

Informe Laboratorio 4

Sección 1

Alumno Kerksen Barros
e-mail: kerssen.barros@mail.udp.cl

Junio de 2024

Índice

1. Descripción de actividades	2
2. Desarrollo (Parte 1)	4
2.1. Detecta el cifrado utilizado por el informante	4
2.2. Logra que el script solo se gatille en el sitio usado por el informante	4
2.3. Define función que obtiene automáticamente el password del documento . . .	5
2.4. Muestra la llave por consola	5
3. Desarrollo (Parte 2)	6
3.1. Reconoce automáticamente la cantidad de mensajes cifrados	6
3.2. Muestra la cantidad de mensajes por consola	6
4. Desarrollo (Parte 3)	7
4.1. Importa la librería cryptoJS	7
4.2. Utiliza SRI en la librería CryptoJS	7
4.3. Repercusiones de SRI inválido	9
4.4. Logra descifrar uno de los mensajes	10
4.5. Imprime todos los mensajes por consola	11
4.6. Muestra los mensajes en texto plano en el sitio web	11
4.7. El script logra funcionar con otro texto y otra cantidad de mensajes	15
4.8. Indica url al código .js implementado para su validación	17

1. Descripción de actividades

Para este laboratorio, deberá utilizar Tampermonkey y la librería CryptoJS (con SRI) para lograr obtener los mensajes que le está comunicando su informante. En esta ocasión, su informante fue más osado y se comunicó con usted a través de un sitio web abierto a todo el público <https://cripto.tiiny.site/>.

Sólo un ojo entrenado como el suyo logrará descifrar cuál es el algoritmo de cifrado utilizado y cuál es la contraseña utilizada para lograr obtener la información que está oculta.

1. Desarrolle un plugin para tampermonkey que permita obtener la llave para el descifrado de los mensajes ocultos en la página web. La llave debe ser impresa por la consola de su navegador al momento de cargar el sitio web. Utilizar la siguiente estructura:
 - La llave es: KEY
2. En el mismo plugin, se debe detectar el patrón que permite identificar la cantidad de mensajes cifrados. Debe imprimir por la consola la cantidad de mensajes cifrados. Utilizar la siguiente estructura: Los mensajes cifrados son: NUMBER
3. En el mismo plugin debe obtener cada mensaje cifrado y descifrarlo. Ambos mensajes deben ser informados por la consola (cifrado espacio descifrado) y además cada mensaje en texto plano debe ser impreso en la página web.

El script desarrollado debe ser capaz de obtener toda la información del sitio web (llave, cantidad de mensajes, mensajes cifrados) sin ningún valor forzado. Para verificar el correcto funcionamiento de su script se utilizará un sitio web con otro texto y una cantidad distinta de mensajes cifrados. Deberá indicar la url donde se podrá descargar su script.

Un ejemplo de lo que se debe visualizar en la consola, al ejecutar automáticamente el script, es lo siguiente:

E
 E
 I
 I
 C
 E



2. Desarrollo (Parte 1)

2.1. Detecta el cifrado utilizado por el informante

Tras leer y analizar varias veces el texto mostrado en la página web, se pudo reconocer un patrón particular por el cuál reconocer la llave pedida en este laboratorio.

El patrón se identificó mirando las letras mayúsculas del texto cada vez que se utilizaba un punto seguido. Si se escriben las letras mayúsculas en orden de aparición, se puede apreciar que forma la palabra **SEGURO** 4 veces.

2.2. Logra que el script solo se gatille en el sitio usado por el informante

Para que el script de *Tampermonkey* sólo gatille en el sitio utilizado por el informante, se especifica en el **@match** la url de la página en cuestión para que de esta forma sólo se active el script cuando se entre al sitio especificado.

```
// ==UserScript==
// @name      Lab4_Cripto
// @namespace  http://tampermonkey.net/
// @version   2
// @description Script del Lab4 de Cripto.
// @author    Ker
// @match     https://cripto.tiiny.site/
// @grant     none
// @require   https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.2.0/crypto-js.min.js
// ==/UserScript==
```

Figura 1: Uso de @match para activar el script en la página del informante.

2.3. Define función que obtiene automáticamente el password del documento

La función utilizada para obtener todas las mayúsculas presentes en el texto es el siguiente:

```
1 function findMayus() {  
2     var text = document.body.textContent;  
3     var mayus = text.match(/[A-Z]/g);  
4  
5     if (mayus) {  
6         var key = mayus.join('');  
7         console.log('La llave es: ' + key);  
8     }  
9 }  
10 }
```

Listing 1: Función para obtener las mayúsculas del texto.

Se obtiene el texto contenido en el documento HTML con el método **document.body.textContent** y se utiliza la expresión regular (**/[A-Z]/g**) para buscar todas las mayúsculas de manera global.

Para la concatenación se utiliza el método **.join** cada vez que se encuentre una mayúscula y así obtener la clave completa del informante.

2.4. Muestra la llave por consola

Tras ejecutar esta función con *Tampermonkey* en la página utilizada por el informante y abrir la consola del navegador, se obtiene lo siguiente:



```
La llave es: SEGUROSEGUROSEGUROSEGURO
```

Figura 2: Llave compartida por el informante a través de la página.

3. Desarrollo (Parte 2)

3.1. Reconoce automáticamente la cantidad de mensajes cifrados

Al analizar la página utilizada por el informante con el **Inspector del navegador**, se puede apreciar que cada *div* del HTML tiene una *class* = M_i , donde se infiere que M_i hace referencia a que hay un mensaje oculto en esa sección del texto, con un total de **6 mensajes**.

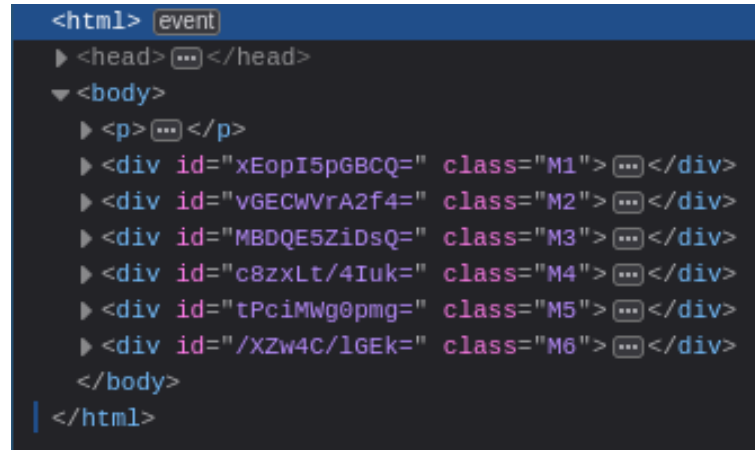


Figura 3: Detección de los mensajes M_i con el Inspector del navegador.

3.2. Muestra la cantidad de mensajes por consola

Para mostrar la cantidad de mensajes cifrados en el texto, se hizo otra función en *Tampermonkey*, la cual se muestra a continuación:

```

1 function countMsg() {
2   var msg = document.querySelectorAll('[class^="M"]');
3   console.log('Los mensajes cifrados son: ' + msg.length);
4 }

```

Listing 2: Función para obtener la cantidad de mensajes cifrados.

Se utilizó el método **document.querySelectorAll** para seleccionar todos los *div* que tengan las clases de los mensajes ocultos y guardarlos en una variable. La cantidad total se obtiene utilizando el método **.length** a la variable.

Tras ejecutar esta función, se obtiene lo siguiente en la consola de la página:

```
Los mensajes cifrados son: 6
```

Figura 4: Cantidad de Mensajes Cifrados mostrada en la página por consola.

4. Desarrollo (Parte 3)

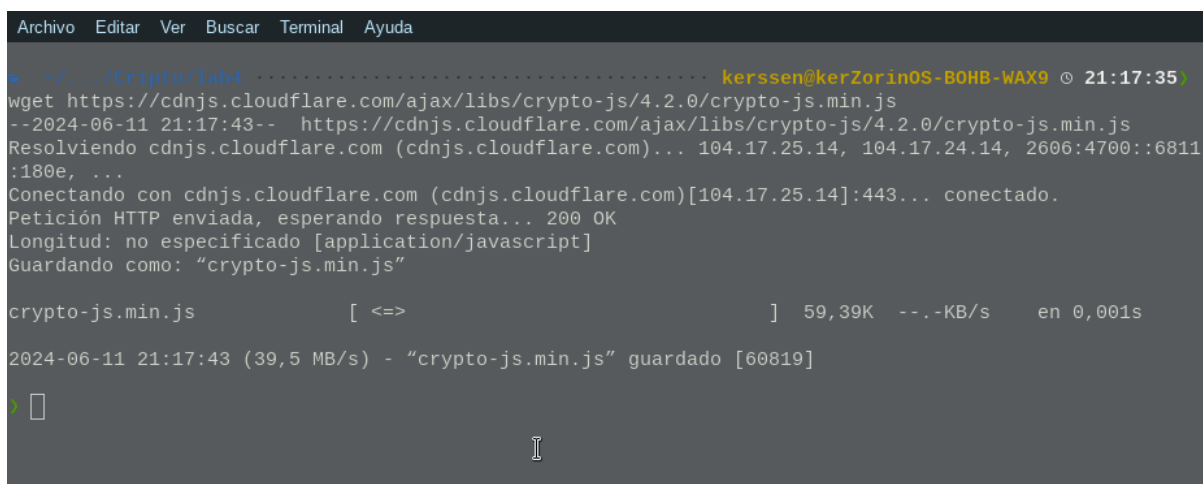
4.1. Importa la librería cryptoJS

La librería de cryptoJS se importó utilizando un `@require` para especificar la url de la librería a utilizar. La importación de se puede apreciar en 2.2, en dónde se explicó cómo se hizo uso de `@match`.

4.2. Utiliza SRI en la librería CryptoJS

El SRI (Subresource Integrity) es una característica de seguridad web que permite a los navegadores verificar que los archivos externos como scripts y estilos no han sido manipulados. En la implementación del script, se aplica esta técnica comparando el hash del archivo **crypto-js.min.js** con el de la página que se usa para importar la librería.

Primero, se descarga el archivo de **crypto-js.min.js** con el comando `wget` y luego se procede a sacar su hash correspondiente con el comando `openssl`.



```

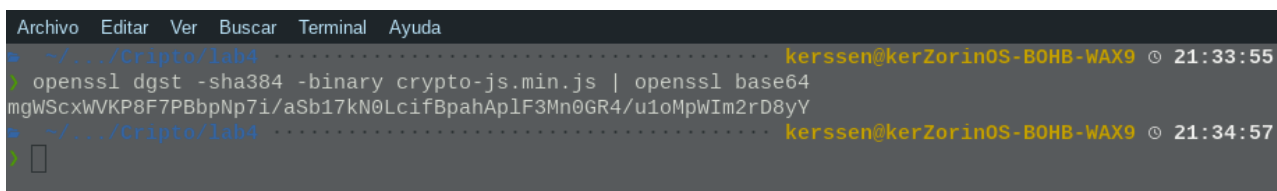
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
kerssen@kerZorin0S-B0HB-WAX9 21:17:35
$ wget https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.2.0/crypto-js.min.js
--2024-06-11 21:17:43-- https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.2.0/crypto-js.min.js
Resolviendo cdnjs.cloudflare.com (cdnjs.cloudflare.com)... 104.17.25.14, 104.17.24.14, 2606:4700::6811:180e, ...
Conectando con cdnjs.cloudflare.com (cdnjs.cloudflare.com)[104.17.25.14]:443... conectado.
Petición HTTP enviada, esperando respuesta... 200 OK
Longitud: no especificado [application/javascript]
Guardando como: "crypto-js.min.js"

crypto-js.min.js          [ <=>          ] 59,39K  --.-KB/s   en 0,001s

2024-06-11 21:17:43 (39,5 MB/s) - "crypto-js.min.js" guardado [60819]

```

Figura 5: Descarga del archivo de `crypto-js.min.js` a través de `wget`.



```

Archivo  Editar  Ver  Buscar  Terminal  Ayuda
kerssen@kerZorin0S-B0HB-WAX9 21:33:55
$ openssl dgst -sha384 -binary crypto-js.min.js | openssl base64
mgWScxwVKP8F7PBbpNp7i/aSb17kN0LcifBpahAp1F3Mn0GR4/u1oMpWIm2rD8yY
kerssen@kerZorin0S-B0HB-WAX9 21:34:57

```

Figura 6: Obtención del hash del archivo a través de `openssl`.

Con el hash obtenido, se coloca como parámetro dentro de la variable **expectedHash** para después poder compararlo con el hash que calcula el script y así verificar la autenticidad de la librería. A continuación se muestra la implementación del SRI en el script:

```
1 const expectedHash = 'mgWScxWVKP8F7PBbpNp7i/aSb17kN0LcifBpahAp1F3Mn0GR4/  
  u1oMpWIm2rD8yY'; // Replace <hash_value> with the actual hash  
2  
3 function calculateHash(buffer) {  
4   return crypto.subtle.digest('SHA-384', buffer).then(hashBuffer =>  
5   {  
6     const hashArray = Array.from(new Uint8Array(hashBuffer));  
7     const hashBase64 = btoa(String.fromCharCode.apply(null,  
8     hashArray));  
9     return hashBase64;  
10  });  
11 }  
12  
13 // Function to load and verify the script  
14 function loadAndVerifyScript(url, expectedHash) {  
15   fetch(url)  
16     .then(response => response.arrayBuffer())  
17     .then(buffer => calculateHash(buffer).then(hash => {  
18       if (hash === expectedHash) {  
19         const script = document.createElement('script');  
20         script.src = url;  
21         document.head.appendChild(script);  
22         script.onload = init;  
23       } else {  
24         console.error('Hash no coincide: Hash esperado: ' +  
25         expectedHash + ', Hash obtenido: ' + hash);  
26       }  
27     })))  
28   .catch(err => console.error('Error al cargar el script.', err  
29   ));  
30 }
```

Listing 3: Implementación del SRI en Tampermonkey

4.3. Repercusiones de SRI inválido

En caso de que los hashes no coincidan, significa que la librería ha sido modificada o comprometida, pudiendo inyectar código malicioso que afecte a la seguridad o filtración de datos.

Para evitar estos problemas en el script, la implementación mostrada en 3 previene que se ejecute el script si detecta que el hash de la página no coincide con el del archivo descargado de la librería. A continuación se muestra un ejemplo de cuando los hashes no coinciden:

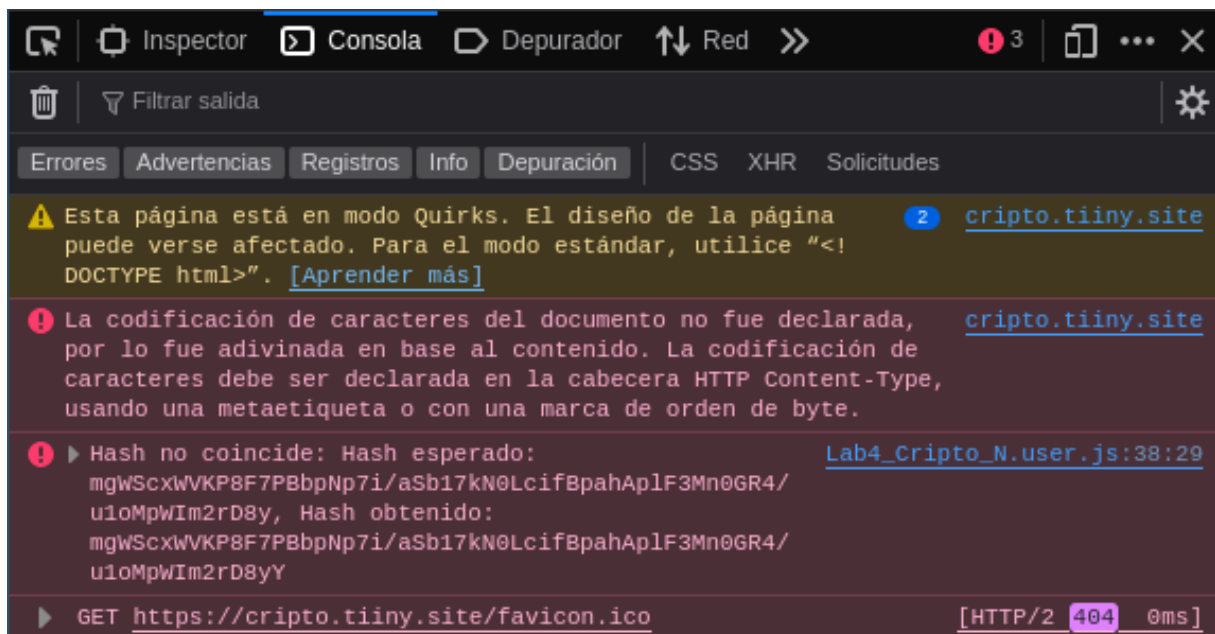


Figura 7: Mensaje que se muestra por consola cuando los hashes no coinciden.

4.4. Logra descifrar uno de los mensajes

Al momento de descifrar los mensajes, se hizo la siguiente función en *Tampermonkey*:

```
1 function decryptMsg(key) {  
2     var messages = document.querySelectorAll('[class^="M"]');  
3  
4     messages.forEach(function(message) {  
5         var text64 = message.id;  
6         var keyUtf8 = CryptoJS.enc.Utf8.parse(key);  
7  
8         var decryptedMessage = CryptoJS.TripleDES.decrypt(  
9             { ciphertext: CryptoJS.enc.Base64.parse(text64) },  
10            keyUtf8,  
11            { mode: CryptoJS.mode.ECB, padding: CryptoJS.pad.Pkcs7 }  
12        );  
13  
14        var decryptedText = decryptedMessage.toString(CryptoJS.enc.Utf8);  
15  
16        if (decryptedText) {  
17            var decryptedElement = document.createElement('div');  
18            decryptedElement.textContent = decryptedText;  
19            message.parentNode.appendChild(decryptedElement);  
20            console.log(text64 + ' ' + decryptedText);  
21        }  
22    });  
23 }
```

Listing 4: Función para descifrar los mensajes.

Se puede apreciar cómo se itera cada mensaje encontrado para obtener el *id* de cada uno y convertir la clave encontrada en 2.4 al formato **UTF-8** utilizando el método de la librería *CryptoJS .enc.Utf8.parse()*.

Con lo anterior se procede a utilizar el método **TripleDES** para descifrar los mensajes (*CryptoJS.TripleDES.decrypt()*), entregando como parámetros el **texto cifrado** (utilizando *CryptoJS.enc.Base64.parse()* para cambiarle el formato), la **key anteriormente calculada en UTF-8** y un **modo de operación** (utilizando *CryptoJS.mode.ECB* para usar el modo **ECB** y *CryptoJS.pad.Pkcs7* para usar el esquema de padding **PKCS7**).

A continuación se muestra uno de los mensajes descifrados por consola:



The image shows a dark rectangular box with a light gray border, containing the text "xEopI5pGBCQ= este" in a monospaced font. This represents the output of the decryption function shown in the previous code block.

Figura 8: Mensaje descifrado por la función *decryptMsg()*.

4.5. Imprime todos los mensajes por consola

La ejecución de la función mostrada en 4 imprime en la consola del navegador lo siguiente:

```
La llave es: SEGUROSEGUROSEGUROSEGURO
Los mensajes cifrados son: 6
xEopI5pGBCQ= este
vGECWVrA2f4= es
MBDQE5ZiDsQ= un
c8zxLt/4Iuk= mensaje
tPciMWg0pmg= de
/XZw4C/lGEk= prueba
```

Figura 9: Mensajes descifrados en la consola del navegador.

4.6. Muestra los mensajes en texto plano en el sitio web

A continuación se mostrará el código completo de *Tampermonkey*, que incluye las funciones de los pasos anteriores junto con la impresión de los mensajes en texto plano dentro de la página web:

```
1 // ==UserScript==
2 // @name      Lab4_Cripto_N
3 // @namespace  http://tampermonkey.net/
4 // @version   3
5 // @description Script del Lab4 de Cripto.
6 // @author    Ker
7 // @match     https://cripto.tiiny.site/
8 // @grant     none
9 // @require   https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.2.0/
10 //           crypto-js.min.js
11 // ==/UserScript==
12
13 (function() {
14     'use strict';
15
16     // SRI
17     const expectedHash = 'mgWScxWVKP8F7PBbpNp7i/
18     aSb17kN0LcifBpahAplF3Mn0GR4/u1oMpWIm2rD8yY'; // Replace <hash_value>
19     with the actual hash
```

```

17
18     function calculateHash(buffer) {
19         return crypto.subtle.digest('SHA-384', buffer).then(hashBuffer =>
20     {
21         const hashArray = Array.from(new Uint8Array(hashBuffer));
22         const hashBase64 = btoa(String.fromCharCode.apply(null,
23     hashArray));
24         return hashBase64;
25     });
26 }
27
28 // SRI
29 function loadAndVerifyScript(url, expectedHash) {
30     fetch(url)
31     .then(response => response.arrayBuffer())
32     .then(buffer => calculateHash(buffer).then(hash => {
33         if (hash === expectedHash) {
34             const script = document.createElement('script');
35             script.src = url;
36             document.head.appendChild(script);
37             script.onload = init;
38         } else {
39             console.error('Hash no coincide: Hash esperado: ' +
40     expectedHash + ', Hash obtenido: ' + hash);
41         }
42     })))
43     .catch(err => console.error('Error al cargar el script.', err
44 ));
45 }
46
47 function init() {
48     // Paso 1
49     function findMayus() {
50         var text = document.body.textContent;
51         var mayus = text.match(/[A-Z]/g);
52
53         if (mayus) {
54             var key = mayus.join('');
55             console.log('La llave es: ' + key);
56
57             // Ejecuci n para el Paso 3
58             countMsg();
59             decryptMsg(key);
60         }
61     }
62
63     // Paso 2
64     function countMsg() {
65         var msg = document.querySelectorAll('[class^="M"]');
66         console.log('Los mensajes cifrados son: ' + msg.length);
67     }

```

```

65 // Paso 3
66 function decryptMsg(key) {
67     var messages = document.querySelectorAll('[class^="M"]');
68
69     messages.forEach(function(message) {
70         var text64 = message.id;
71         var keyUtf8 = CryptoJS.enc.Utf8.parse(key);
72
73         var decryptedMessage = CryptoJS.TripleDES.decrypt(
74             { ciphertext: CryptoJS.enc.Base64.parse(text64) },
75             keyUtf8,
76             { mode: CryptoJS.mode.ECB, padding: CryptoJS.pad.Pkcs7
77         });
78
79         var decryptedText = decryptedMessage.toString(CryptoJS.enc
80 .Utf8);
81
82         if (decryptedText) {
83             var decryptedElement = document.createElement('div');
84             decryptedElement.textContent = decryptedText;
85             message.parentNode.appendChild(decryptedElement);
86             console.log(text64 + ' ' + decryptedText);
87         }
88     });
89
90     window.addEventListener('load', findMayus);
91 }
92
93 // Verificar con SRI
94 loadAndVerifyScript('https://cdnjs.cloudflare.com/ajax/libs/crypto-js
95 /4.2.0/crypto-js.min.js', expectedHash);
96 }());

```

Listing 5: Código completo Tampermonkey.

Se utilizó un condicional *if* para crear un *div* y colocar el mensaje en texto plano en caso de existir un mensaje descifrado. El resultado tras ejecutar el script es el siguiente:

criptoanalistas incluyen evaluar, analizar y localizar las debilidades en los sistemas y algoritmos de seguridad criptográfica. Sin el conocimiento de información secreta, el criptoanálisis se dedica al estudio de sistemas criptográficos con el fin de encontrar debilidades en los sistemas y romper su seguridad. El criptoanálisis es un componente importante del proceso de creación de criptosistemas seguros. Gracias al criptoanálisis, podemos comprender los criptosistemas y mejorarlos identificando los puntos débiles. Un criptoanalista puede ayudarnos a trabajar en el algoritmo para crear un código secreto más seguro y protegido. Resultado del criptoanálisis es la protección de la información crítica para que no sea interceptada, copiada, modificada o eliminada. Otras tareas de las que pueden ser responsables los criptoanalistas incluyen evaluar, analizar y localizar las debilidades en los sistemas y algoritmos de seguridad criptográfica.

este
es
un
mensaje
de
prueba

Figura 10: Mensajes en texto plano impresos en la página web.

4.7. El script logra funcionar con otro texto y otra cantidad de mensajes

Para comprobar que el script hecho en *Tampermonkey* funciona en otros sitios, se creó una página similar a la del informante pero con otro texto y más mensajes ocultos. A continuación se mostrará el script que se utilizó para encriptar los mensajes ocultos de la página:

```

1  const CryptoJS = require('/home/kerssen/Documentos/Cripto/lab4/crypto-js.
    min.js');
2
3  // Se buscaron todas la may sculas del nuevo texto para crear la Clave.
4  const key = "EELLNETFLDALMCPEERLTAFNA";
5
6  const messages = [
7      "hola",
8      "esta",
9      "es",
10     "una",
11     "nueva",
12     "pagina",
13     "para",
14     "probar",
15     "el",
16     "script"
17 ];
18
19 function encryptMessages(messages, key) {
20     let encryptedMessages = [];
21
22     messages.forEach(message => {
23         let encrypted = CryptoJS.TripleDES.encrypt(message, CryptoJS.enc.
            Utf8.parse(key), {
24             mode: CryptoJS.mode.ECB,
25             padding: CryptoJS.pad.Pkcs7
26         }).toString();
27         encryptedMessages.push(encrypted);
28     });
29
30     return encryptedMessages;
31 }
32
33 let encryptedMessages = encryptMessages(messages, key);
34
35 // Imprimir los mensajes cifrados en la consola para luego colocarlos en
    el HTML.
36 encryptedMessages.forEach((encryptedMessage, index) => {
37     console.log('Mensaje ${index + 1}: ${encryptedMessage}');
38 });

```

Listing 6: Script para encriptar los mensajes ocultos.

4.7 El script logra funcionar con otro texto y otra cantidad de mensajes

DESARROLLO (PARTE 3)

Luego, se utilizó el mismo *HTML* del sitio del informante para colocarle un nuevo texto y los mensajes encriptados anteriormente, subiendo la página a **tiiny.host** para hacer la comprobación (<https://prueba214.tiiny.site/>).

A continuación se mostrar el funcionamiento del script en esta página (se recuerda cambiar el **@match** por la url a probar para que lo detecte *Tampermonkey*):

una pausa. La plaza central estaba llena de niños jugando y adultos conversando animadamente. Todos parecían disfrutar de la vida al máximo. Aquel día, el atardecer nos sorprendió mientras admirábamos un paisaje increíble desde lo alto de una colina. Fue un momento mágico, lleno de paz y serenidad. Nos sentimos agradecidos por tener la oportunidad de vivir aquella experiencia única. Al día siguiente, continuamos nuestro camino hacia nuestro destino final.

hola
esta
es
una
nueva
pagina
para
probar
el
script

Figura 11: Página de prueba con los mensajes descifrados.



Figura 12: Consola de la página de prueba.

4.8. Indica url al código .js implementado para su validación

Todos los códigos implementados estarán en el **Github** de este laboratorio, pero de igual forma, se subió el archivo .js de *Tampermonkey* a **Mediafire** para su descarga:

<https://www.mediafire.com/file/ggqsxch139vck0v/tamperMonkey.js/file>

Conclusiones y comentarios

Se puede concluir que se finalizó la experiencia de laboratorio de manera exitosa debido a que se lograron cumplir los objetivos planteados del mismo, descubriendo la llave del informante en su página web, contando el número de mensajes cifrados y descifrando cada uno de ellos.

Se logró aprender el funcionamiento de *Tampermonkey* para modificar páginas e implementar funcionalidades externas para estas mismas. El uso de la librería *CryptoJS* aportó bastante a la comprensión de los algoritmo de cifrado y descifrado utilizados en la práctica de este laboratorio, logrando complementar bastante bien los conocimientos obtenidos en la cátedra.

A continuación se explicarán las problemáticas que se dieron en la elaboración del laboratorio:

1. En un inicio, fue difícil encontrar el patrón que utilizaba el informante para obtener la clave debido a que la página mostraba caracteres raros, por lo que se pensó que el patrón podría estar ahí. La solución llegó cuando el profesor del laboratorio nos informó que esos caracteres estaban así debido a un mal uso del tipo de codificación empleado en la página, haciendo que ya no se le tomara importancia a eso y se empezará a buscar por otra parte.
2. Al momento de implementar la búsqueda de los mensajes cifrados, no se tomó en cuenta que la cantidad de esto podría variar, haciendo que sólo se programara la lógica de encontrar 6 clases M_i . La solución fue implementar otro tipo de lógica la cual buscaba la etiqueta `class="M"` y así encontrar más mensajes en caso de existir.
3. Cuando se quiso implementar SRI en el script de *Tampermonkey*, el hash del archivo descargado *crypto-js.min.js* no coincidía con el de la librería importada en el script. Esto fue porque al momento de sacar el hash del archivo con *openssl*, se estaba ocupando la flag `-A` que alteraba el hash resultante, por lo que la solución fue no utilizar esa flag.
4. Por último, se dificultó bastante crear la página de prueba para comprobar el script debido a que primero se estaba intentando implementar la lógica del cifrado dentro del *HTML* utilizando la etiqueta *script*. Por alguna razón, este método cifraba mal los mensajes, por lo que se optó por crear un script aparte para cifrar los mensajes y luego colocar los resultados en el *HTML*.

Repositorio de Github: https://github.com/KerssenB/Lab4_cripto_KB