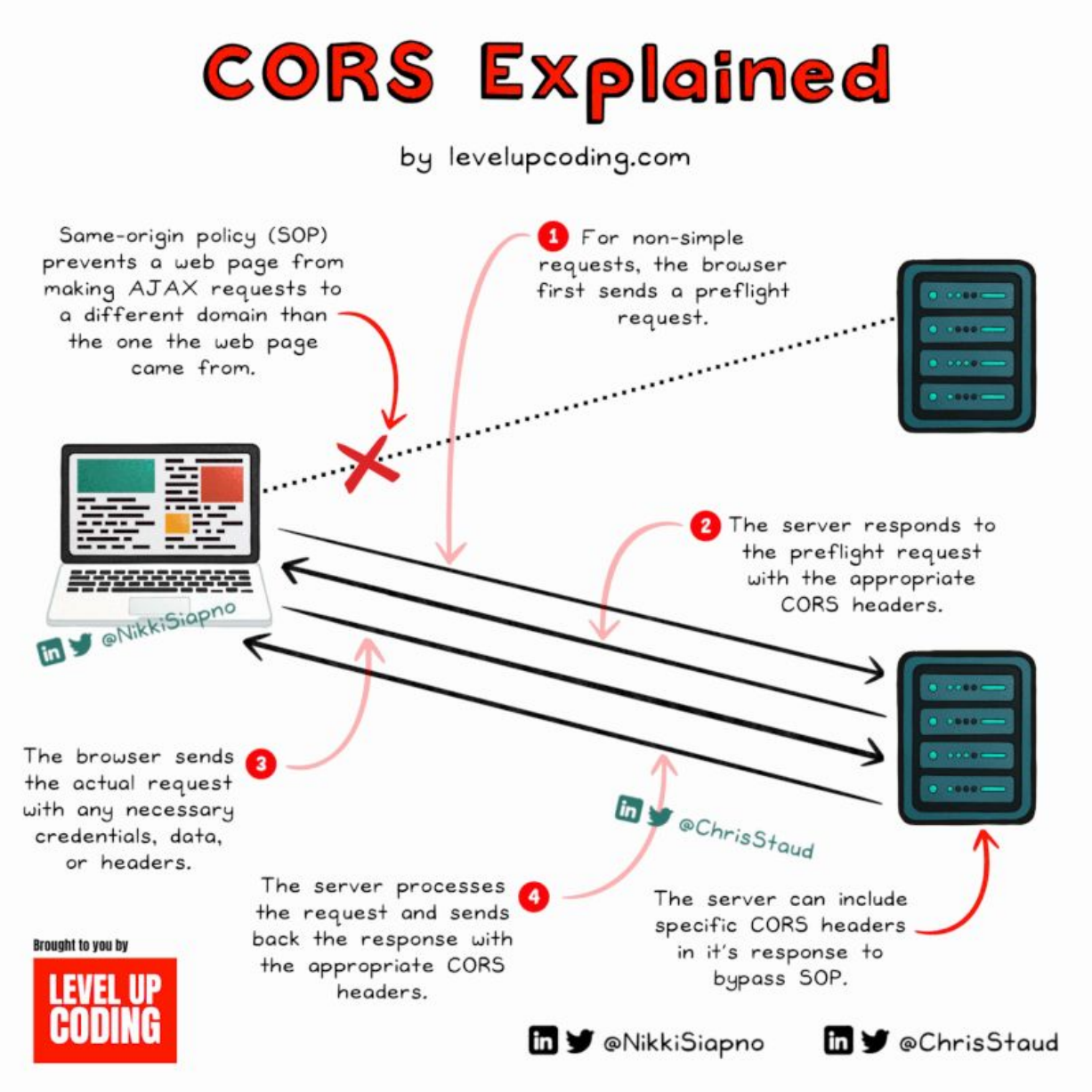
```
@CrossOrigin(origins = "http://localhost:4200") // permite acceso a este dominio
@CrossOrigin(origins = "*") // permite acceso a cualquier dominio (inseguro)
```



Cabeceras

Hay otra manera mejor de manejar esto, configurando CORS dentro del `securityFilterChain`

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.cors(corsCustomizer -> corsCustomizer.configurationSource(new CorsConfigurationSource() {
            @Override
            public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
                CorsConfiguration config = new CorsConfiguration();
                config.setAllowedOrigins(Collections.singletonList("http://localhost:4200"));
                config.setAllowedMethods(Collections.singletonList("*"));
                config.setAllowCredentials(true);
                config.setAllowedHeaders(Collections.singletonList("*"));
                config.setMaxAge(3600L);
                return config;
            }
        })).authorizeHttpRequests((requests)->requests
                .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
                .requestMatchers("/notices", "/contact", "/register").permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();
}
```


CORS Explained
by levelupcoding.com
Same-origin policy (SOP) prevents a web page from making AJAX requests to a different domain than the one the web page came from.
1 For non-simple requests, the browser first sends a preflight request.
2 The server responds to the preflight request with the appropriate CORS headers.
@NikkiSiapno
The browser sends 3 the actual request with any necessary credentials, data, or headers.
@ChrisStaud
The server processes the request and sends back the response with the appropriate CORS headers. 4
The server can include specific CORS headers in it's response to bypass SOP.
Brought to you by
LEVEL UP CODING
@NikkiSiapno
@ChrisStaud

2. CSRF

2.1 Introducción a CSRF

¿Qué es CSRF?

CSRF (Cross-Site Request Forgery) es un tipo de protección que impide que un sitio web pudiera hacer peticiones maliciosas a otro sitio web que fuese seguro, en caso de tener ambos abiertos a la vez en dos pestañas diferentes del navegador, y nos hubiésemos autenticado el segundo.

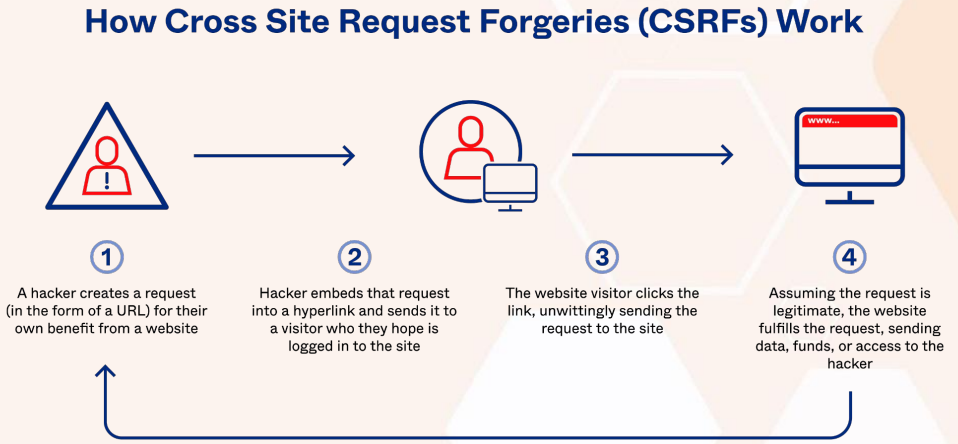
Un ejemplo sería el siguiente:

1. Iniciamos sesión en mibanco.com.
2. Sin cerrar sesión, abrimos otra pestaña y accedemos a sitiodudoso.com.

¿Qué es CSRF?

Entonces, sin CSRF, sitiodudoso.com podría tener algún tipo de mecanismo, por ejemplo, un enlace que fuese muy llamativo para pinchar en el mismo, la carga de una imagen o simplemente un script de JavaScript, que pudiera hacer una petición POST.

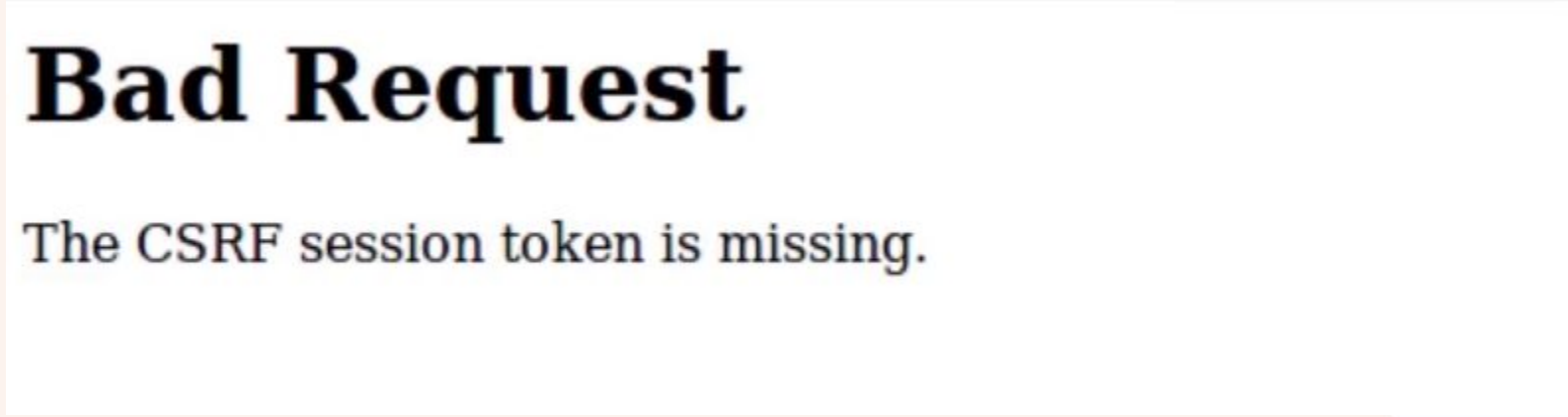
En esta petición, conociendo mínimamente la estructura de la API o aplicación en la cual estamos logueados en mibanco.com, se podría tratar de hacer un traspaso de dinero de una cuenta a otra.



¿Cómo funciona CSRF?

CSRF es un mecanismo mediante el cual, para realizar este tipo de peticiones, sobre todo las peticiones POST, el cliente tiene que enviar un token, conocido como token CSRF o XSRF.

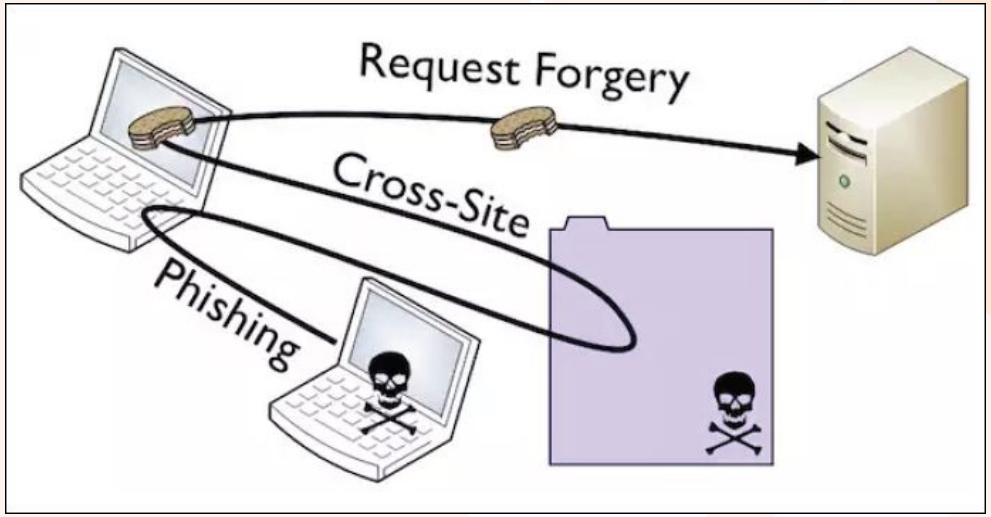
Si no se envía ese token o el que se ha enviado no es válido, la operación no sé realizaría y podríamos obtener un mensaje de respuesta como el que vemos en la imagen.



¿Cuáles son los riesgos de una vulnerabilidad CSRF?

Los riesgos se generan cuando el atacante logra que la víctima lleve a cabo una acción involuntaria. Dependiendo de la acción, el atacante podría obtener el control total sobre la cuenta del usuario.

Por ejemplo, si el usuario comprometido tiene un rol privilegiado dentro de la aplicación, entonces el atacante podría tomar el control total de todos los datos y de las funcionalidades de la aplicación.



¿Cómo funciona una vulnerabilidad CSRF?

Para que un ataque CSRF sea posible, deben cumplirse tres condiciones:

1. Existe una acción relevante que el atacante quiere inducir. Puede ser una acción privilegiada (modificar permisos para otros usuarios) o una acción sobre datos específicos del usuario (cambiar la propia contraseña del usuario).

2. El manejo de sesiones debe estar basado en cookies dado que realizar una acción implica realizar una o más solicitudes HTTP. Y si la aplicación utiliza cookies de sesión, estas siempre serán enviadas para identificar al usuario que ha realizado las solicitudes.

¿Cómo funciona una vulnerabilidad CSRF?

3. Las solicitudes no contienen parámetros impredecibles, es decir, no existen valores que el atacante tenga que determinar o adivinar. Por ejemplo, al hacer que un usuario cambie su contraseña, la función no es vulnerable si un atacante necesita conocer el valor de la contraseña existente.

Spring Security y CSRF

Spring Security, por defecto, habilita la protección CSRF, y, además, provee de muchos mecanismos para generar dicho token, donde suelen intervenir cookies.

Por ejemplo, utilizando Thymeleaf, a la hora de implementar un formulario sin tocar nada más, si vemos el código fuente que genera el código HTML, podemos ver como el servidor se ha encargado, al procesar la plantilla y generar todo el código del formulario, de añadir ese campo de token cada vez que se vaya a enviar un campo de formulario.

Si desarrollamos una API, por ejemplo, para una aplicación de Angular, desde Spring podríamos también generar dicho token y enviarlo dentro de un encabezado.

¿Es necesario CSRF para un API REST?

Algunos autores piensan que si no va a haber cookies no haría falta la protección CSRF, por varios motivos:

Los navegadores normalmente, y es algo que queda transparente para el usuario, se están enviando cookies constantemente. Esto se puede comprobar desde la inspección que ofrecen algunos navegadores, buscando dónde se ha hecho la petición, ver peticiones y respuestas y ver que intervienen bastantes cookies por detrás.

Los ataques CSRF dependen mucho de este comportamiento y de la utilización de cookies, con lo cual, si no utilizamos cookies o no se confían en ellas, no habría necesidad de protegerse frente ataques de este tipo.

¿Es necesario CSRF para un API REST?

Por otro lado, está también el argumento de que REST es stateless (sin estado), por lo tanto:

Una aplicación REST no se encargará de rastrear el estado del lado cliente.

Si se utilizan sesiones a través de cookies, estaríamos tratando de rastrear este estado y, por tanto, nuestra aplicación no es totalmente REST, y, por ende, no es totalmente stateless.

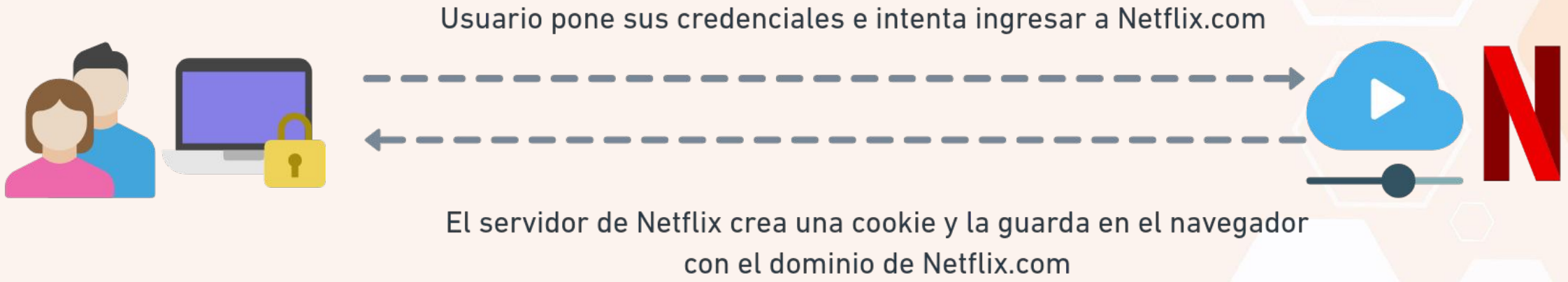
Si RESTful no debe necesitar cookies ni sesiones, por tanto, no deberíamos tener la necesidad protegernos frente a CSRF.

¿Cómo trabaja? (a detalle)

Un típico ataque CSRF trata de realizar una operación en una app web en nombre del usuario (obviamente sin su consentimiento).

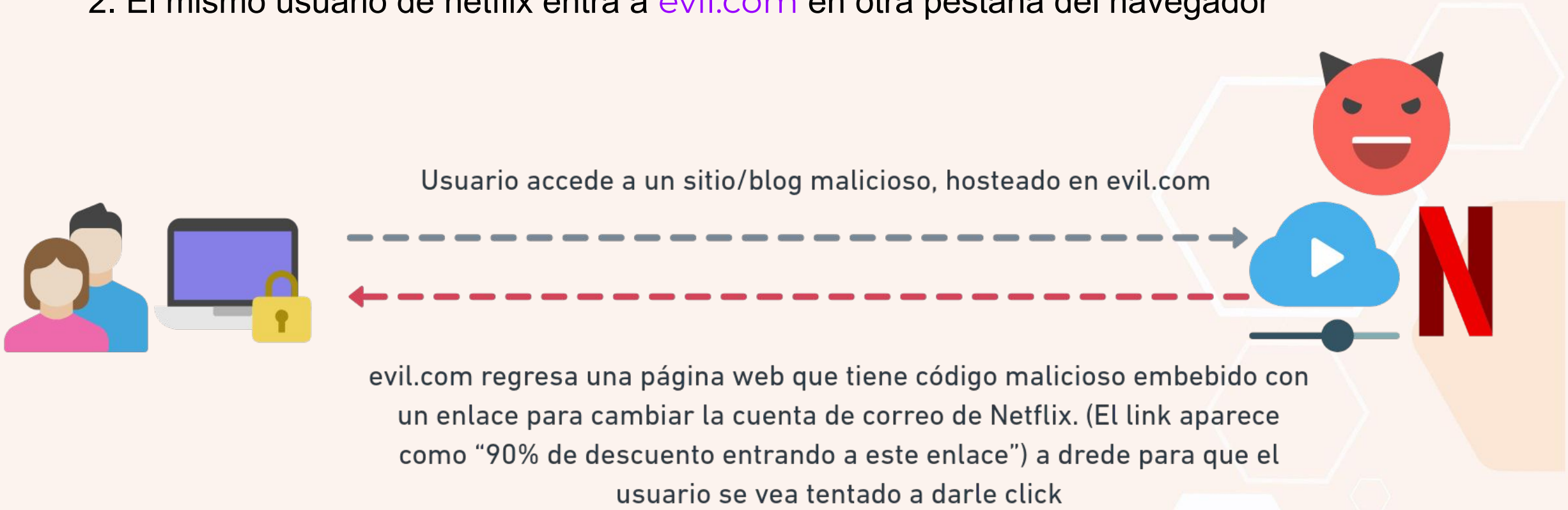
Considere ustedes estan en netflix.com y el atacante en evil.com

1. El usuario se loguea en netflix.com y el back de netflix provee una cooke que se guarda en el navegador sobre el dominio per se de netflix



¿Cómo trabaja? (a detalle)

2. El mismo usuario de netflix entra a evil.com en otra pestaña del navegador

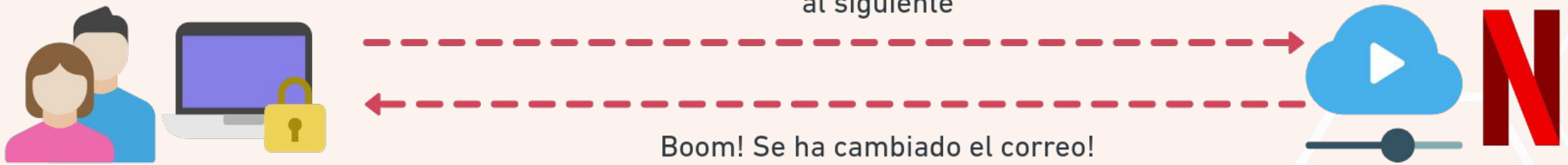


¿Cómo trabaja? (a detalle)

3. El usuario se siente tentado y hace clic en el enlace malicioso, y realiza un request a netflix.com. Como la cookie ya está presente en el navegador y el request para *cambiar el correo* se hace al mismo dominio netflix.com, el servidor de backend de netflix.com no puede diferenciar de donde viene el request! Entonces evil.com falsificó la solicitud como si fuera a venir de netflix.com

¿Cómo trabaja? (a detalle)

Usuario da click en el enlace malicioso con un contenido parecido al siguiente



Boom! Se ha cambiado el correo!

```
<form action="https://netflix.com/changeEmail"
  method="POST" id="form">
    <input type="hidden" name="email" value="user@evil.com">
<form>

<script>
    document.getElementById('form').submit()
</script>
```

Solución

Para parar el ataque CSRF, las aplicaciones necesitan una manera de determinar si el request HTTP es legítimamente generado por la interfaz del usuario. La mejor manera de realizar esto es mediante un token CSRF.

- El token CSRF es un token seguro random, que se usa para prevenir ataques. El toquen necesita ser único por sesión de usuario, y debe de ser un valor largo y random para que sea difícil de operar.

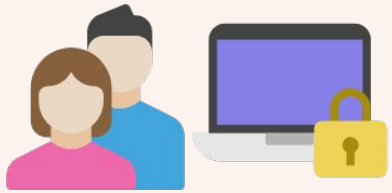
1. El usuario Netflix entra a netflix.com y el backen de netflix provee una cookie que se guarda en el navegador con el dominio netflix.com pero junto con un único token CSRF para esta particular sesión. El token CSRF es insertado mediante parámetros escondidos en forms de HTML para evitar exposición a cookies de sesión

Solución

Usuario pone sus credenciales e intenta ingresar a Netflix.com

El servidor de Netflix crea una cookie y genera un token CSRF de manera aleatoria

2. El mismo usuario netflix abre evil.com en otra tab del navegador

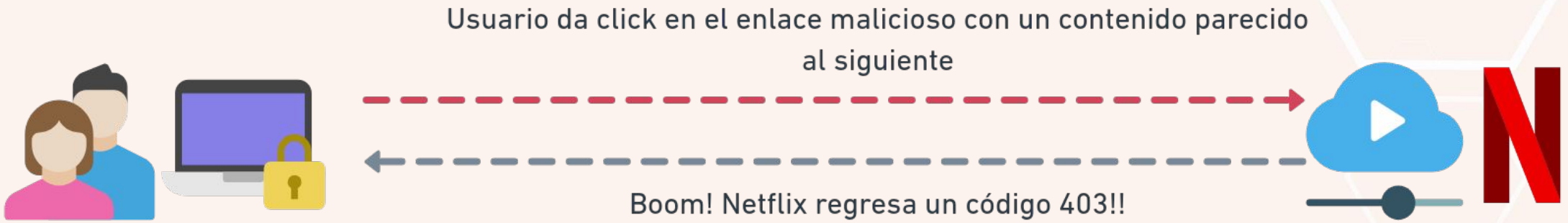


Usuario accede a un sitio/blog malicioso, hosteado en evil.com

evil.com regresa una página web que tiene código malicioso embebido con un enlace para cambiar la cuenta de correo de Netflix. (El link aparece como “90% de descuento entrando a este enlace”) a drede para que el usuario se vea tentado a darle click

Solución

3. El usuario es tentado y da click en un link malicioso que realiza un request a netflix.com. Y dado que la cookie de inicio de sesión ya está presente en el mismo navegador y la solicitud de cambio de correo electrónico se realiza en el mismo dominio netflix.com. Esta vez, el servidor backend de netflix.com espera el token CSFR junto con la cookie. El token CSRF debe ser el mismo que el valor inicial generado durante la operación de inicio de sesión.



Solución

El servidor de aplicaciones utilizará el token CSRF para verificar la legitimidad de la solicitud del usuario final si proviene de la misma interfaz de usuario de la aplicación o no. El servidor de aplicaciones rechaza la solicitud si el token CSRF no coincide con la prueba.

De forma predeterminada, la seguridad de Spring bloquea todas las operaciones HTTP POST, PUT, DELETE, PATCH con un error 403, si no hay una solución CSRF implementada dentro de una aplicación web. Podemos cambiar este comportamiento predeterminado deshabilitando la protección CSRF proporcionada por Spring Security.

Solución

```java
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf((csrf) -> csrf.disable())
        .authorizeHttpRequests((requests)->requests
            .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
            .requestMatchers("/notices", "/contact", "/register").permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();
}
```

Solución

Con la configuración dada de Spring Security, podemos permitir que el marco genere un token CSRF aleatorio que se puede enviar a la interfaz de usuario después de iniciar sesión correctamente. La interfaz de usuario debe enviar el mismo token para cada solicitud posterior que realice al backend. Para ciertas rutas, podemos desactivar CSRF con la ayuda de `ignoreRequestMatchers`

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
    requestHandler.setCsrfRequestAttributeName("_csrf");

    http.securityContext((context) -> context
            .requireExplicitSave(false))
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.ALWAYS))
        .cors(corsCustomizer -> corsCustomizer.configurationSource(new CorsConfigurationSource() {...}))
        .csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler).ignoringRequestMatchers("/contact", "/register")
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
            .addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
        .authorizeHttpRequests((requests)->requests
            .requestMatchers("/myAccount", "/myBalance", "/myLoans", "/myCards", "/user").authenticated()
            .requestMatchers("/notices", "/contact", "/register").permitAll())
        .formLogin(Customizer.withDefaults())
        .httpBasic(Customizer.withDefaults());
    return http.build();
}
```

Conclusiones

De todo lo anterior, podemos sacar varias conclusiones

- Si de alguna manera vamos a acabar usando cookies, sí que vamos a necesitar este tipo de protección que ofrece CSRF.

- Por ejemplo, si en lugar de almacenar el token JWT dentro del almacenamiento local del navegador, que quizás no sea una buena práctica, lo queremos mantener a través de alguna cookie, en ese caso tendríamos que tener cuidado y sería bueno que implementásemos la protección CSRF.

Conclusiones

- Si lo que estamos desarrollando son aplicaciones web como tales, que tengan un motor de plantillas cómo podría ser Thymeleaf, la protección CSRF se convierte en una auténtica obligación, a la cual nos va a ayudar Spring Security.

`https://docs.spring.io/spring-security/reference/features/exploits/csrf.html`

Contacto

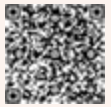
Mtro. Alfonso Gregorio Rivero Duarte

Senior Data Manager - CBRE

devil861109@gmail.com

Tels: (+52) 55 289970 69

Redes sociales:



https://www.linkedin.com/in/alfonso-gregorio-rivero-duarte-139a9225/