Introducción

En el siguiente informe se detallará la implementación de Python y Javascript del cipher Camelia, para la tarea de Cifrado Simétrico. Camelia es un cifrado de bloque de cifrado simétrico, con tamaño de bloque de 128 y tamaño de clave de 128, 192 o 256 bits. El modo de operación más común para este tipo de cifrado es CBC, pero el que se usará para este trabajo será EBC, junto con el tamaño de clave de 256 bits.

Implementación en Python

En Python la librería que se utilizó para poder realizar el proceso de cifrado se llama "Python Camellia" (https://pypi.org/project/python-camellia/), el cual entrega una posibilidad variada de modos de operación por unidad de bloque para elegir. Sin embargo, esta librería falla al no tener una función de padding para los textos/llaves, que no se entreguen en múltiplos de 16 bytes. Para solucionar esto, se tuvo que agregar la función Pkcs (en específico, Pkcs#5) de la librería "PyPadding" (https://pypi.org/project/PyPadding/).

```
import camellia
from pypadding import pkcs
import binascii, os
from base64 import b64encode

texto=bytes(input("Introducir texto claro:"), 'utf-8')
padding=pkcs.Encoder(camellia.block_size)
encoder=padding.encode(texto)
llave=bytes(input("Introducir llave:"), 'utf-8')
iv=os.urandom(16)
rounds=int(input("Introducir numero de rondas:"))
```

Figura 1: Primera parte del código Python.

En primera instancia, el código pide que se introduzca el texto claro a cifrar y lo convierte en bytes, para después generar el padding correspondiente en caso de que no se cumpla la regla de los 16 bytes. Luego de esto, se le concatena el padding a la variable. En el caso de la llave hay que entregar una llave de 128, 192 o 256 bits, en el caso de prueba se usó 00112233445566778899aabbccddeeff. Para generar el IV se genera un número random que cumpla la regla de los 16 bytes. Y finalmente, se pide el número de rondas en el que se ejecutará el cifrado Camellia.

Cabe destacar que Camellia por si solo 24 rondas al cifrarse, en el caso específico de 256 bits. Por lo tanto, el número de rondas que se verá ampliado por las rondas que ya se están generando en cada vuelta.

```
modo=camellia.MODE_ECB
cypher=camellia.CamelliaCipher(key=llave, iv=iv, mode=modo)
encriptado=cypher.encrypt(encoder)
for x in range(rounds-1):
    encriptadoRounds=cypher.encrypt(encoder)
encriptado64=b64encode(encriptado).decode('utf-8')
llave64=b64encode(llave).decode('utf-8')
iv64=b64encode(iv).decode('utf-8')
```

Figura 2: Segunda parte del código Python.

Lo que sigue en el código es el proceso de cifrado como tal. Se escoge el modo de operación ECB (Electronic Code Book) y se genera un cifrador en la variable llamada *cypher* que contiene las variables **llave**, **iv** y **modo**.

Luego este cifrador genera el proceso llamado "encrypt" por la cantidad de rondas que se haya introducido anteriormente.

Al hacer esto se genera otro problema. El texto está cifrado, pero si uno trata de revisar los resultados el programa te los muestra en cadenas de bits. Por lo tanto, se importó b64encode de la librería base64, para traducir los resultados y facilitar el cálculo que hará posteriormente el script de TamperMonkey. Estas variables son almacenadas en **encriptado64**, **llave64** e **iv64**.

```
html=["<p>Este sitio contiene un mensaje secreto</p>",
    '<div class="llave" id="'+llave64+'"></div>'
    '<div class="iv" id="'+iv64+'"></div>',
    '<div class="msg" id="'+encriptado64+'"></div>']

file = open("index.html","w")
for x in html:
    file.write(x + '\n')
file.close()
```

Figura 3: Tercera parte del código Python.

En las últimas líneas del código, se genera una tupla con las líneas html que albergarán el código que no estará a simple vista dentro de la página y se escribirá en el archivo index.html.

Implementación en Javascript

Para la implementación en Javascript, se tuvo que hacer un userscript compatible con TamperMonkey. En la búsqueda de una librería adecuada para poder generar el proceso de descifrado, es notorio que Camellia muy poca compatibilidad con algoritmos para cifrar o descifrar en este lenguaje. La única librería que explícitamente es compatible con Camellia es Cypher de Nodejs, pero al ser un entorno de desarrollo backend, no era compatible con los userscripts. Se intentó implementar la librería "f4st-crypt" (https://github.com/F4stHosting/F4st_Crypt/blob/master/index.js), pero el intento no fue exitoso.

```javascript
// ==UserScript==
// @name         Descifrador Camellia
// @namespace    http://github.com/benjaminignc
// @version      0.9.3
// @description  Cifrado Simetrico
// @author       Benjamín Contreras
// @match        https://benjaminignc.github.io/Cripto-Tarea3/index.html
// @grant        none
// @updateURL    https://raw.githubusercontent.com/benjaminignc/Cripto-Tarea3/index/descifradocamellia.user.js
// @downloadURL  https://raw.githubusercontent.com/benjaminignc/Cripto-Tarea3/index/descifradocamellia.user.js
// @require      https://cdn.jsdelivr.net/npm/buffer@5.6.0/index.min.js
// @require      https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.0.0/crypto-js.js
// @require      https://cdn.jsdelivr.net/npm/base64-js@1.3.1/index.js
// @require      https://cdn.jsdelivr.net/npm/ieee754@1.1.13/index.js
// ==/UserScript==

(function() {
    'use strict';
    var mensaje=document.getElementsByClassName('msg')[0].id;
    var iv=document.getElementsByClassName('iv')[0].id;
    var llave=document.getElementsByClassName('llave')[0].id;

    function decrypt(text, passwd, algorithm, ivt) {
        if (typeof algorithm === 'undefined') { algorithm = 'aes-256-cbc'; }
        let iv = Buffer.from(ivt, "hex");
        let key = crypto.createHash('sha256').update(passwd).digest('base64').substr(0, 32);
        let encryptedText = Buffer.from(text, 'hex');
        let decipher = crypto.createDecipheriv(algorithm, Buffer.from(key), iv);
        let decrypted = decipher.update(encryptedText);
        decrypted = Buffer.concat([decrypted, decipher.final()]);
        return decrypted.toString();
       }

    var mensajeclaro=decrypt(mensaje,llave,'camellia-256-ebc',iv);
    alert(mensajeclaro);

})();
```

Figura 3: Implementación fallida en Javascript.

A pesar de haber fallado en intentar descifrar el mensaje, el cual era el objetivo principal de la tarea, se pudo cumplir ciertos objetivos pequeños como implementar un Userscript en TamperMonkey, proveniente de Github, el cual se actualiza automáticamente después de un commit + push del archivo Javascript. También se obtienen los elementos de los divs provenientes del archivo html.

El Github de la tarea se puede observar en el siguiente link: https://github.com/benjaminignc/Cripto-Tarea3/tree/index