

# Análisis del ataque CSRF en aplicaciones web

Alfredo Jesús Campos Inga

Universidad de Lima  
20190345@aloe.ulima.edu.pe

## Abstract

Con el pasar de los años, internet se ha ido diversificando alrededor de todo el mundo, siendo totalmente necesario en la actualidad para poder llevar a cabo muchas de nuestras actividades del día a día. Asimismo, con el aumento exponencial de usuarios con internet en todo el planeta, también se ha visto un incremento sustancial en la cantidad de sitios web, pasando de tener objetivos científicos en sus inicios a poder ser realizadas por cualquier persona. Sin embargo, no todos los objetivos actuales de los sitios web son bien intencionados, sino que existen algunos maliciosos que se aprovechan de vulnerabilidades que presentan otros con distintos motivos. De entre todas estas amenazas, está el Cross-Site Request Forgery (CSRF), que en sus inicios fue bastante peligroso y fue el motivo por el que nacieron los tokens en los sitios web actuales. A continuación, en el presente artículo de investigación se especificará en qué consiste el ataque CSRF y cómo se puede prevenir haciendo uso de tokens. Del mismo modo, se llevará a cabo una experimentación donde se detalle de manera práctica todo lo mencionado previamente, tanto el ataque CSRF como su prevención mediante tokens.

## Palabras clave

Vulnerabilidad web, amenaza web, OWASP, CSRF, tokens

## Introducción

Hace más de 30 años atrás, un 6 de agosto de 1991, el científico británico Tim Berners-Lee publicó el primer sitio web de la historia de manera pública en el que explicaba su proyecto de la World Wide Web (W3). De acuerdo a Armstrong (2021), para finales de 1992 habían tan solo diez sitios web, que fue cuando el CERN hizo pública la tecnología de la W3, lo que hizo que poco a poco se incrementara la cantidad de sitios web alrededor del mundo hasta llegar a tener la cantidad monstruosa que conocemos hoy en día. En el año 1994, el número de sitios web en internet se incrementó a aproximadamente 3000, pasaron los años y este número fue en aumento, llegando a estar en 238 millones hace poco más de 10 años y, en el año 2021, superar más de 1'800 millones de sitios web (Armstrong, 2021). Esto lo podemos apreciar en el siguiente gráfico.

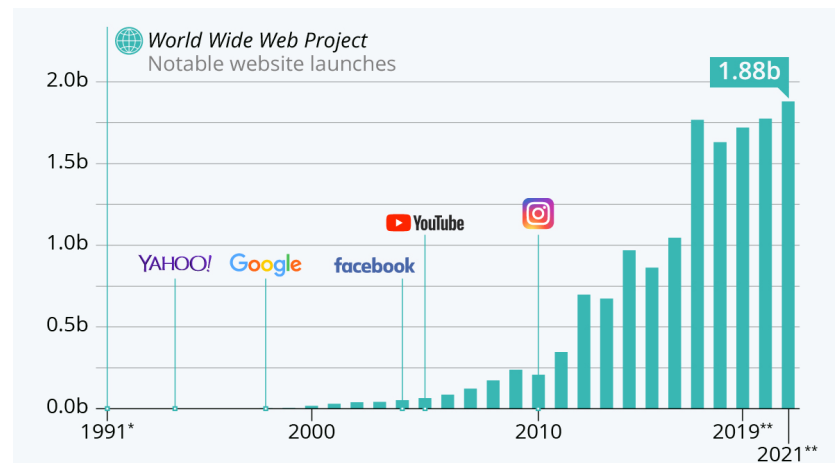
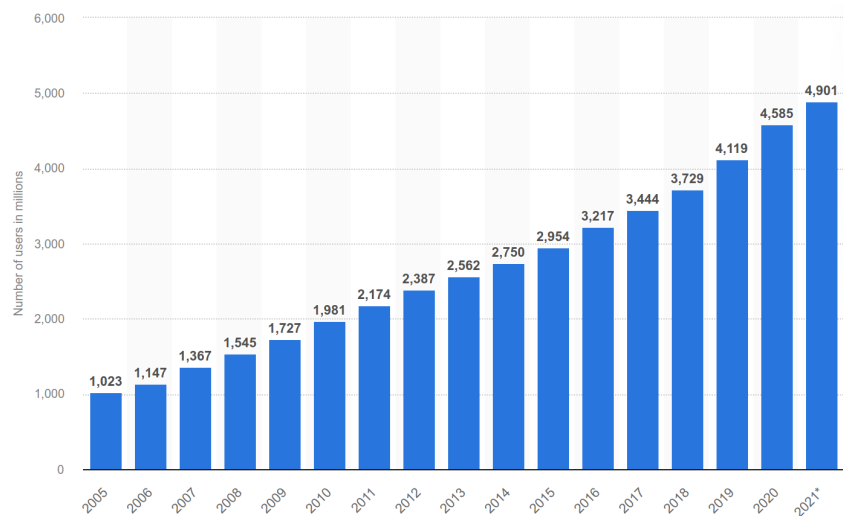


Figura 1. Cantidad de sitios web a lo largo del tiempo (Armstrong, 2021).

Sin embargo, no solo la cantidad de páginas web ha ido en aumento, sino que también el número de usuarios de internet ha crecido de manera exponencial en los últimos años. De acuerdo con Johnson (2021), en el año 2005 internet contaba con 1'023 millones de usuarios, y para el año 2021 llegó a más de 4'901 millones de usuarios alrededor de todo el mundo, lo que representa más del 62% de la población mundial actual. Este incremento de usuarios podemos apreciarlo a continuación.



**Figura 2. Cantidad de usuarios de internet a lo largo del tiempo (Johnson, 2021).**

Estos dos datos nos demuestran que, en la actualidad, el internet y los sitios web están muy presentes entre todos nosotros, e inclusive forman parte importante de nuestras vidas. Asimismo, nos deja en evidencia que son muchas las personas que, hoy en día, hacen muchas de sus actividades cotidianas a través de sitios web, tales como buscar información acerca de un tema, entretenimiento, para estudiar, trabajar, entre otros. En este sentido, los sitios web deben de almacenar información de sus clientes para ofrecerles la mejor experiencia de usuario posible. Sin embargo, no todos estos son desarrollados de manera segura, sino que dejan expuestas ciertas vulnerabilidades que pueden llegar a ser explotados por usuarios maliciosos para llevar a cabo lo que les apetezca, como robar datos de usuarios, borrar datos, modificarlos, etc.

Si revisamos en la OWASP, proyecto open source dedicado a identificar y combatir causas que hacen inseguros a los software, en el año 2021 actualizaron su ranking de riesgos de seguridad de aplicaciones web, en el que podemos visualizar algunas vulnerabilidades comunes de los sitios web como Broken Access Control, inyecciones, malas configuraciones de seguridad, Server-Side Request Forgery, entre otros (OWASP, 2021). Pasemos a hacer una revisión de la literatura para darnos una idea de lo que es cada uno de estos.

En primer lugar, Broken Access Control se posiciona en el primer lugar del ranking de OWASP, donde el 94% de las aplicaciones web donde realizaron pruebas demostraron sufrir de esta vulnerabilidad (OWASP, 2021). De acuerdo con He et al. (2021), Broken Access Control es una vulnerabilidad donde hay una falta de mecanismos de control de acceso que le permite a usuarios no autorizados ser capaces de acceder a activos de información sin solicitar los permisos respectivos. De este modo, estos usuarios podrían explotar esta vulnerabilidad para acceder a cuentas de otros usuarios, ver información sensible, modificar datos de otros usuarios, etc (He et al., 2021). Esto se logra accediendo a las cookies del usuario objetivo, por ejemplo, mediante técnicas de ingeniería social, con lo que se conectan a la misma sesión desde otro ordenador. Este es un fallo grave de la aplicación web ya que no está manejando sesiones y cookies de manera adecuada (He et al., 2021).

En segundo lugar, encontramos las inyecciones, que en el año 2017 estaba colocada en el primer lugar del ranking de OWASP, y actualmente se ubica en la tercera posición del mismo ranking siendo un 94% de las aplicaciones web evaluadas las que sufrían de esta (OWASP, 2021). Según Stasinopoulos et al. (2018), al hablar de inyecciones nos referimos a insertar código arbitrario dentro de una aplicación vulnerable, lo que da como resultado un comportamiento de ejecución no planificado. Hay muchos tipos de ataques de inyecciones de código, tales como las inyecciones de comandos, las inyecciones SQL, las secuencias de comandos entre sitios, las inyecciones de XPath y las inyecciones de LDAP (Stasinopoulos et al., 2018).

En tercer lugar, tenemos las malas configuraciones de seguridad, el cual ocupa el puesto 5 en el ranking de OWASP y se encontró en el 90% de las aplicaciones web evaluadas (OWASP, 2021). De acuerdo con Loureiro (2021), esta vulnerabilidad indica que las configuraciones de seguridad de la aplicación web en cuestión no cumple con los estándares de seguridad de la industria, como los benchmarks del Center for Internet Security (CIS), el Top 10 de OWASP, entre otros. Dicho de otro modo, esta vulnerabilidad ocurre cuando el administrador de sistemas, el desarrollador y/o el administrador de base datos no configuran de manera adecuada el marco de seguridad del sistema, aplicación web y/o base de datos, respectivamente, lo que es una oportunidad para usuarios malintencionados para infiltrarse en los servidores, las instancias en la nube, las aplicaciones web, etc, lo que pone en riesgo la continuidad de negocio y la reputación de la empresa. Un ejemplo de esto es la exposición de Teletext de 530'000 archivos de datos en el año 2019, que fue causada por un servidor de Amazon Web Services configurado de manera insegura (Loureiro, 2021).

Por último, tenemos a Server-Side Request Forgery, más conocido como SSRF, el cual se encuentra en la décima posición del ranking de OWASP y fue la más elegida en la encuesta de la comunidad (OWASP, 2021). El ataque SSRF consiste en que el atacante puede leer y controlar los recursos internos del servidor atravesando el firewall del mismo enviando una URL dentro de una petición web a la aplicación web (Al-talak & Abbass, 2021). Un ejemplo de este ataque ocurrió en el año 2019, cuando un atacante pirateó la base de datos de Capital One Bank usando las credenciales de Amazon Web Services, que luego usó para acceder a la base de datos de Capital One, robando en total más de 100 millones de información de clientes (Al-talak & Abbass, 2021). Cabe resaltar que, así como este ataque apunta principalmente al servidor, hay otro que usa técnicas muy parecidas, pero que apunta a un usuario en específico. Este ataque se llama Cross-Site Request Forgery. En la presente investigación, se abordará este último, se detallará en lo que consiste este ataque y las vulnerabilidades de las que se aprovecha. Luego de esto, se desarrollará una experimentación donde se evidenciará lo mencionado en el detalle del ataque. Finalmente, se brindarán controles para poder prevenirlo.

## Marco teórico

Cross-Site Request Forgery, también conocido como CSRF por sus siglas, es uno de los ataques más antiguos que existe en la web que consiste en obligar al usuario hacer una petición HTTP sin su consentimiento, siendo descubierto a principios de la década del 2000, pero aún así en la actualidad sigue siendo muy efectivo en muchos sitios web (Calzavara et al., 2020). De acuerdo con Pellegrino et al. (2017), CSRF se posicionó en el ranking de OWASP durante muchos años consecutivos junto con otras dos vulnerabilidades web, Cross-Site Scripting (XSS) y SQL Injection (SQLi), hasta el año 2017 que fue retirado. Asimismo, en el año 2020, Acunetix, solución integral de seguridad de aplicaciones web, realizó pruebas de vulnerabilidad a aproximadamente 5'000 sitios web seleccionados de manera aleatoria, en donde se encontró que 36% de estos eran vulnerables a ataques CSRF (Cheah & Selvarajah, 2021).

Para poder entender cómo es que funciona CSRF, veamos el siguiente modelo propuesto por Calzavara et al. (2020).

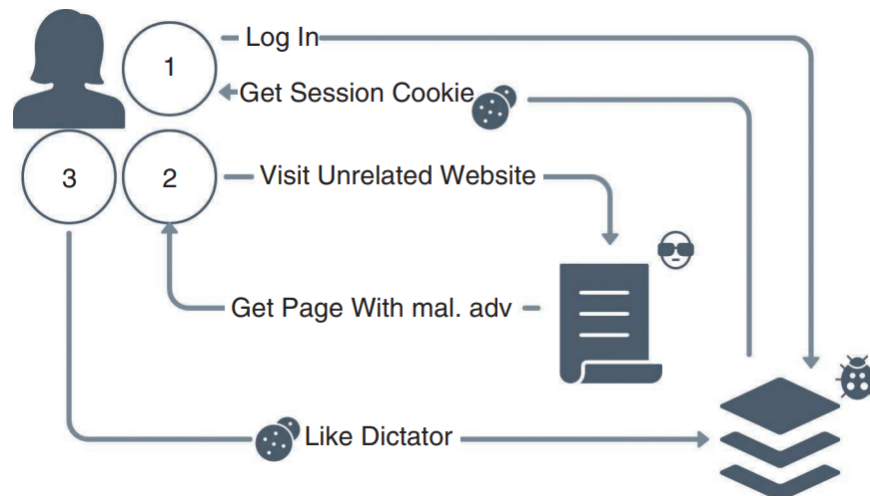


Figura 3. Modelo de ataque CSRF (Calzavara et al., 2020).

El funcionamiento de CSRF es el siguiente según Calzavara et al. (2020).

1. El usuario objetivo inicia sesión en alguna plataforma online. Al haber iniciado sesión, se almacenaron las cookies en el navegador.
2. El usuario objetivo abre otra pestaña y entra en un sitio web malicioso, o entra a un sitio web confiable y recibe un anuncio malicioso.
3. El usuario realiza una acción cualquiera, ya sea en el sitio web malicioso o en el anuncio malicioso, lo que dispara una petición HTTP vía HTML o JavaScript al sitio web en el que se ingresó en el primer punto. Esta petición es procesada con éxito ya que es validada con las credenciales del usuario que se encuentran en las cookies.

Cabe resaltar que lo mencionado en el punto 2 no se limita únicamente a entrar a un sitio web malicioso, sino que esto también puede ocurrir mediante phishing. Y es que se le puede enviar un correo electrónico a las víctimas con una URL mal intencionada que, tal y como en el punto 3, disparará una petición HTTP al sitio web objetivo donde se encuentra autenticado (Cheah & Selvarajah, 2021).

Asimismo, es esta petición HTTP donde se encuentra el peligro de este ataque, ya que, depende de los conocimientos del atacante, se pueden llevar a cabo desde cosas muy sencillas y aparentemente inofensivas como dar like a una determinada publicación en una red social (Calzavara et al. 2020) hasta temas complejos realizar transferencias económicas a una determinada cuenta bancaria (Cheah & Selvarajah, 2021), todo esto sin autorización previa del usuario.

De acuerdo con Pellegrino et al. (2017), no se le ha prestado suficiente atención a los ataques CSRF en comparación a otros como XSS o SQLi, y es que en estos dos últimos se han desarrollado técnicas de análisis dinámicos y estadísticos para detectarlos, mientras que esto no ocurre con CSRF, donde se debe de hacer inspecciones manuales para identificar si la vulnerabilidad existe. Sin embargo, aún así existen algunos métodos a seguir para prevenir CSRF. Se podría pensar que se puede prevenir CSRF agregando verificaciones extras para el usuario, solicitar verificaciones como con un captcha, o añadir atributos extras a las cookies, pero esto no es suficiente. De acuerdo con Calzavara et al. (2020), CSRF se evita de las siguientes maneras de las siguientes maneras:

- Verificando el valor estándar de las cabeceras de las peticiones HTTP estándar, como Referrer y Origin, para indicar al sitio web y, por lo tanto, al usuario, que se está originando una petición HTTP.
- Verificar la presencia de encabezados personalizados de la petición HTTP, como X-Requested-With, que no se pueden configurar desde una posición cross-site.
- Verificar la presencia de tokens CSRF impredecibles, configurados por el servidor en formas cambiantes.

Esto último relacionado con tokens CSRF se encuentra presente en la actualidad en diferentes frameworks de desarrollo web, como es el caso de Django, Spring, ASP.NET, Laravel, entre otros, lo que le permite a los desarrolladores lidiar de una manera más sencilla ante esta vulnerabilidad siempre y cuando hagan un uso adecuado de las herramientas que les proporciona el framework.

## **Metodología**

Para la presente investigación, se elaboró una simulación del ataque CSRF, así como su prevención mediante el uso de tokens CSRF. Todo esto se pasa a detallar a continuación.

### **Especificación del entorno de pruebas**

El desarrollo de la experimentación y las pruebas se llevaron a cabo en el sistema operativo Linux, en la distribución de ArcoLinux. Esta distribución está basada en Arch Linux y sigue una filosofía de rolling release, que indica que no existe una versión específica del sistema operativo y sus distribuciones derivadas, sino que se lanzan actualizaciones constantes para que sus usuarios siempre estén al día. Asimismo, como shell se utilizó fish en su versión 3.4.1.

Además, se desarrollaron dos sitios web, ambos en NodeJS en su versión 18.3.0 con NPM en su versión 8.5.5, haciendo uso del framework de Express en su versión 4.18.1. Cabe resaltar que es recomendable tener conocimientos básicos de backend y del protocolo HTTP para entender de la mejor manera posible lo que se explicará. Los detalles para poder acceder al código fuente se encuentran en el **Anexo 1**.

## **Descripción del sitio web objetivo**

El primer sitio web, el cual es el objetivo, cuenta con un login, un home, un logout y una ruta para editar el email del usuario. Todas estas rutas están protegidas por middlewares, los cuales se encargan de asegurar que, para que un usuario pueda acceder a todas las funcionalidades, primero debe de autenticarse en el servidor iniciando sesión en el login. Además, de todas estas rutas presentadas, la última mencionada cuenta con una vulnerabilidad ante un ataque CSRF. Esto se debe a que, para cambiar el email, se cuenta con un formulario que es el mismo para todos los usuarios, no cuenta con un diferenciador de sesiones como lo son los tokens.

## **Descripción del atacante**

Es en este punto donde entra en juego el atacante, el cual conoce la vulnerabilidad del sitio web objetivo debido a que ha estudiado la manera en la que opera su backend haciendo uso de herramientas como Burp Suite para analizar el tráfico HTTP. Sabiendo esto, este atacante tiene un sitio web camuflado, que parece que no hace nada, pero que internamente, cuando termina de cargar, manda una petición HTTP haciendo uso de JavaScript a la ruta sensible del sitio web objetivo, haciendo creer al servidor que el usuario que se encuentra autenticado dentro de este fue quien solicitó cambiar su email, cuando en verdad esto no es así.

## **Descripción de la prevención**

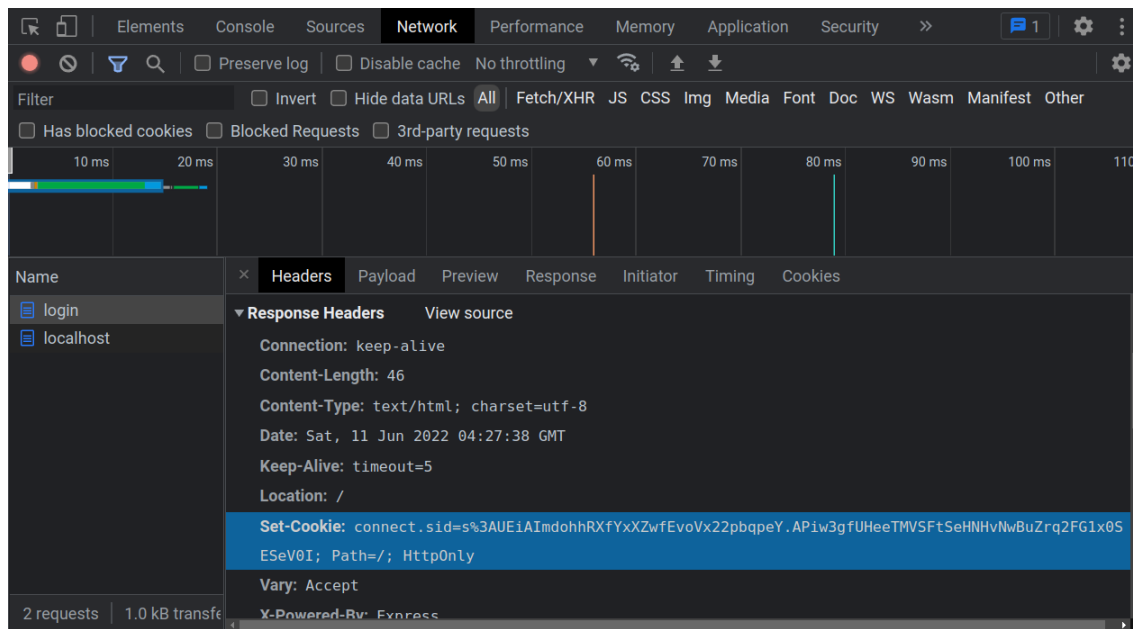
Para finalizar, se brindará una de las posibles soluciones ante este ataque. Como menciona Calzavara et al. (2020), existen distintas maneras de prevenir el ataque CSRF, de las cuales se optó por implementar tokens CSRF debido a que, actualmente, prácticamente todos los frameworks cuentan con librerías para prevenir este ataque de la forma más sencilla para los desarrolladores. Sin embargo, cabe resaltar que, con motivos de examinar a detalle cómo es que funcionan estos tokens, no se hizo uso de la librería de Express para evitar el ataque CSRF, sino que se desarrolló manualmente una función y un middleware para poder contrarrestar esto.

## **Resultados**

Como se mencionó anteriormente, se desarrolló tanto el ataque CSRF como su prevención a través de tokens CSRF. Estos dos se detallan a continuación.

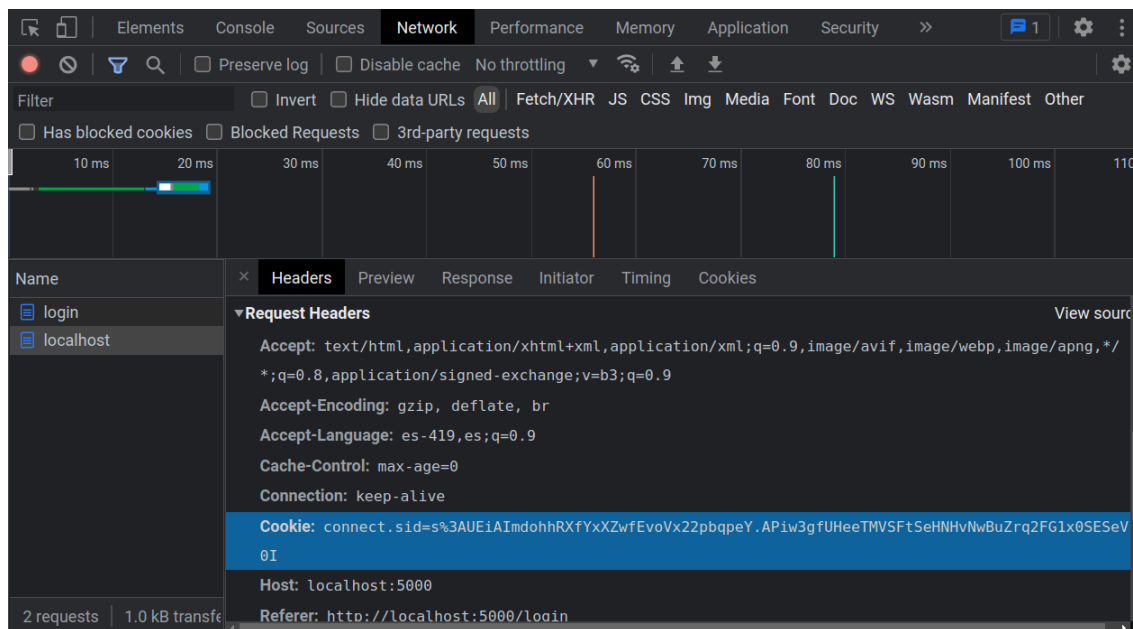
### **Ataque CSRF**

Para comenzar, se explicará el ataque CSRF que se desarrolló. Por un lado, tenemos el login. Se deben de introducir las credenciales de “test@test.com” para el email y “123” para la contraseña. Hecho esto, el usuario será redirigido al home. Una vez iniciada sesión, si pasamos a revisar las cabeceras de respuesta, nos encontraremos con lo siguiente.



**Figura 4. Cabecera HTTP del login.**

Vemos que, una vez un usuario inicie sesión de manera satisfactoria, envía como respuesta una cabecera *Set-Cookie* que tiene un hash con el id de la sesión del usuario. Si revisamos las cabeceras que solicita el home, veremos que pide este id de la sesión de usuario, esto en la cabecera de *Cookies*.



**Figura 5. Cabecera HTTP del home.**

Con estas cookies, el servidor sabe quién es este usuario en específico. Esto es debido a que el servidor guarda en el navegador del usuario estas cookies con el id de su sesión y las utiliza para poder reconocerlo entre todos los usuarios que tiene almacenado en su base de datos, esto porque tiene una estructura de datos que asocia los id de las sesiones de cada usuario a un objeto clave-valor, que en este caso son los id de la base de datos de estos usuarios. De este modo, cuando se le hace una petición al servidor, este sabe quién se la ha hecho.

Ahora, revisemos el contenido de la ruta vulnerable del sitio web objetivo, la cual es la de cambiar el email del usuario. El código fuente del mismo se muestra a continuación.

```
<form action="/edit" method="POST">
  <input type="email" name="email" placeholder="Nuevo email">
  <input type="submit">
</form>
```

**Figura 6. Formulario sensible para editar email.**

Como se puede apreciar, simplemente se tiene un formulario de tipo POST que contiene un campo para que el usuario ingrese su nuevo email y un botón para aceptar este cambio. Es acá donde encontramos la vulnerabilidad ante un ataque CSRF, que es que este formulario sea exactamente igual para todos los usuarios sin importar su sesión.

Sabiendo todo esto por parte del objetivo, pasemos a revisar al atacante. El código fuente desarrollado para su sitio web se presenta a continuación.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="index.js"></script>
  <title>Web atacante</title>
</head>
<body>
  <h1>Web atacante camuflada</h1>

  <form name="form" action="http://localhost:5000/edit" method="POST">
    <input type="hidden" name="email" value="attacker@csrf.com">
  </form>
</body>
</html>
```

**Figura 7. Código fuente del sitio web del atacante.**

Como se puede apreciar, simplemente se tiene un título con un formulario, esto es debido a que el contenido del sitio web no afecta al ataque CSRF, pero en caso de un ataque real, el atacante debería de elaborar el sitio web de manera atractiva para que el usuario no tenga ni idea que será víctima del mismo.

Este formulario es igual al del objetivo, con la diferencia de que está escondido, por lo que un usuario común no sabrá de su existencia a menos que se ponga a revisar la consola del navegador. Del mismo modo, vemos que este formulario apunta a la ruta sensible del objetivo, la de editar email, y que en el campo donde el usuario ingresaría su nuevo email se tiene como valor por defecto “attacker@csrf.com”. Sin embargo, este formulario por sí solo no hace nada. Para hacer que se envíe justo cuando se termine de cargar el sitio web, se tiene un archivo JavaScript el cual se presenta a continuación.

```
window.addEventListener('load', () => {
  document.form.submit()
})
```

**Figura 8. Código JavaScript para envío de formulario.**

Por lo tanto, si el usuario entra al sitio web del atacante, ya sea por phishing o cualquier otro motivo, el formulario se enviará automáticamente y se le cambiará el email sin su consentimiento. Sin embargo, este método no es nada sigiloso ya que, al tener un formulario que apunta a una URL y se envía con JavaScript, una vez se ingrese al sitio web atacante el usuario será redirigido automáticamente al sitio web objetivo, donde se le indicará al usuario que ha cambiado su email, por lo que podría tomar las medidas necesarias para revertir esto.

Es por esto que un atacante haría uso de métodos más discretos, como lo es mediante peticiones HTTP utilizando la función fetch de JavaScript. Esto lo vemos a continuación.

```

window.addEventListener('load', () => {
  // document.form.submit()

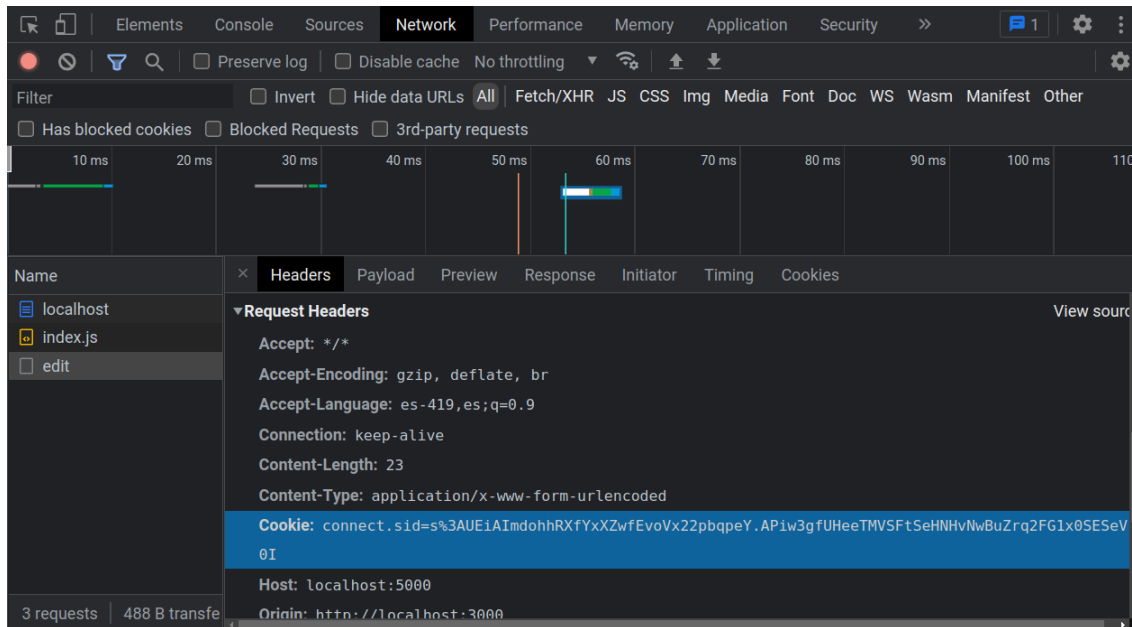
  fetch('http://localhost:5000/edit', {
    method: 'POST',
    credentials: 'include',
    mode: 'no-cors',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
    },
    body: 'email=attacker@csrf.com'
  })
})

```

**Figura 9. Código JavaScript para envío de formulario mediante petición HTTP.**

Lo que indica esta función fetch es que se va a realizar una petición HTTP de tipo POST a la ruta para editar email del sitio web objetivo. Se están incluyendo las credenciales del usuario, es decir las cookies, por lo que el servidor al recibir la petición HTTP revisará estas cookies y obtendrá el id de la sesión del usuario, verá que se encuentra autenticado y dará luz verde a dicha petición HTTP.

Básicamente, esta función hace lo mismo que el formulario presentado previamente, pero al ser enviado por una petición HTTP, hace que el usuario objetivo no se entere a menos que se ponga a revisar las cabeceras de red del navegador, las cuales pasaremos a analizar a continuación.



**Figura 10. Cabecera HTTP de editar email con CSRF.**

Como se puede apreciar, en la petición HTTP se tuvo como destino el sitio web objetivo y se incluyó el id de la sesión del usuario, por lo que el servidor objetivo creyó que se trataba del usuario y permitió que el cambio de email se lleve a cabo. Esto lo revisamos en los logs de la terminal del servidor objetivo.



```

CSRF/target-server master v1.0.0 v18.3.0
> npm run start

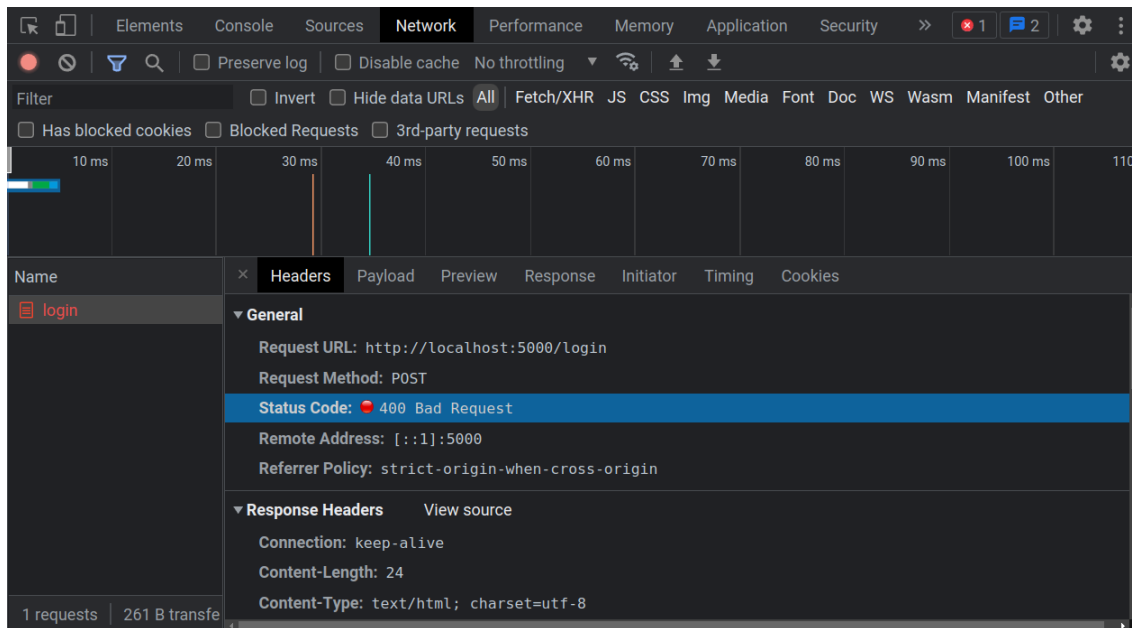
> csrf@1.0.0 start
> node src/app.js

Servidor corriendo en el puerto 5000
El usuario id: 1, email: test@test.com ha cambiado su email a attacker@csrf.com

```

**Figura 11. Log del servidor objetivo cuando un usuario ingresa al sitio web del atacante.**

Del mismo modo, como se mencionó previamente, el usuario no notará nada ya que el cambio fue a través de una petición HTTP, a menos que cierre sesión y luego quiera autenticarse nuevamente. En este caso, ingresaría las credenciales con las que creó su cuenta, pero no se le permitiría ingresar debido a que no se reconocen las credenciales que ingresa. Esto lo vemos a continuación.



**Figura 12. Cabecera HTTP de login no válido.**

Examinando las peticiones de red, vemos que se le devuelve un error al usuario al momento de intentar ingresar con sus credenciales debido a que estas no se encuentran en la base de datos. Específicamente, vemos que se le retorna el error *400 Bad Request*, el cual indica que el servidor no pudo procesar la petición debido a un error del usuario, como error de sintaxis, algún mensaje no válido, etc. Sin embargo, el usuario no se está equivocando escribiendo las credenciales con las que se registró, sino que estas han sido cambiadas por el atacante. Por lo tanto, se puede concluir que el ataque CSRF fue ejecutado con éxito.

## Tokens CSRF

Llegado a este punto, se ha detallado cómo es que funciona el ataque CSRF. Ahora, se pasará a detallar una manera de contrarrestarlo, que es haciendo uso de tokens CSRF. El código fuente desarrollado para el servidor para hacer uso de estos tokens se muestra a continuación.

```
// CSRF TOKENS

const tokens = new Map()

const csrfToken = (sessionId) => {
  const token = uuid()
  tokens.get(sessionId).add(token)
  setTimeout(() => tokens.get(sessionId).delete(token), 30000)

  return token
}

// CSRF Middleware
const csrf = (req, res, next) => {
  const token = req.body.csrfToken

  if (!token || !tokens.get(req.sessionID).has(token)) {
    return res.status(422).send('Falta el token CSRF o ha caducado')
  }
  next()
}
```

**Figura 13. Código fuente de la implementación de tokens CSRF.**

Básicamente, estamos creando un hashmap llamado *tokens* el cual asociará el id de la sesión cada usuario que se encuentra autenticado en el servidor con un id único, un token, al que le llamaremos *token*. La generación de este token se lleva a cabo en la función que tiene por nombre *csrfToken*, la cual recibe como parámetro el id de la sesión del usuario, genera un id único, lo almacena en la sesión del usuario y se lo enviará al formulario para editar el email. Cabe resaltar que, en este caso, se está usando el paquete *uuid v4* de NodeJS para la generación de los tokens, el cual retorna una cadena de 128 bits que se conforma de 32 caracteres totalmente aleatorios y 4 guiones, por lo que la probabilidad de que se repita un token es totalmente o incluso nula, ya que se cuentan con  $16^{32}$  posibilidades (NPM, 2020). Sin embargo, este token dura solamente 30 segundos, una vez llegado a ese tiempo, el mismo será borrado de la sesión del usuario.

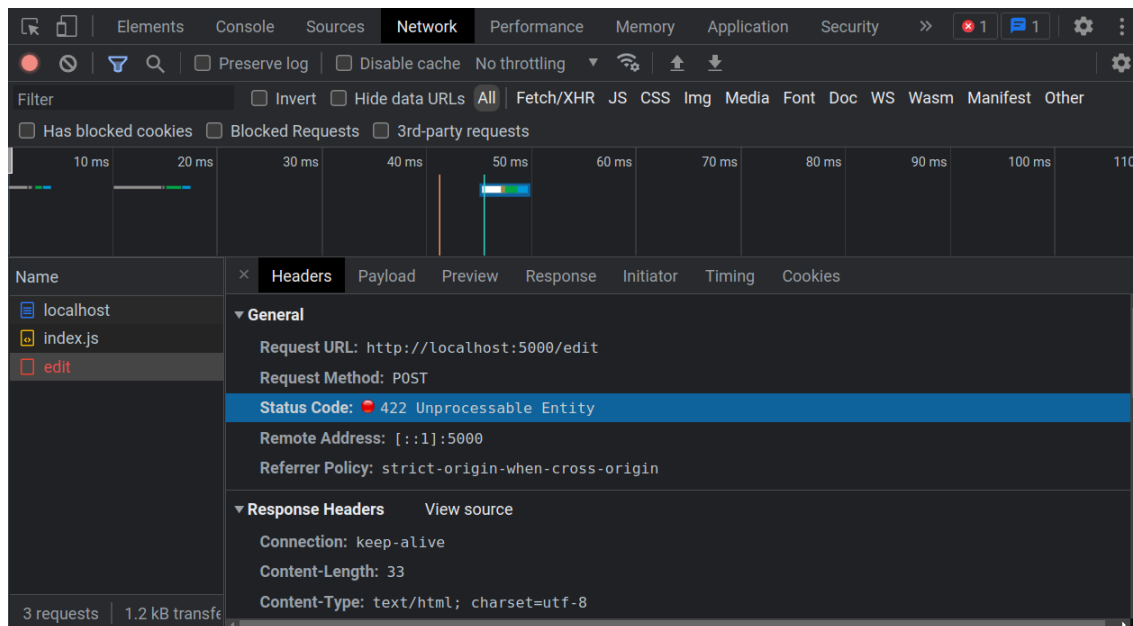
De igual modo, el formulario que se revisó en la **Figura 6** cambia ligeramente, ya que, como se acaba de mencionar, ahora cuenta con un campo extra escondido el cual contiene el token CSRF que el servidor generó para la sesión del usuario cuando este ingresó a cambiar su email. Esto lo vemos a continuación.

```
<form action="/edit" method="POST">
  <input type="email" name="email" placeholder="Nuevo email">
  <input type="hidden" name="csrfToken" value="<%= csrfToken %>">
  <input type="submit">
</form>
```

**Figura 14. Formulario para cambiar email con token CSRF.**

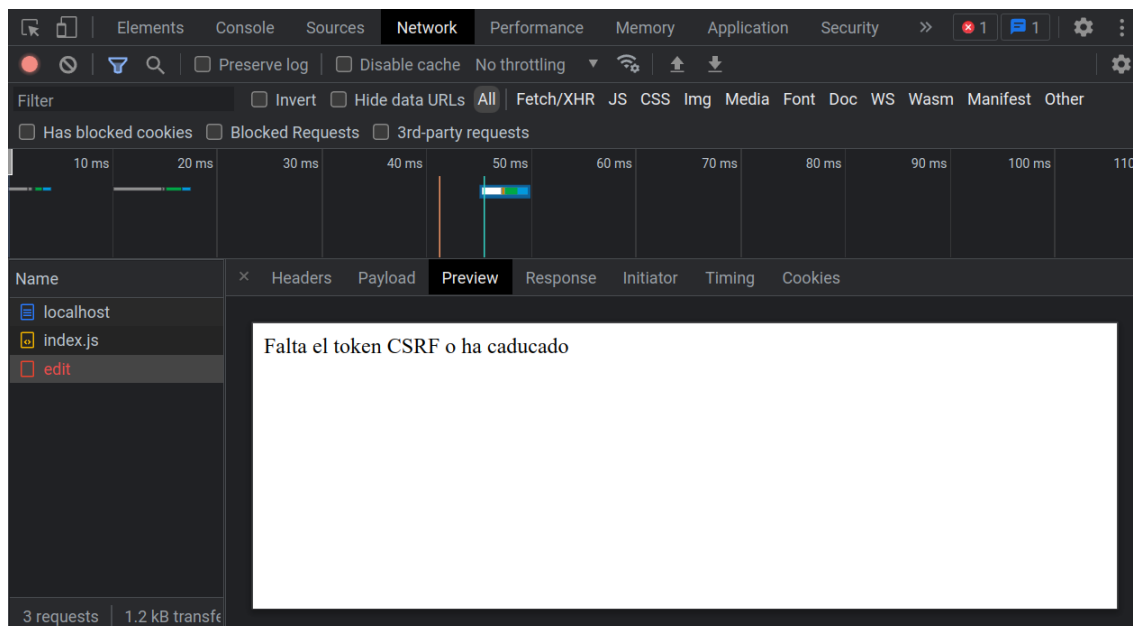
A su vez, vemos que también se ha implementado un nuevo middleware, el cual se llama *csrf*. Este middleware protegerá la ruta de editar un email cuando se haga una petición de tipo POST, es decir cuando se envíe el formulario, donde el servidor verificará si el formulario cuenta con un token CSRF, si este es válido y/o si aún sigue disponible. En caso falle en cualquiera de estas tres comprobaciones, se le retornará un error al usuario que llevó a cabo la petición y no efectuará el cambio de email.

Teniendo todo esto en cuenta, ahora el usuario está seguro en el sitio web del atacante, ya que la petición HTTP interna que tenía ahora no funcionará. Esto se aprecia revisando las cabeceras de red del sitio web del atacante cuando ingresa el usuario autenticado en el servidor objetivo.



**Figura 15. Cabecera HTTP de ataque CSRF fallido.**

Vemos que se retorna el error *422 Unprocessable Entity*, el cual indica que, a diferencia del error *400 Bad Request* que se vio previamente, no se tuvieron errores por parte del usuario, pero por algún motivo no se pudieron procesar lo solicitado en la petición HTTP. Este mencionado motivo es la falta del token CSRF, ya que cuando el sitio web atacante envía el formulario para cambiar el email mediante la petición HTTP vista, primero se pasará por el middleware visto en la **Figura 13**, se verá que el formulario que se quiere procesar no cuenta con un token CSRF, por lo que retornará el mencionado error. Del mismo modo, si revisamos la vista previa de la solicitud, veremos que, tal y como se acaba de mencionar, se indica que falta el token CSRF o que este ha caducado.



**Figura 16. Vista previa de token CSRF faltante o caducado.**

Este error se debe a que, tal y como se vio en el formulario del atacante, si bien se indica que se apunta al sitio web objetivo con el nuevo email ya mencionado, este no cuenta con el token CSRF que se genera únicamente para el usuario que ingresa a la ruta para editar su email. Además, a pesar de que uno podría pensar que el atacante podría seguir vulnerando al servidor registrándose y entrando a la ruta para editar el email y colocar el token CSRF que se le generó para él, esto no funcionaría ya que este token fue

generado para su sesión, la sesión del atacante, no para la del usuario objetivo, por lo que se seguiría teniendo error.

## Discusión

Hemos podido observar cómo se puede ejecutar un ataque CSRF para cambiar el email de un determinado objetivo, pero este ataque no se limita solo a esto. Un atacante puede hacer que un usuario haga cualquier cosa dentro de un servidor donde se encuentre autenticado sin previa autorización del mismo. Sin embargo, esto tampoco es algo del todo sencillo ya que, para poder efectuarlo, el atacante debe conocer cómo es que funciona el backend del sitio web objetivo, específicamente debe de estudiar a su objetivo para saber cómo son las URL que se mandan para ejecutar una petición HTTP. Esto se puede llevar a cabo haciendo uso de herramientas como Burp Suite, la cual sirve para analizar y manipular tráfico HTTP o HTTPS de manera local. Con ayuda de esta herramienta, el atacante podría registrarse en el sitio web objetivo y empezar a analizar el tráfico HTTP en busca de esta vulnerabilidad, por lo que sabría exactamente qué colocar en la función vista en la **Figura 9** y hacer exitoso el ataque CSRF.

Asimismo, uno podría dudar de lo presentado en la **Figura 13** pensando que las cadenas únicas retornadas por el paquete uuid podría retornar valores repetidos, lo que podría representar un riesgo ya que los atacantes podrían probar con distintos tokens que se les generen a sus usuarios hasta dar con uno que sea el del objetivo. Sin embargo, como se mencionó, la probabilidad de que una cadena se repita en este paquete de NPM es realmente baja. Para comprobar esto, se desarrolló un pequeño algoritmo en JavaScript el cual busca valores repetidos dentro de un array. Este se presenta a continuación.

```
const tokens = [1, 1, 2, 4, 1, 6, 5, 2]

const hashmap = new Map()

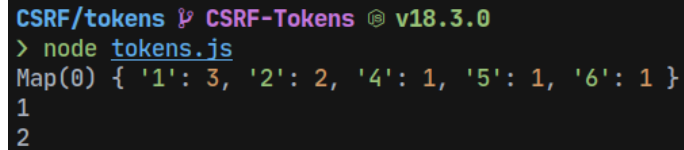
// Contar duplicados
for (let token of tokens) {
  if (hashmap[token] === undefined) {
    hashmap[token] = 1
  } else {
    hashmap[token]++
  }
}

console.log(hashmap)

// Mostrar duplicados
for (let key in hashmap) {
  if (hashmap[key] > 1) {
    console.log(key)
  }
}
```

**Figura 17. Algoritmo que busca valores repetidos en un array.**

Vemos que se tiene un array llamado *tokens* el cual almacena varios números, de los cuales los números 1 y 2 se repiten. Luego de esto, se recorre el arreglo y se genera un hashmap contando cada vez que se repiten los elementos del arreglo de tokens, donde la llave será el valor del arreglo y su valor la cantidad de veces que se repite. Por último, se recorre el hashmap y, si uno de los valores de alguna llave es mayor a 1, se retorna ya que es un elemento que se ha repetido en el arreglo. Por lo tanto, si corremos el algoritmo, se nos retornará estos dos números ya que, como se acaba de mencionar, se repiten. Esto se comprueba a continuación.



```

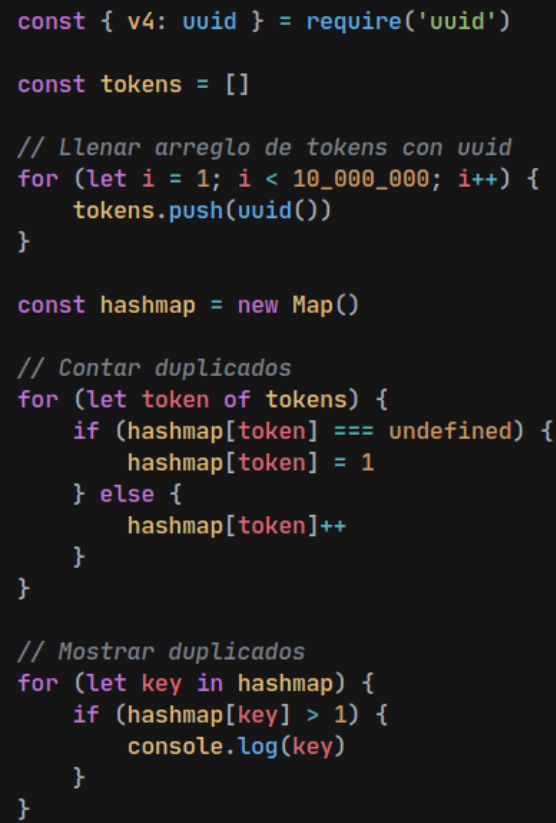
CSRF/tokens  CSRF-Tokens  v18.3.0
> node tokens.js
Map(0) { '1': 3, '2': 2, '4': 1, '5': 1, '6': 1 }
1
2

```

**Figura 18. Prueba del correcto funcionamiento del algoritmo para encontrar elementos repetidos de un array.**

Revisando el hashmap que se generó, vemos que, efectivamente, contiene como llaves a los valores de cada elemento del arreglo y, como valores, las veces que se repiten en el mismo. Por lo tanto, se verifica que el algoritmo funciona correctamente.

Ahora, usemos este algoritmo para generar un total de 10 millones de tokens usando el paquete uuid de NPM, y luego compare cada uno de estos para ver si alguna de las cadenas generadas se repite. Para esto, el algoritmo presentado sufrirá un leve cambio, el cual se muestra a continuación.



```

const { v4: uuid } = require('uuid')

const tokens = []

// Llenar arreglo de tokens con uuid
for (let i = 1; i < 10_000_000; i++) {
  tokens.push(uuid())
}

const hashmap = new Map()

// Contar duplicados
for (let token of tokens) {
  if (hashmap[token] === undefined) {
    hashmap[token] = 1
  } else {
    hashmap[token]++
  }
}

// Mostrar duplicados
for (let key in hashmap) {
  if (hashmap[key] > 1) {
    console.log(key)
  }
}

```

**Figura 19. Algoritmo que busca elementos repetidos en un array de tokens generados con el paquete uuid.**

Vemos que se ha añadido un bucle donde al arreglo de tokens se le está agregando valores de el paquete uuid 10 millones de veces. Del mismo modo, se ha eliminado el mostrar por pantalla el hashmap generado ya que este sería demasiado grande. Este algoritmo será ejecutado un total de 10 veces para comprobar que, efectivamente, las probabilidades que se genere una cadena repetida con el paquete uuid son sumamente bajas. Para agilizar este proceso, se elaboró un fish script, que es parecido a un bash script pero usando la shell de fish, el cual se presenta a continuación.

```
#!/bin/fish

echo -----

for i in (seq 10)
  echo "Iteracion $i"
  node tokens.js
  echo -----
end
```

**Figura 20. Fish script para ejecutar el algoritmo 10 veces.**

Teniendo todo esto en cuenta, pasemos a revisar los resultados que se obtuvieron al ejecutar el mencionado fish script. Estos se presentan a continuación.

```
CSRF/tokens  CSRF-Tokens  v18.3.0
> fish script.fish
-----
Iteracion 1
-----
Iteracion 2
-----
Iteracion 3
-----
Iteracion 4
-----
Iteracion 5
-----
Iteracion 6
-----
Iteracion 7
-----
Iteracion 8
-----
Iteracion 9
-----
Iteracion 10
-----
```

**Figura 21. Resultado del fish script.**

Como se puede observar, en ninguna de las iteraciones se retornaron valores repetidos del paquete uuid, lo que nos demuestra que, efectivamente, las probabilidades de obtener valores repetidos son sumamente bajas, o inclusive nulas debido a los  $16^{32}$  posibles valores que puede llegar a retornar (NPM, 2020).

## Conclusiones

A lo largo de la presente investigación, se ha presentado unas cuantas de las vulnerabilidades web más comunes en la actualidad, como Broken Access Control, inyecciones, malas configuraciones de seguridad, Server-Side Request Forgery, entre otros (OWASP, 2021). Se puso un enfoque especial en el Cross-Site Request Forgery, el cual se asimila bastante al Server-Side Request Forgery, y que, a pesar de tener ya sus años de antigüedad, sigue encontrándose en aplicaciones web actuales y pueden llegar a causar consecuencias bastante graves dependiendo del nivel de conocimiento del atacante (Cheah & Selvarajah, 2021).

Además, se desarrolló una implementación donde se simuló cómo es que un usuario malintencionado podría aprovecharse de una ruta sensible de un determinado sitio web para hacer que un usuario objetivo que se encuentra autenticado en el mismo lleve a cabo una petición sin su consentimiento y, a su vez, sin que tenga conocimiento de esto. De igual modo, se presentó una de las maneras posibles para evitar este

ataque, el cual fue mediante la implementación de tokens únicos para cada sesión de cada usuario y un middleware que proteja a la ruta sensible cuando se haga un POST. Todo esto con el objetivo de que para que un usuario lleve a cabo una operación crítica, darle la seguridad de que el único que podrá hacer esto es él.

No obstante, la manera en la que se generaron los tokens CSRF en la presente investigación se hizo para poder analizar a detalle cómo es que se lleva a cabo el proceso ya que, como se mencionó anteriormente, los frameworks actuales ya incluyen maneras sencillas para lidiar con esta vulnerabilidad. Por ejemplo, en el caso del framework utilizado en la presente investigación, Express, existe una librería llamada *csrf* que, instalándola en el servidor, importándola y haciendo una pequeña configuración, permite generar tokens CSRF en las sesiones de los usuarios y validarlos mediante un middleware (Express, 2017). Además, hay frameworks que vienen con funciones incluidas para prevenir ataques CSRF, tal es el caso de Laravel, donde no hace falta instalar ni configurar nada, sino que cuenta con una función específica para prevenir ataques CSRF y una directiva para tener un código más limpio y colocar simplemente *@csrf* (Laravel, 2022) en lugar el campo escondido con el token CSRF visto en la **Figura 14**.

Por este motivo, se puede concluir que el Cross-Site Request Forgery es un ataque ya bastante conocido debido a su antigüedad, y que, debido a que son muchos los frameworks que ofrecen soporte nativo a este, no genera muchos problemas a los desarrolladores al momento de colocar controles de seguridad en las aplicaciones web. Sin embargo, esto no indica que no se debe de excluir en el desarrollo de software, sino que es necesario estudiarlo para conocer a detalle las maneras que tiene un atacante para vulnerar una ruta que lleve a cabo operaciones críticas. De nada sirve tener a mano distintas medidas de seguridad por parte de diversos frameworks de desarrollo web si es que no conocemos las amenazas a las que están expuestas las aplicaciones web elaboradas, y esto no es únicamente con el CSRF, sino con todas las amenazas que existen. Es necesario siempre estar al tanto de las nuevas amenazas que surjan sin dejar de lado las que ya llevan un tiempo existiendo, ya que sin importar la antigüedad o el nivel de investigación que se le haya dado a una de estas, siempre pueden causar graves consecuencias ante los usuarios.

## Referencias

- Al-talak, K., & Abbass, O. (2021). Detecting Server-Side Request Forgery (SSRF) Attack by using Deep Learning Techniques. *International Journal of Advanced Computer Science and Applications*, 12(12). <https://doi.org/10.14569/ijacsa.2021.0121230>
- Armstrong, M. (2021, agosto 6). *How many websites are there?* Statista. <https://www.statista.com/chart/19058/number-of-websites-online/>
- Calzavara, S., Conti, M., Focardi, R., Rabitti, A., & Tolomei, G. (2020). Machine Learning for Web Vulnerability Detection: The Case of Cross-Site Request Forgery. *IEEE Security and Privacy*. <https://doi.org/10.1109/msec.2019.2961649>
- Cheah, S. C., & Selvarajah, V. (2021). A Review of Common Web Application Breaching Techniques (SQLi, XSS, CSRF). *Atlantis Highlights in Computer Sciences*. <https://doi.org/10.2991/ahis.k.210913.068>
- Express. (2017). *Express csrf middleware*. Express. <http://expressjs.com/en/resources/middleware/csrf.html>
- He, Y., Camacho, R. S., Soygazi, H., & Luo, C. (2021). Attacking and defence pathways for Intelligent Medical Diagnosis System (IMDS). *International Journal of Medical Informatics*, 148(104415). <https://doi.org/10.1016/j.ijmedinf.2021.104415>
- Johnson, J. (2021, diciembre 22). *Number of internet users worldwide from 2005 to 2021*. Statista. <https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/>
- Laravel. (2022). *CSRF Protection*. Laravel. <https://laravel.com/docs/9.x/csrf>
- Loureiro, S. (2021). Security misconfigurations and how to prevent them. *Network Security*, (5), 13-16. [https://doi.org/10.1016/S1353-4858\(21\)00053-2](https://doi.org/10.1016/S1353-4858(21)00053-2)
- OWASP. (2021). *OWASP Top Ten Web Application Security Risks*. OWASP Foundation. <https://owasp.org/www-project-top-ten/>
- Pellegrino, G., Johns, M., Koch, S., Backes, M., & Rossow, C. (2017). Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. <https://doi.org/10.1145/3133956.3133959>
- Stasinopoulos, A., Ntantogian, C., & Xenakis, C. (2018). Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications. *International Journal of Information Security*, 18(1), 49-72. <https://doi.org/10.1007/s10207-018-0399-z>



## **Anexos**

### **Anexo 1. Repositorio del código fuente**

El código fuente se encuentra en el siguiente repositorio:

<https://www.github.com/JCamposx/CSRF/>

Las instrucciones para hacer uso de lo que ha desarrollado se encuentran detalladas en el mismo.