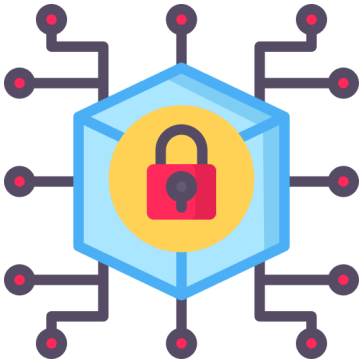
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS, 2025-1

Criptografía y Seguridad



PRÁCTICA 4:

CIFRADOS CLÁSICOS

PROFESORA:

Anayanzi Delia Martínez Hernández

INTEGRANTES CYBERWIZARDS:

Fernández Blancas Melissa Lizbeth (319281778)

López Prado Emiliano (319205806)

Sánchez Salmerón Ethan Damian (319122323)

Introducción

Para poder proteger la información de posibles riesgos de seguridad es importante conocer el cifrado y manipulación de datos.

En esta práctica nos centramos en el desarrollo de codificadores y decodificadores en base 64 y el uso de magic bytes para cifrar y descifrar archivos. Además los objetivos son conocer el funcionamiento de los cifrados clásicos (como el cifrado de César, cifrado diezmado, cifrado afín y la codificación base 64), así como las firmas de archivos y la manipulación de bytes.

Los Magic bytes (firmas de archivos) son secuencias de bytes al inicio de un archivo que permite (principalmente) a los sistemas Unix identificar el tipo de archivo con el que se está tratando. El comprender la manera en que se pueden manipular los bytes de un archivo para cifrar y descifrar los datos es de mucha importancia para poder interpretar éstos correctamente.

La codificación en base 64 se usa para convertir datos binarios a una representación textual (ASCII) de manera que los datos binarios no puedan ser tan fácilmente corrompidos o malinterpretados durante su transmisión.

Para esta práctica se realizaron programas de codificación, decodificación, cifrado y descifrado en Python para aprovechar las funciones predefinidas de este lenguaje para leer bytes. A través de ésta también se identificarán las situaciones en las que es más conveniente usar cada tipo de cifrado.

Desarrollo

A tu compa el menos hacker, que le encanta andar descargando videojuegos de internet vía Bit Torrent fue descuidado y por accidente descargó un *virus* que se activó al momento de ejecutar el *setup.exe*, haciendo que todos los archivos en su carpeta de *Escritorio* fueran cifrados, y como tú estudias en la FC, te ha pedido ayuda para restaurar sus archivos como estaban.

Lo único que sabe tu amigo es que los achivos están cifrados con un *cifrado simétrico*, es decir, es posible que se rompa el cifrado sin conocer la llave original. También recuerda que tenía un *video*, una *canción*, un *PDF* y una *imagen* (mp4, mp3, pdf, png).

Para realizar este descifrado lo primero que hicimos fue un programa (DecryptProg.py) para descifrar utilizando las firmas de cada tipo de archivo.

```
file_signatures1 = {'png': [0x89, 0x50, 0x4e, 0x47, 0x0d, 0x0a, 0x1a, 0x0a],
                    'pdf': [0x25, 0x50, 0x44, 0x46, 0x2D],
                    'mp3': [0x49, 0x44, 0x33],
                    'mp4': [0x66, 0x74, 0x79, 0x70],
                    'jpg': [0xFF, 0xD8, 0xFF, 0xE0],
                    'jpeg': [0xFF, 0xD8, 0xFF, 0xE0]}
```

Lo primero que hicimos fue leer los datos del archivo dado y convertirlos a enteros. De la misma manera, cada secuencia de magic bytes fue convertida a enteros.

```
file = open(file_path, "rb")
data = file.read()
numbers = [x for x in data] # bytes as integers
hexadecimal = [hex(x) for x in numbers ] # bytes as hexadecimal

#Transform the hexadecimal numbers to decimal
decimal_file = [int(x, 16) for x in hexadecimal]
print("Decimal file: ", decimal_file[:10])

#Transform the file signatures1 to decimal
decimal_signatures = {}
for key in file_signatures1:
    decimal_signatures[key] = [int(x) for x in file_signatures1[key]]
```

Posteriormente, para cada tipo de archivo se hizo un sistema de ecuaciones módulo 256 (pues cada byte puede representar 256 valores), de manera que:
Sean a (first_file) y b (second_file) los primeros bytes significativos del programa a desencriptar (para el mp4 se tuvo en cuenta que la firma comienza a partir del quinto byte).
Sean c (first_sign) y d (second_sign) los primeros bytes de la firma del archivo que estamos revisando.

```
first_file = decimal_file[0] # First byte of the file
second_file = decimal_file[1] # Second byte of the file
first_sign = decimal_signatures[key][0]  # First byte of the signature
second_sign = decimal_signatures[key][1]  # Second byte of the signature
```

Entonces el sistema de ecuaciones planteado fue:

$ax + y \equiv c \bmod 256$ y $bx + y \equiv d \bmod 256$

de donde obtenemos que:

$ax - bx + d \equiv c \bmod 256$ por lo que $x \equiv (c - d)(a - b)^{-1} \bmod 256$ y $y \equiv d - b((c - d)(a - b)^{-1}) \bmod 256$.

```
alpha_mult = (first_file - second_file) % 256
right_side = (first_sign - second_sign) % 256
```

```
inverses_alpha_mult = [x for x in range(256) if (x * alpha_mult) % 256 == 1] # Get the inverses of alpha_mult
```

```
alpha = right_side * inverses_alpha_mult[0] % 256
beta = (first_sign - (first_file * alpha)) % 256
```

Esto se programó de manera que se obtuvieron los valores de x (α en nuestro programa) y y (β en nuestro programa) y posteriormente cada byte del archivo cifrado fue modificado multiplicándolo por α y sumándole β

```
decoded = []
for i,num in enumerate(decimal_file):
    decoded.append((num * alpha + beta) % 256)
```

Si los bytes correspondientes a la firma del archivo resultante coinciden con la firma del archivo con el que se está probando, entonces el archivo se escribe y se ha encontrado el archivo descifrado. De otra manera se sigue verificando con las otras firmas de archivo.

```
found = True
if key == 'mp4':
    for i in range(4, len(signature)):
        if decimal_file_signature[i] != signature[i]:
            found = False
            break
else:
    for i in range(len(signature)):
        if decimal_file_signature[i] != signature[i]:
            found = False
            break

# Write the file
if found:
    file = open(decrypted_name +"." + key, "wb")
    file.write(bytearray(decoded))
    file.close()
    break
```

Para la decodificación en base 64 del file1.lol se optó por realizar un decodificador casero.

La idea detrás es utilizar un arreglo de todos los caracteres que se pueden usar en una codificación de 64 caracteres (2^6, 6 bits), estos son:

‘ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/’

y lo que vamos hacemos es convertir todos los caracteres del archivo codificado a su representación binaria, cada caracter va a terminar representando 6 bits.

```
# Convierte cada carácter de base64 en sus 6 bits correspondientes
for char in data:
    binary_string += format(base64_chars.index(char), '06b')
```

Y después de convertir todo el archivo completo a binario, agrupamos los bits de 8 en 8 (2^8, 256 caracteres, la codificación clásica) y de esta forma ahora podemos considerar estos valores como bytes y podemos hablar de un archivo funcional.

```
# Agrupar los bits en bytes (grupos de 8)
bytes_array = []
for i in range(0, len(binary_string), 8):
    byte = binary_string[i:i+8]
    if len(byte) == 8:
        bytes_array.append(int(byte, 2))
```

Ya teniendo un arreglo de todos los bytes que conforman al archivo decodificado, simplemente podemos escribirlos en un archivo y ya tendríamos nuestro archivo final, sin embargo, se optó por realizar un paso extra para facilitar un poco su funcionamiento en sistemas Windows: Teniendo el archivo decodificado, podemos visualizar los Magic Bytes y de esta forma saber el tipo de extensión que tiene el archivo, por ahora lo que hacemos es guardar en un arreglo los magic bytes más comunes y compararlos con el archivo final, de esta forma le agregamos también su respectiva extensión y podremos visualizar el archivo final más fácilmente.

```
def detect_extension(bytes_array):
    # Toma los primeros bytes para compararlos con las firmas de archivos
    for extension, signature in file_signatures1.items():
        if bytes_array[:len(signature)] == signature:
            return '.' + extension
    return '.bin'  # Si no se detecta, guarda como archivo binario genérico
```

```
# Diccionario de "file signatures" con las extensiones correspondientes
file_signatures1 = {
    'png': [0x89, 0x50, 0x4e, 0x47, 0x0d, 0x0a, 0x1a, 0x0a],
    'pdf': [0x25, 0x50, 0x44, 0x46, 0x2D],
    'mp3': [0x49, 0x44, 0x33],
    'mp4': [0x66, 0x74, 0x79, 0x70],
    'jpg': [0xFF, 0xD8, 0xFF, 0xE0],
    'jpeg': [0xFF, 0xD8, 0xFF, 0xE0]
}
```

Preguntas

Contesta las siguientes preguntas con tus propias palabras. Serán sometidas a la prueba turing.

1. ¿Cuántos primos relativos hay en Z_{256}?

Para este ejercicio nos podemos basar en la función fi de Euler, su finalidad es contar el número de primos relativos en n (números tales que mcd(k, n) = 1) y se representa de la siguiente manera:

$$\phi(n) = n \prod_{i=1}^{m} (1 - \frac{1}{pi}) \text{ con pi factores primos de n}$$

Por lo que el cálculo es relativamente sencillo, puesto que queremos calcular $\phi(256)$, lo primero que tenemos que hacer es conseguir los factores primos que componen a 256, sin embargo, 256 es parte de las potencias del 2 (2^8), por lo que su único factor primo es el número 2, por lo tanto, efectuamos la siguiente operación:

$$256\,(1 - \frac{1}{2}) = 256\,(\frac{1}{2}) = 128$$

Por lo tanto, podemos decir que existen exactamente 128 primos relativos en Z_{256}

2. Sea $f : Z_n \rightarrow Z_n$ tal que $f(i) = k * i$ para $i \in A$, con A un alfabeto cualquiera. ¿Qué se debe cumplir para que f sea una función *biyectiva*?

Primero que nada, notemos que se nos dice arma la función como $f(i)\ tq\ i \in A$ donde A es un alfabeto cualquiera, esto nos está diciendo el dominio de la función, en pocos términos, tenemos que $n = |A|$, retomaremos esto más adelante.

para que f sea una función biyectiva debe cumplir 2 condiciones:

- Inyectiva [si f(x) = f(y) entonces x = y]
- Suprayectiva [$\forall b \in A, \exists a \in A\ tq\ f(a) = b$]

Para que el primer punto se cumpla, que sea inyectiva, vamos a ver que se requiere:

Tenemos que f(i) = k * i, por lo tanto, para que sea inyectiva se debe cumplir que si $f(i_1) = f(i_2) \rightarrow i_1 * k \equiv i_2 * k$ (mod n) $\rightarrow i_1 \equiv i_2$ (mod n)

Ya que nos encontramos usando módulos, la única forma en la que teniendo $i_1 \neq i_2$ tengamos que $i_1 * k \equiv i_2 * k$ (mod n), es que k y n tengan divisores en común, por lo tanto, requerimos que k sea primo relativo con n, es decir mcd(k, n) = 1. De esta forma se cumple la propiedad de inyectividad.

Por otro lado, la condición de suprayectividad se cumple si es que cada elemento del contra dominio tiene un elemento del dominio que lo genera, es decir si para existe una solución $k * i \equiv b$ (mod 26) para cada $b \in Z_n$.

Esto solo es posible si k tiene un inverso multiplicativo en Z_n, notemos que si se cumple la condición anterior, de que mcd(k, n) = 1, entonces también tendremos un inverso multiplicativo asegurado para k.
Por lo que podemos asegurar que para que $f : Z_n \rightarrow Z_n$ tal que $f(i) = k * i$ para $i \in A$ sea una función biyectiva se debe cumplir que k debe ser primo relativo con n, donde n es el número de elementos contenidos en el alfabeto A. También notemos que si |A| tiene un número primo de elementos entonces cualquier $k \neq 0$ cumple la condición.

3. ¿Cuántas posibles combinaciones no triviales existen para cifrar bytes con *César*, *Decimado* y *Afin*?

Para el Cifrado de Cesar, estamos realizando desplazamientos, por lo que, sin contar el desplazamiento del 0 que es trivial, en bytes, tenemos 256-1=255 posibles combinaciones para cifrar con Cesar.

Para un cifrado Dicimado, necesitamos multiplicar cada byte por un número que sea primo relativo a 256. Utilizando la fórmula de Tonient de Euler, que nos devuelve cuantos números son primos relativos a un número n, dado por:

$$\phi(n) = n \prod_{i=1}^{m} (1 - \frac{1}{pi}) \text{ con pi factores primos de n}$$

Así, decompoiendo a 256 en factores primos tenemos que

$$256 = 2^8$$

Entonces, resolviendo la fórmula

$$\phi(256) = 256(1 - \frac{1}{2}) = 128$$

Por lo tanto hay 128 posibles combinaciones para cifrar bytes con Dicimado.
Ahora, para funciones afines, sabemos que son de la forma:

$$f(x) = (ax + b) \, mod \, 256$$

Y sabemos que a tiene que ser primo relativo a 256. Y como ya vimos, hay 128 posibles combinaciones para esto. por lo que hay:

$$128 * (256 - 1) = 32640$$

Posibles combinaciones no triviales para las funciones afines

4. ¿Por qué el sistema de archivos de *UNIX*, aunque un archivo tenga una extensión diferente (o incluso no tenga), sigue reconociendo al archivo original?

   Linux puede identificar el tipo de un archivo sin la extensión correcta a través de distintos métodos:

   - *Magic Bytes:* Los magic bytes son secuencias específicas de bytes al inicio de un archivo (cada tipo de archivo tiene una secuencia específica de bytes). Basta con comparar estas secuencias de bytes al inicio del archivo para que Unix pueda saber qué tipo de archivo es.
   - *Metadatos:* Los sistemas Linux guardan metadatos que pueden dar información acerca del tipo de archivo.
   - *Líneas Shebang (#!):* para los scripts ejecutables, Linux puede buscar por la línea #!/bin/bash al inicio del archivo para saber cómo ejecutarlo.

5. ¿Por qué los archivos descifrados tienen exactamente el mismo tamaño que antes de cifrar, pero no pudimos leerlos? ¿Por qué no tuvimos que agregar/quitar nada?

   Con los tipos de cifrado que se están usando para esta práctica, el tamaño de los datos no cambia, es decir, no estamos agregando ni quitando información, solamente cambia la manera en la que se está representando esta, de manera que el archivo no se pueda visualizar correctamente a menos que se descifre.

   En el caso del file1.lol, se está utilizando base 64, en la cual se convierte de binario a ASCII, de manera que la información binaria se maneje fácilmente por sistemas diseñados para trabajar con texto. Generalmente, los textos codificados con base 64 son más grandes que el original debido al tipo de representación que se está utilizando (cada 3 bytes de datos originales se convierten a 4 bytes en base 64), pero el tamaño de los datos no cambia.

   En el caso de los archivos descifrados a través de magic bytes, el formato del archivo no cambia, pues solamente se modifican los bytes, pero no se puede leer porque el lector de archivos no los reconoce.

6. Ya que *base64* no es un cifrado, sino codificación, ¿en qué casos podemos usarlo?

La codificación base64 es realmente útil para fines de comunicación a través de internet, la transmisión de datos binarios en forma de texto es mucho más sencilla de esta manera.

Esto también facilita el guardar multimedia en forma de texto en bases de datos, en campos que solo reciben valores de texto.

También nos permite meter cierto nivel de ofuscación a cualquier dato que requiramos para que no sea tan evidente simple vista.

7. Supongamos que estuvieras en Hogwarts y tuvieras que utilizar un búho para comunicarte, ¿cuál crees que sería la mejor opción para mandar mensajes seguros a través de la lechuza?
   Ya que podría caer en manos equivocadas. ¿Cómo cifrarías tu mensaje, o cómo harías el intercambio de llaves? Puedes darte una idea con este video. Puedes ser creativo :)
   Para mandar la carta por una lechuza, usaría un hechizo de transfiguración a la carta, convirtiéndola en una pluma de la lechuza, además de poner una protección que hace que la carta no regrese a su forma original a menos que la tome el destinatario.
   Además, como refuerzo, la carta primero la haría una carta voficadora, que se destruye en caso de que no se abra por el destinatario.

Punto extra

A partir del *descifrador*, reutiliza el código para crear un *cifrador* de archivos, pasando por banderas o flujo de programa los parámetros para los valores del cifrado, incluyendo base64.

Para esta parte, se entrega un cifrador de archivos, para cifrar archivos con funciones afines, se usa una función de la forma

$$f(x) = (ax + b) \, mod \, 256$$

Donde a tiene que ser un número primo relativo a 256.

```python
def affine_encrypt_file(file_path,otput_name, a):
    if not is_relative_prime(a, 256) or not isinstance(a, int):
        raise ValueError("a tiene que ser un número, primo relativo a 256")
    b = random.randint(0, 255)
    with open(file_path, "rb") as f:
        data = f.read()

    encrypted_data = bytes([affine_encrypt(x, a, b) for x in data])

    with open(otput_name, "wb") as f:
        f.write(encrypted_data)
```

Mientras que para base64 convertimos a binario el archivo, llenar con ‘0’ si el archivo binario no es múltiplo de 6, romper en bloques de 6 bits y recodificarlos en alguno de los caracteres posibles en dicho rango añadiendo = para completar los grupos de 6.

```python
def encode_base64(data):
    BASE64_CHARS = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    # Convert the data to a binary string
    binary_str = ''.join(f'{byte:08b}' for byte in data)

    # Make the binary string a multiple of 6
    extraBits = (6 - len(binary_str) % 6)
    extraBits = extraBits%6 # If the length is already a multiple of 6.
    binary_str = binary_str + '0' * extraBits

    # Encode the binary string in Base64 by grouping 6 bits at a time (2^6)
    encoded_str = ''
    for i in range(0, len(binary_str), 6):
        six_bits = binary_str[i:i+6]
        index = int(six_bits, 2)
        encoded_str += BASE64_CHARS[index]
    if extraBits > 0:
        encoded_str += '=' * (extraBits // 2)

    return encoded_str
```

Conclusiones

En el cifrado afín, aunque su forma matemática es interesante, tiene muchas debilidades frente a otro tipo de cifrados mas fuertes como el RSA que se usa mucho actualmente, el cifrado afín es muy débil a ataques como análisis de frecuencias, por lo que consideremos que es una mala idea usarlo para intentar proteger información, para fines educativos o históricos, es importante conocerlo, apero para fines prácticos no.

Por otra parte, el cifrado en base64, aunque no sea un método de cifrado si no mas bien una codificación de texto, es usado actualmente para transmitir datos a través de sistemas que manejan texto, pero para que sea seguro mandarlos por estos canales. es necesario aplicarle un tipo de cifrado más fuerte, es interesante como muchos textos e información es codificada a base64, ya que hoy en día muchos sistemas son capaces de leerlo.

En esta práctica nos enfocamos en desarrollar y comprender los diferentes tipos de cifrados clásicos, que en este caso fueron dos, cifrados con funciones afines y cifrados con base64, asu vez, pudimos entender la forma en que se pueden desencriptar los archivos usando estos cifradores.

Además, se explora la importancia de los magic bytes para la identificación de archivos en sistemas Unix y la manipulación de bytes para cifrar y descifrar datos. A través de la implementación de programas en Python, se destaca la utilidad de estas técnicas para proteger la información que se quiera transmitir por medios inseguros, su importancia radica en salvaguardar datos que no queremos que caigan en manos equivocadas.

REFERENCIAS

- José Galaviz Casas, *Introducción a la criptología* (2010)

- *Base64 Algorithm Description*, https://medium.com/swlh/base64-encoding-algorithm-42abb929087d 3. *Magic Bytes*

- Wikipedia contributors. (2024, 30 agosto). *List of file signatures*. Wikipedia. https://en.wikipedia.org/wiki/List_of_file_signatures

- Spanning Tree, *How to Send a Secret Message*. https://www.youtube.com/watch?v=I6Unxb PFhs

- *How does Linux identify file types without extensions? And why can't Windows do so?* (s. f.). Quora. https://www.quora.com/How-does-Linux-identify-file-types-without-extensions-And-why-cant-Windows-do-so

- Pozo, S. (s. f.). *Blog con clase | Codificación Base64*. © 2000 Salvador Pozo. https://conclase.net/blog/item/base64

- Wikipedia contributors. (n.d.). *Euler's totient function*. Wikipedia. https://en.wikipedia.org/wiki/Euler%27s_totient_function

- dCode. (s. f.). *Euler's Totient*. https://www.dcode.fr/euler-totient