Cuadernos Jypiter Documentación

Autenticación de Mensajes y Funciones Hash

Seguridad Informática 2021/2022

Christian Luna Escudero

Universidad de Córdoba

Índice general

1

Encriptación Clásica y Conceptos básicos de encriptación

1.1. Introducción

En este cuadernos ***jupyter***, se pretenderá mostrar y resolver los diferentes ejercicios propuestos en la práctica 4 de Seguridad.

Las técnicas de cifrado que veremos serán aquellas basadas en la sustitución y transposición de los caracteres del mensaje. Debemos de saber que a lo largo de la historia han surgido muchos algoritmos de cifrado, los cuales han servido como base a las técnicas modernas de encriptación computacional.

1.2. Técnicas de Encriptación por sustitución

Uno de los primeros cifrados basados en la sustitución sería el cifrado **Caesar**, el cual consistiría en el desplazamiento de la cadena del alfabeto, “k” posiciones de la posición original de la letra.Provocando así que el mensaje cifrado tenga las mismas frecuencias de repetición.

El algoritmo original utilizaba una k = 3, la cual sería como podemos saber la clave secreta, para encriptar y desencriptar el mensaje.

Una vez entendido el fundamento del algoritmo, en nuestro caso utilizaremos la cadena de caracteres del alfabeto ingles, como alfabeto para encriptar, siendo necesario que nuestro algoritmo desplace los caracteres de la cadena, k veces a la derecha. Una vez realizado esto, procederemos al cambio de caracteres por su correspondiente.

Esto lo podemos ver gráficamente en la figura.

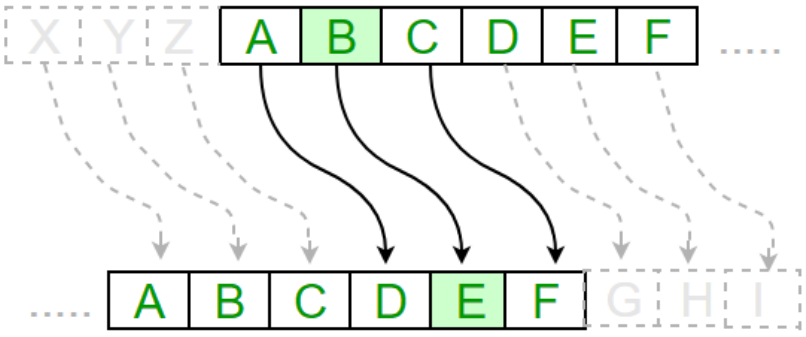

Figura 1.1: Algoritmo César para K = 3

Una vez entendido de forma teórica el funcionamiento del algoritmo **Caesar**, procederemos a realizar las diferentes líneas necesarias para implementarlo en Python.

1.2.1. Encriptación mediante Caesar

Primero deberemos de importar los módulos necesarios para el tratamiento de argumentos y para poder trabajar con cadenas de texto y usar la función para obtener el alfabeto ingles.

```
import argparse # Nota este modulo no se puede utilizar en jupyter
import string
```

Después deberemos de realizar el tratamiento de argumentos del programa, para ello utilizaremos la clase ***ArgumentParser***, proporcionada por el modulo argparse. Las siguientes instrucciones se recomiendan no ejecutar debido a que jupyter no permite el tratamiento por linea de argumentos. Estos fragmentos siguientes son explicativos únicamente, si desea utilizar solo el cuaderno Jupyter, se recomienda no ejecutarlos.

```
parser = argparse.ArgumentParser(description='Encriptar un fichero de texto mediante la
encriptacion Caesar')
```

De esta forma en la variable parser tendremos un objeto de la clase, el cual mediante su constructor hemos rellenado la descripción del algoritmo. Para agregar un argumento obligatorio utilizaremos la función add_argument, mediante la cual podremos.

```
parser.add_argument("fileIn", help="Fichero que se desea encriptar", type=str)
```

Si deseamos agregar a nuestro programa un argumento opcional, podemos utilizar la misma función indicándole.

```
parser.add_argument('-o', dest = "fileOut", type=str, metavar='fileOut', help='Especifica
 el fichero de Salida')
```

Una vez tengamos todos los argumentos deseados deberemos de parsear los argumentos. Para ello utilizaremos la función ***parse_args***, la cual lo que hará será parsear los diferentes argumentos que se le pasen al script a la hora de ejecutarlo.

```
argumentos = parser.parse_args()
```

Una vez ya parseados los argumentos, tendremos acceso a ellos a través de la variable argumentos, utilizados el "." seguido del nombre del argumento especificado. Un ejemplo de acceso sería el siguiente.

```
nombreFicheroEncriptar = argumentos.fileIn
```

Un aspecto muy importante a tener en cuenta es que los argumentos opcionales en el caso que no hayan sido suministrados, deberemos de realizar un control de errores ya que nos devuelve **None**, en el caso que accedamos al argumento sin haber sido suministrado.

Por ello deberíamos de realizar la siguiente comprobación.

En caso que no sea suministrado el nombre lo que haremos sera utilizar el mismo nombre que el de origen con la extensión enc.

```
nombreFicheroDestino = argumentos.fileOut
if nombreFicheroDestino is None:
    nombreFicheroDestino = nombreFicheroEncriptar[:nombreFicheroEncriptar.find('.')]
    nombreFicheroDestino = nombreFicheroDestino + ".enc"
```

Tras haber leído los argumentos y comprobado sus contenidos, procedemos a coger el alfabeto inglés para trabajar con el algoritmo **Caesar**. Para ello utilizaremos la clase string, la cual mediante **ascii_letters**, obtendremos el conjunto de letras permitidas.

```
cadenaAlfabeto = string.ascii_letters
print (cadenaAlfabeto)
```

Como podemos observar, en la ejecución anterior, cadenaAlfabeto posee todas las letras del alfabeto minúsculas y mayúsculas juntas. Si no deseáramos utilizar todas las letras sino solo las minúsculas podemos utilizar **ascii_lowercase** para obtener las minúsculas o **ascii_uppercase** para las mayúsculas.

```
cadenaAlfabeto = string.ascii_lowercase
print (cadenaAlfabeto)

```

```
cadenaAlfabeto = string.ascii_uppercase
print (cadenaAlfabeto)
```

Una vez obtenido el alfabeto lo que haremos será generar una lista con cada uno de los caracteres y posteriormente desplazar los elementos de la lista ”k”posiciones. Para ello en nuestro caso para facilitarnos un poco el trabajo utilizaremos la función desplazarlista creada por nosotros la cual se le pasara la lista a modificar y sus desplazamientos.

```
    def desplazarlistaDerecha(lista, desplazamiento=0):
        longitoLista = len(lista)
        for i, elemento in enumerate(lista[:]):
            lista[(i + desplazamiento) % longitoLista] = elemento
        return lista
```

Como podemos ver en la función lo que hacemos es realizar un bucle el cual recorrerá la lista, para ello utilizaremos la función enumerate la cual nos proporcionar una tupla de dos elementos, formada por el índice de la iteración actual junto con el elemento de la lista en la posición que nos encontramos. De esta forma podremos calcular realizar los desplazamientos de los caracteres.

```
    listaCaracteres = list(cadenaAlfabeto)
    print(f"{listaCaracteres}=")
    claveCaesar = 3 #Si se desea modificar la clave tocar esta variable
    listaCaracteres = desplazarlistaDerecha(listaCaracteres,claveCaesar)
    print(f"{listaCaracteres}=")
```

Una vez realizado esto ya podemos proceder a abrir el fichero que deseamos encriptar y recorrerlo realizando los cambios. Para realizar los cambios utilizaremos la función maketrans, la cual nos devuelve un mapeo de caracteres, relacionando los caracteres del alfabeto a la cadenaDesplazada.

De esta forma podremos después realizar una traducción de la linea del fichero usando este mapeo, generando así nuestra cadenaEncriptada mediante **Caesar**.

Y finalmente escribimos en el ficheroEncriptado el resultado.

```
    nombreFicheroEntrada = "hola.txt"
    nombreSalida="hola.enc"
    ficheroPlano = open(nombreFicheroEntrada, 'r')
    ficheroEncriptado = open(nombreSalida, 'w')
    cadenaEncriptar = "".join(listaCaracteres)
    try:
        # Procesamiento del fichero
        for linea in ficheroPlano:
            trantab = linea.maketrans(cadenaAlfabeto, cadenaEncriptar)
            lineaEncriptada = linea.translate(trantab)
            ficheroEncriptado.write(lineaEncriptada)

    finally:
        ficheroPlano.close()
        ficheroEncriptado.close()
```

Finalmente cerramos los ficheros y ya tendríamos los ficheros encriptados.

1.2.2. Desencriptación mediante Caesar

Para el caso de desencriptar utilizaremos muchos aspectos antes vistos. Como por ejemplo el tratamiento de argumentos.

```
    import argparse
    import string

    parser = argparse.ArgumentParser(description='Encriptar un fichero de texto mediante la
    encriptacion Caesar')

    parser.add_argument("fileIn", help="Fichero que se desea desencriptar", type=str)

    parser.add_argument('-o', dest = "fileOut", type=str, metavar='fileOut',
                        help='Especifica el fichero de salida')

    argumentos = parser.parse_args()

    nombreFicheroDesencriptar = argumentos.fileIn

    nombreFicheroDestino = argumentos.fileOut
    if nombreFicheroDestino is None:
```

```
        nombreFicheroDestino = nombreFicheroDesencriptar[:nombreFicheroDesencriptar.find('.')
    ]
        nombreFicheroDestino = nombreFicheroDestino + ".txt"
```

```
    def desplazarlistaIzquierda(lista, desplazamiento=0):
        longitoLista = len(lista)
        for i, elemento in enumerate(lista[:]):
            lista[(i - desplazamiento) % longitoLista] = elemento
        return lista
```

```
    cadenaAlfabeto = string.ascii_letters

    listaCaracteres = list(cadenaAlfabeto)

    claveCaesar = 3 #Si se desea modificar la clave tocar esta variable la cual debera de ser
     la misma que la de encriptar
    listaCaracteres = desplazarlistaIzquierda(listaCaracteres,claveCaesar)

    nombreFicheroDesencriptar = "hola.enc" # Esta variable se cogeria del argumentos.fileIn
    nombreFicheroDestino = "holaDesencriptar.txt" # Esta variable se cogeria del argumentos.
    fileOut

    cadenaEncriptar = ''.join(listaCaracteres)

    ficheroEncriptado = open(nombreFicheroDesencriptar, 'r')
    ficheroDesencriptado = open(nombreFicheroDestino, 'w')
    try:
        # Procesamiento del fichero
        for linea in ficheroEncriptado:
            trantab = linea.maketrans(cadenaAlfabeto, cadenaEncriptar)
            lineaDesencriptada = linea.translate(trantab)
            ficheroDesencriptado.write(lineaDesencriptada)

    finally:
        ficheroEncriptado.close()
        ficheroDesencriptado.close()
```

Si deseamos realizar que estos mismos algoritmos permitan varían la clave de desplazamiento, debemos de agregar este argumento.

```
    parser.add_argument("clave", help="Desplazamiento para la encriptacion Caesar", type=int)
    argumentos = parser.parse_args()
```

Posteriormente únicamente necesitaremos coger la clave.

```
    claveCaesar = argumentos.clave
```

1.2.3. Combinación algoritmo Caesar y compresión

A la hora de realizar esta parte de la práctica se nos propone unir el algoritmo Caesar con la posibilidad de comprimir el archivo encriptado, en este caso mediante el módulo **ZipFile**, podremos trabajar con ficheros comprimidos. Ya sea que queramos descomprimirlos o crearlos.

Para ello lo primero que tendremos que hacer será importar los módulos correspondientes, y a partir de ello ya podremos ir trabajando poco a poco.

Encriptación Caesar + Compresión

```
    import argparse
    import string
    import zipfile
    from os import remove #Este para poder eliminar ficheros temporales
```

Una vez importados los módulos que necesitamos utilizaremos muchos aspectos ya vistos antes. Como por ejemplo las funciones de desplazamiento y el tratamiento de argumentos.

El trozo de código necesario para comprimir el fichero encriptado sería el siguiente:

```
    nombreFicheroEncriptar="hola.txt"
```

```
    nombreFicheroDestino="hola.enc"

    listaCaracteres = list(cadenaAlfabeto)
    claveCaesar = 3 #Si se desea modificar la clave tocar esta variable
    listaCaracteres = desplazarlistaDerecha(listaCaracteres,claveCaesar)

    ficheroPlano = open(nombreFicheroEncriptar, 'r')
    ficheroEncriptado = open(nombreFicheroDestino, 'w')
    cadenaEncriptar = "".join(listaCaracteres)
    try:
        # Procesamiento del fichero
        for linea in ficheroPlano:
            trantab = linea.maketrans(cadenaAlfabeto, cadenaEncriptar)
            lineaEncriptada = linea.translate(trantab)
            ficheroEncriptado.write(lineaEncriptada)

    finally:
        ficheroPlano.close()
        ficheroEncriptado.close()
```

```
    nombreFicheroZip = nombreFicheroEncriptar[:nombreFicheroEncriptar.find('.')] + ".zip"
    try:
        with zipfile.ZipFile(nombreFicheroZip, 'w') as myzip:
            myzip.write(nombreFicheroDestino)
            remove(nombreFicheroDestino)
    finally:
        myzip.close()
```

Con ello lo que haremos será crear un objeto de la clase **Zipfile**, en el caso que se pueda abrir lo que ocurrirá será agregar el fichero encriptado al fichero Zip. De esta forma una vez que se haya agregado lo que haremos será borrar el fichero encriptado dejando solo el fichero comprimido. Para eliminar el fichero encriptado utilizaremos la función **remove**.

Descompresión + Desencriptación Caesar

En este caso primero deberemos de descomprimir el fichero y posteriormente lo desencriptaremos como ya lo hemos hecho antes.

Para descomprimir utilizaremos la función **extractall**, para extraer la totalidad de archivos del fichero .zip. La función **printdir**, únicamente la utilizaremos para ver que ficheros se encuentran en el zip

```
    nombreFicheroComprimido ="hola.txt"
    nombreFicheroZip = nombreFicheroComprimido[:nombreFicheroComprimido.find('.')] + ".zip"
```

```
    try:
        with zipfile.ZipFile(nombreFicheroZip, 'r') as zip:
            zip.printdir()
            zip.extractall()
    finally:
        zip.close()
```

Ahora una vez descomprimido, aplicaremos el algoritmo de desencriptación de Caesar.

```
    cadenaAlfabeto = string.ascii_letters

    listaCaracteres = list(cadenaAlfabeto)

    claveCaesar = 3 #Si se desea modificar la clave tocar esta variable la cual debera de ser
     la misma que la de encriptar
    listaCaracteres = desplazarlistaIzquierda(listaCaracteres,claveCaesar)

    cadenaEncriptar = ''.join(listaCaracteres)
    nombreFicheroDesencriptar = nombreFicheroComprimido[:nombreFicheroComprimido.find('.')] +
     ".enc"
    nombreFicheroDestino = nombreFicheroDesencriptar[:nombreFicheroComprimido.find('.')] + "
    Descomprimido.txt"

    ficheroEncriptado = open(nombreFicheroDesencriptar, 'r')
    ficheroDesencriptado = open(nombreFicheroDestino, 'w')
    try:
```

```
        # Procesamiento del fichero
        for linea in ficheroEncriptado:
            trantab = linea.maketrans(cadenaAlfabeto, cadenaEncriptar)
            lineaDesencriptada = linea.translate(trantab)
            ficheroDesencriptado.write(lineaDesencriptada)

    finally:
        ficheroEncriptado.close()
        ficheroDesencriptado.close()
        remove(nombreFicheroDesencriptar)
```

1.2.4. Ataque de fuerza bruta a un texto crifado por Caesar

En primer lugar, debemos de saber en que consiste un ataque de fuerza bruta. Un ataque de fuerza bruta es aquel intento de descifrar un texto, contraseña, mediante el método de prueba y error con la esperanza de dar con la combinación correcta finalmente.

En el caso del algoritmo de César, deberemos de realizar distintos descifrados con distintas claves. Para ello realizaremos las comprobaciones necesarias con diferentes combinaciones y lo mostraremos por pantalla.

El código necesario para ello podría ser el siguiente:

```
    nombreficheroEncriptado = "hola.enc"
    try:
        # Procesamiento del fichero
        cadenaAlfabeto = string.ascii_letters

        listaCaracteres = list(cadenaAlfabeto)
        for i in range(len(cadenaAlfabeto)):
            listaCaracteres = desplazarlistaIzquierda(listaCaracteres,i)
            cadenaFuerzaBruta = ''.join(listaCaracteres)
            ficheroEncriptado = open(nombreficheroEncriptado, 'r')
            print('Para un valor de clave ', i + 1)
            print('El texto desencriptado seria')
            for linea in ficheroEncriptado:
                trantab = linea.maketrans(cadenaAlfabeto, cadenaFuerzaBruta)
                lineaDesEncriptada = linea.translate(trantab)
                print(lineaDesEncriptada)
            ficheroEncriptado.close()
            print()
    finally:
        print("Fin")
```

De esta forma mediante un simple bucle y nuestra función de descifrar podemos fácilmente romper un cifrado Caesar.

1.3. Técnicas de Encriptación Mono-alfabética

En esta técnica lo que se pretende es generar una cadena de sustitución para encriptar de forma aleatoria para ello utilizaremos la función **shuffle**, del módulo **random**. A la cual le pasaremos la lista de caracteres del alfabeto.

Para todo ello deberemos de importar el módulo random.

```
    import random
    listaCaracteres = list(cadenaAlfabeto)
    random.shuffle(listaCaracteres)
    cadenaEncriptar = ''.join(listaCaracteres)
    print (f'{cadenaAlfabeto=}')
    print (f"{cadenaEncriptar=}")
```

1.3.1. Encriptación Mono-alfabética

Posteriormente ya podremos recorrer el fichero que se desea encriptar y realizar las traducciones oportunas. En primer lugar deberemos de introducir en el fichero encriptado la cadena de encriptación para poder realizar luego el proceso de desencriptación mediante ella.

Esto lo podemos ver en el siguiente fragmento:

```
    nombreFicheroEncriptar= "quijote.txt"
    nombreFicheroSalida = "quijote.enc"
    ficheroPlano = open(nombreFicheroEncriptar, 'r')
    ficheroEncriptado = open(nombreFicheroSalida, 'w')
    try:
        # Procesamiento del fichero
        ficheroEncriptado.write(cadenaEncriptar)
        ficheroEncriptado.write("\n")
        for linea in ficheroPlano:
            trantab = linea.maketrans(cadenaAlfabeto, cadenaEncriptar)
            lineaEncriptada = linea.translate(trantab)
            ficheroEncriptado.write(lineaEncriptada)

    finally:
        ficheroPlano.close()
        ficheroEncriptado.close()
```

1.3.2. Desencriptación Mono-alfabética

Para realizar el desencriptado recorreremos el fichero, y tendremos que tratar la primera línea leída de forma diferente ya que ella será nuestra clave de desencriptación.

```
    nombreFicheroDesencriptar = "monoAlfHola.enc"
    nombreFicheroSalida = "holaMonoAlfDesencriptado.txt"
    ficheroEncriptado = open(nombreFicheroDesencriptar, 'r')
    ficheroDesencriptado = open(nombreFicheroSalida, 'w')
    try:
        # Procesamiento del fichero
        primeraLinea = True
        cadenaDesencriptar = ""
        for linea in ficheroEncriptado:
            if primeraLinea:
                cadenaDesencriptar = linea.strip("\n")
                primeraLinea= not(primeraLinea)
            else:
                trantab = linea.maketrans(cadenaDesencriptar, cadenaAlfabeto)
                lineaEncriptada = linea.translate(trantab)
                ficheroDesencriptado.write(lineaEncriptada)

    finally:
        ficheroEncriptado.close()
        ficheroDesencriptado.close()
```

Como podemos ver lo primero que hacemos es abrir los ficheros, leemos cada línea del fichero y en el caso que sea la primera línea, esta será nuestra clave una vez que le quitemos el \n. Posteriormente desencriptaremos mediante la función maketrans y translate como ya hemos explicado anteriormente.

1.3.3. Ataque de fuerza bruta mediante análisis de frecuencias

En este caso, para realizar el ataque lo que deberemos de realizar será en primer lugar calcular las frecuencias de aparición de cada una de las letras del alfabeto, ya que como sabemos los algoritmos de encriptación que se basan en un cambio simple, provocará que las frecuencias de las letras sean las mismas, la única diferencia será que no corresponderán a sus caracteres reales.

Por ello, lo que deberemos de saber será las frecuencias relativas de nuestro texto. Esto lo podemos realizar mediante este pequeño fragmento de código.

```

    mapa_frecuencias = {}
    nombreFicheroEncriptado = "quijote.enc"
    fichero = open(nombreFicheroEncriptado, 'r')
    try:
        # Procesamiento del fichero

        for linea in fichero:
            for caracter in linea:
                if caracter in mapa_frecuencias:
                    mapa_frecuencias[caracter] = mapa_frecuencias[caracter] + 1
                else:
                    mapa_frecuencias[caracter] = 1
```

```
14
15      finally:
16          fichero.close()
```

De esta manera hemos calculado facilmente las frecuencias de cada uno de los caracteres, si deseamos echarle un vistazo podemos:

```
1       print (mapa_frecuencias)
```

Una vez visualizadas las frecuencias, lo que tendríamos que hacer sería poseer las frecuencias de los caracteres en el idioma a trabajar. Y una vez conocido ello podríamos realizar la sustitución.

Para ello usaríamos el siguiente fragmento.

```
1       ordenDeFrecuencias = " ,E,A,O,S,R,N,I,D,L,C,T,U,M,P,B,G,V,Y,Q,H,F,Z,J,X,K,W"
2       ordenDeFrecuencias = ordenDeFrecuencias.split(sep=',')
3       sumatorio = 0
4       for caracter in mapa_frecuencias.keys():
5           sumatorio += mapa_frecuencias[caracter]
6
7       for caracter in mapa_frecuencias.keys():
8          mapa_frecuencias[caracter]/=sumatorio
9
10      mapa_frecuenciasOrdenado = dict(sorted(mapa_frecuencias.items(),key = lambda item: item
        [1], reverse = True))
```

Para realizar el cambio lo que haremos será recorrer el fichero y mediante la función replace realizaremos el cambio.

```
1       nombreFicheroAtFrecuencias = "quijote.enc"
2       nombreFicheroDestinoAtFrecuencias = "quijoteFrecuencias.txt"
```

```
1       fichero = open(nombreFicheroAtFrecuencias, 'r')
2       fichero2 = open(nombreFicheroDestinoAtFrecuencias,'w')
3       try:
4           # Procesamiento del fichero
5
6           for linea in fichero:
7               i = 0
8               lineaAux = linea
9               for caracter in mapa_frecuenciasOrdenado.keys():
10                  if i < len(ordenDeFrecuencias):
11                      linea = linea.replace(caracter, ordenDeFrecuencias[i])
12                      i+=1
13                  else:
14                      break
15              fichero2.write(linea)
16              print("Linea Tratada " + linea)
17              print("Linea Original " + lineaAux)
18
19      finally:
20          fichero.close()
21          fichero2.close()
```

1.4. Técnicas de Transposición de Caracteres

En este caso tenemos diferentes técnicas las cuales algunas que hemos visto podrían ser:

1.4.1. Transposición diagonal (Rail Fence)

Este algoritmo se basa en la codificación del mensaje mediante una matriz la cual poseerá “r” filas, siendo r el número de raíles deseados (la clave) y “n” columnas, siendo n el tamaño del mensaje a transmitir. De esta forma lo que haremos será ir recorriendo la matriz en diagonal, introduciendo uno a uno los caracteres del mensaje. Una vez rellenada la matriz lo que hacemos es leer fila por fila juntando el contenido de las celdas. Un ejemplo visual sería el siguiente:

Por ello en primer lugar para cifrar el mensaje lo que haremos será lo siguiente:

```
1       def cipherRailFence (textoPlano, railes):
2           #Creamos la matriz donde guardaremos el texto plano para posteriormente rellenarlas.
3
```

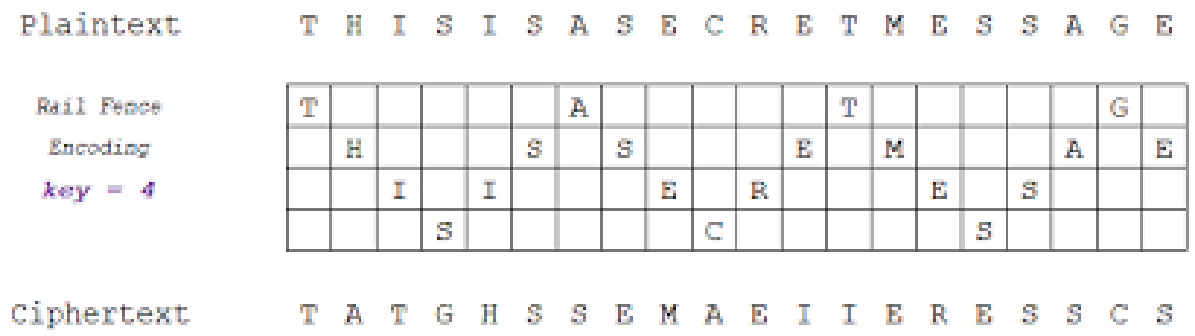

Figura 1.2: Cifrado Rail Fence

```
        matriz = [["-" for x in range(len(textoPlano))] for y in range (railes)]

        fila = 0

        for i in   range(len(textoPlano)):
            matriz[fila][i] = textoPlano[i]
            if fila >= (railes -1):
                fila-=1
            elif fila >= 0:
                fila+=1
        for i in range(railes):
            print("".join(matriz[i]))

        ct=[]
        for i in range(railes):
            for j in range(len(textoPlano)):
                if matriz[i][j]!='-':
                    ct.append(matriz[i][j])

        cipherText="".join(ct)
        print("Cipher Text: ",cipherText)
        return cipherText
```

Con esta función podremos cifrar los mensajes fácilmente. Para descifrar lo que haremos será lo siguiente:

```
    def transpose( m ):
    result = [ [ 0 for y in range( len(m) ) ] for x in range( len(m[0]) ) ]

      for i in range( len(m) ):
        for j in range( len(m[0]) ):
          result[ j ][ i ] = m[ i ][ j ]

      return result
```

Para descifrar el mensaje utilizaremos la siguiente función, en la cual le pasaremos el texto plano y el número de railes.

```
    def descipherRailFence (textoPlano, railes):
        #Creamos la matriz donde guardaremos el texto plano para posteriormente rellenarlas.

        result = ""

        matrix = [["" for x in range(len(textoPlano))] for y in range(railes)]

        idx = 0
        increment = 1

        for selectedRow in range(0, len(matrix)):
            row = 0
            for col in range(0, len(matrix[ row ])):
                if row + increment < 0 or row + increment >= len(matrix):
                    increment = increment * -1
```

```
                if row == selectedRow:
                    matrix[row][col] += textoPlano[idx]
                    idx += 1

                row += increment

        matrix = transpose( matrix )
        for list in matrix:
            result += "".join(list)

        return result
```

```

    mensaje = "Hola Esto Es Una Prueba"
    railes=2
    mensajeCifrado = cipherRailFence(mensaje,railes)
    mensajeDescifrado = descipherRailFence(mensajeCifrado,railes)

    print (f"{mensajeDescifrado=}")
```

1.4.2. Transposición de columna (Columna Cipher)

En este caso el algoritmo se basa en el cifrado de un texto plano modificando la posición de los caracteres, siguiendo un esquema definido. En este caso lo que hacemos es a partir de una palabra obtendremos su longitud y crearemos una matriz que tenga como número de columnas esta longitud, a partir de ello rellenaremos la matriz por filas, y finalmente cogeremos el contenido del mensaje recorriendo la matriz.

En el siguiente código se podrá ver como se ha desarrollado el cifrado y descifrado mediante la transposición de columnas.

Para este caso utilizaremos la función ceil para redondear el número de celdas que necesitamos. Para ello deberemos de importar el módulo **math**.

```
    import math
```

Una vez realizado esto podemos definir nuestras funciones para encriptar y desencriptar mediante el algoritmo de **Columna Cipher**.

```
    def cifrarMensajeColumnarCypher(longitud, mensaje):
        textoCifrado = [''] * longitud
        for columna in range (longitud):
            i = columna
            while i < len(mensaje):
                textoCifrado[columna]+= mensaje [i]
                i+=longitud

        return ''.join(textoCifrado)
```

```
    def desCifrarMensajeColumnarCypher(longitud, mensaje):
        numeroColumnas = math.ceil(len(mensaje)/longitud)
        numeroFilas = longitud
        numeroCeldasOcupadas = (numeroColumnas * numeroFilas) - len(mensaje)
        textoPlano = [''] * numeroColumnas
        columna = 0
        fila = 0

        for caracter in mensaje:
            textoPlano[columna] += caracter
            columna +=1

            if (columna == numeroColumnas) or (columna == numeroColumnas - 1) and (fila >=
    numeroFilas - numeroCeldasOcupadas):
                columna = 0
                fila +=1

        return ''.join(textoPlano)
```

De esta forma podremos cifrar y descifrar los mensajes facilmente.

Si deseamos probar esto podemos utilizar este pequeño fragmento de código:

```
    clave = "dos"
    mensaje = input ("Introduzca el mensaje que se desea cifrar: ")
    modo = 2
    if modo == 1:
        mensajeTratado = cifrarMensajeColumnarCypher(len(clave),mensaje)
    elif modo == 2:
        mensajeTratado = desCifrarMensajeColumnarCypher(len(clave),mensajeEncriptado)

    print("Mensaje Tratado -> " + mensajeTratado)
    mensajeEncriptado = mensajeTratado
```

2

Python Hacking Resolución

En este cuaderno se pretende enseñar los conceptos necesarios para realizar la práctica de **Python Hacking 1**. En el se proponen diferentes ejercicios para familiarizarnos con Python.

Gracias a la posibilidad de Jupyter, podremos ver rápidamente los resultados fácilmente.

2.1. Módulo Zipfile en Python

Gracias al módulo Zipfile, podremos trabajar facilmente con el archivos Zip, en el caso que deseemos generar ficheros comprimidos de una forma rápida y sencilla.

Para ello lo primero que debemos de tener en cuenta es que deberemos de importar dicho módulo.

```
1  import zipfile
```

Una vez importado ya tendremos acceso a la clase **Zipfile**, con el que podremos trabajar con ficheros Zip, ya sea creándolos, leyéndolos, escribiendo o agregando archivos.

Para crear un fichero Zip lo primero que tendremos que hacer será indicar el nombre del fichero que deseamos crear.

```
1  nombreFichero = "hola.txt"
2  nombreZip = nombreFichero[:nombreFichero.find('.')] + ".zip"
```

Una vez tenemos el nombre del fichero a comprimir y el nombre del fichero que vamos a generar, podemos crear el objeto.

```
1  with zipfile.ZipFile (nombreZip, 'w', zipfile.ZIP_DEFLATED) as objetoZip:
2
3          objetoZip.write(nombreFichero)
```

De esta forma tan simple hemos creado el fichero Zip, mediante el objeto Zipfile.

Puede ocurrir que no entandamos el funcionamiento de with o porque motivo no cerramos el fichero. La sentencia with cerrará el fichero Zipfile, cuando finalice todas sus sentencias. De esta forma podemos omitir el cierre.

A la hora de crear el objeto Zipfile, deberemos de pasarle el nombre que tendrá, el modo de acceso el cual podrá tomar los siguientes valores:

- **r ->** Para leer un archivo existente.
- **w ->** Si deseamos crear nuevo archivo en el cual escribir.
- **a ->** Para añadir nuevos archivos a un objeto zip, ya creado. En caso de no existir el archivo creara uno nuevo.
- **x ->** Para crear y escribir exclusivamente un nuevo archivo. En caso de que ya exista dará un FileExistsError.

Después del modo, indicaremos el método de compresión, el puede ser:

- ZIP_STORED
- ZIP_DEFLATED

- ZIP_BZIP2
- ZIP_LZMA

Si deseáramos comprimir varios archivos mediante un solo script lo que deberíamos de hacer sería realizar un bucle como el siguiente:

```
listadoFicheros = "hola.txt,quijote.txt";
nombreZip = "archivos.zip"
listadoNombreFicheros = listadoFicheros.split(',')
with zipfile.ZipFile (nombreZip, 'w', zipfile.ZIP_DEFLATED) as objetoZip:

    for fichero in listadoNombreFicheros:
        objetoZip.write(fichero)
```

Una vez realizado este ejemplo, independientemente de donde se encuentre el listado de archivos a comprimir se pueden derivar de este ejemplo.

2.1.1. Descomprensión de Archivos

Si leemos la documentación de **Zipfile**, se nos indica que para descomprimir un fichero Zip, únicamente necesitaremos crear una instancia de un objeto ZipFile al cual le pasaremos como argumento al constructor el nombre del fichero Zip, una vez hecho esto únicamente necesitaremos llamar a la función **extractall**, la cual descomprimirá el fichero. En caso que el fichero poseyera contraseña deberemos anteriormente haberle introducido la contraseña a la instancia del objeto mediante la función **setpassword**.

Una vez explicado vamos a realizar un pequeño ejemplo:

```
nombreArchivoZip = "archivos.zip"
zipFileDescomprimir = zipfile.ZipFile(nombreArchivoZip)
zipFileDescomprimir.extractall()
zipFileDescomprimir.close()
```

Como sabemos el módulo Zipfile, nos permite descomprimir ficheros con contraseña, pero todavía no podemos encriptar los ficheros mediante él. Por ello si deseamos agregar una contraseña al fichero comprimido deberemos de utilizar la herramienta zip del sistema.

Un ejemplo sería : **zip -e fichero.zip fichero.txt**

Posteriormente nos pedirá la contraseña. Y tras ello tendremos nuestro fichero comprimido con contraseña.

2.2. Subprocess

El módulo subprocess nos mejora el potencial de Python, permitiendo a los scripts de Python a que tengan acceso a los comandos del Sistema Operativo.

Para ello deberemos importar el módulo **subprocess**, para tener acceso a la función **run**, la cual será la que nos hará de intermediario, con los comandos del sistema. **Nota:** Jupyter no permite utilizar esta función.

```
import subprocess
```

Esta función nos pide la cadena que se desea ejecutar. De esta forma podremos fácilmente, que nuestro script de Python pueda introducir una contraseña en nuestros zip, mediante el comando **zip -e** .

Con el módulo subprocess permitimos que los script de Python tengan acceso a los comandos del Sistema Operativo. De esta forma en esta parte se nos plantea la idea de combinar el módulo Zipfile que nos ofrece Python con el comando zip del Sistema Operativo. Esto debido a que actualmente el módulo Zipfile todavía no nos permite encriptar archivos con contraseña directamente.

Con todo esto explicado, para utilizar el módulo **subprocess**, únicamente tendremos que importarlo, y posteriormente utilizar la función **run**, la cual nos permitirá pasarle el comando con los argumentos deseados, y los inputs que debemos introducir para los diferentes comandos del sistema.

En nuestro caso lo que haremos será pasarle el comando y los argumentos en una lista para que no pueda ocurrir problemas y luego introducirle como un input la contraseña deseada para encriptar.

En este aparatado se pidieron realizar los siguientes scripts:

- Crea un script **(czip_password.py)** que use subprocess y el programa zip y reciba un fichero y una clave y lo comprima y encripte usando la clave proporcionada (deberás tener instalada al menos la versión 3 de zip para ejecutar en la línea de comandos).

```
import subprocess

import argparse

parser = argparse.ArgumentParser(description='Comprime un fichero pasado como argumento')

parser.add_argument("fileIn", help="fichero que se desea comprimir", type=str)

parser.add_argument("key", help="Clave del archivo comprimido", type=str)

args = parser.parse_args()

nombreFicheroEntrada = args.fileIn

clave = args.key

ordenTerminal = ["zip", "-e"]

nombreFicheroSinExtension = nombreFicheroEntrada[:nombreFicheroEntrada.find('.')]

EXTENSIONZIP = ".zip"

nombreFicheroComprimir = nombreFicheroSinExtension + EXTENSIONZIP

ordenTerminal.append(nombreFicheroComprimir)

ordenTerminal.append(nombreFicheroEntrada)


subprocess.run(ordenTerminal,input=clave.encode())
```

- Crea un script **(cunzip_password.py)** que use subprocess y el programa unzip y reciba un fichero comprimido y una clave y lo descomprima usando la clave proporcionada informando si la clave es o no correcta.

```

import subprocess

import argparse

parser = argparse.ArgumentParser(description='Descomprime un fichero pasado como
argumento con su clave')

parser.add_argument("fileIn", help="fichero que se desea descomprimir", type=str)

parser.add_argument("key", help="Clave del archivo a descomprimir", type=str)

args = parser.parse_args()

nombreFicheroEntrada = args.fileIn

clave = args.key

ordenTerminal = ["unzip", "-P"]

ordenTerminal.append(clave)
ordenTerminal.append(nombreFicheroEntrada)

subprocess.run(ordenTerminal)
```

2.2.1. Desencriptar ficheros con contraseña

Como hemos indicado con anterioridad, zipfile lo permite, por ello lo único que debemos de hacer será introducir la contraseña a la hora de intentar descomprimir.

```
1
2
3     nombreFicheroZip = "archivoConPassword.zip"
4     clave = "claveZip"
5
6     ficheroZip = zipfile.ZipFile(nombreFicheroZip)
7
8     try:
9
10        ficheroZip.extractall(pwd=clave.encode())
11        print("[+] Password Correcta" )
12
13    except:
14        print("[-] Password Incorrecta " + clave)
```

2.3. Zip-Cracker

Si deseáramos romper la protección de la contraseña de un fichero zip, podemos realizarlo mediante fuerza bruta, a partir de un diccionario de contraseñas. Para ello, en primer lugar necesitaremos un fichero que nos sirva como diccionario de contraseñas, el cual posea un listado de contraseñas y que contenga la contraseña valida. De esta forma podremos ir probando cada una de las contraseñas.

Para ello deberemos de recorrer el fichero donde se encuentren las contraseñas, y posteriormente “intertar” descomprimir el fichero.

```
1     diccionario ="diccionario.txt"
2     nombreFicheroZip = "archivoConPassword.zip"
3     try:
4         fichero = open (diccionario,'r')
5         ficheroZip = zipfile.ZipFile(nombreFicheroEntrada)
6         encontradaKey = False
7         for password in fichero:
8             try:
9                 clave = password[:password.find('\\')]
10                ficheroZip.extractall(pwd=clave.encode())
11                encontradaKey = not(encontradaKey)
12                print("[+] Password Correcta " + clave)
13                break;
14            except:
15                    print("[-] Password Incorrecta " + password)
16
17    finally:
18        fichero.close()
19        ficheroZip.close()
20    if encontradaKey is False :
21        print ("Lo sentimos pero el diccionario actual no posee la password del fichero Zip")
22    else :
23        print("El fichero ha sido descomprimido correctamente")
```

3

Autenticación de Mensajes y Funciones Hash

3.1. Introducción

Con este cuaderno se pretende exponer el funcionamiento de las funciones Hash, además de su aplicación en la autenticación de mensajes. Para ello haremos diferentes scrypts en Python.

Como sabemos las funciones Hash generan a partir de cadenas binarias una salida de longitud fija, con el aspecto importante de que es prácticamente imposible obtener a partir de los datos de salida los datos originales. Siendo así una función unidireccional.

Además de que es prácticamente imposible que dos mensajes generen el mismo hash.

Una de sus principales uso sería la gestión de identificadores y contraseñas. Por ejemplo, a la hora de realizar el inició de sesión en un software, el sistema deberá de comprobar que el usuario y la contraseña introducidas son las correctas para poder acceder al servicio. Para que exista un mayor nivel de seguridad, el sistema no guarda la contraseña, sino que guarda el Hash de la contraseña. Y, por tanto, cuando introducimos nuestra contraseña para acceder, el sistema calcula el Hash de la contraseña y lo compara con el guardado en el sistema. Si ambos coinciden, permitirá el acceso.

Una vez entendido los funcionamientos y utilidades nos centraremos en como trabajar con ellas mediante Python.

3.2. PyCryptodome

Para trabajar con funciones hash, Python posee el módulo **PyCryptodome**, el cual pretende actualizar el módulo **PyCrypto**. Con el podremos utilizar funciones de bajo nivel para criptográfica.

En primer lugar deberemos de instalar el módulo **pycryptodome**, para ello mediante un terminal utilizaremos el comando ***pip pycryptodome***.

Una vez hecho eso podremos importarlo en un nuestros scripts y trabajar con él. En concreto deberemos de importar las funciones del submodulo Hash de Crypto. En nuestro caso en primer lugar trabajaremos con MD5 y SHA256, por ello las importaremos solo ellas.

```
1             from Crypto.Hash import MD5
2             from Crypto.Hash import SHA256
3
```

Para generar un hash mediante ellas podemos crearnos una función simple para cada uno de los métodos para simplificarnos el trabajo a la hora de hashear un texto.

```
1             def generarHashCodeMD5 (datoEntrada):
2                 objetoHash = MD5.new(data=datoEntrada.encode())
3                 return objetoHash.digest()
4
5             def generarHashCodeSHA256 (datoEntrada):
6                 objetoHash = SHA256.new(data=datoEntrada.encode())
7                 return objetoHash.digest()
8
```

Para ello en cada una de las funciones lo que hemos hecho ha sido crear un objeto de la clase MD5 en el caso que queramos trabajar con MD5, al cual en el constructor le hemos pasado el dato (una cadena binaria). Posteriormente llamaremos a la función digest, para obtener el hash del objeto.

Como podemos ver ambas clases poseen la misma interfaz para trabajar con ellos.

Si deseamos encriptar un texto podemos utilizar el siguiente fragmento en el caso que queramos utilizar MD5.

```
            nombreFichero = "fichero.txt"

```

```
            print ("Hash generado por MD5")
            hashFileGenerado = generarHashCodeMD5(nombreFichero)
            print (f"{hashFileGenerado = }")

```

Si deseáramos usar el SHA-256 únicamente necesitamos utilizar el **generarHashCodeSHA256**

```
            print ("Hash generado por SHA-256")
            hashFileGenerado = generarHashCodeSHA256(nombreFichero)
            print (f"{hashFileGenerado = }")

```

Si deseamos comprobar el hash de un fichero o de un mensaje lo que deberos de hacer es generar el hash del fichero y compararlo con el que el usuario introduzca.

Para ello podemos utilizar el siguiente script el cual mediante diferentes argumentos podemos realizar las comprobaciones de Hash tanto para ficheros, mensajes, etc.

```
parser = argparse.ArgumentParser(description='Aglutina todos los script pedidos')

parser.add_argument('-f1', dest = "fileIn1", type=str,  metavar='FILEIN1',
                                help='Nombre del fichero 1 que se desea tratar')

parser.add_argument('-f2', dest = "fileIn2", type=str,  metavar='FILEIN2',
                                help='Nombre del fichero 2 que se desea tratar')

parser.add_argument('-hc', dest = "hashCode", type=str, metavar='HASHCODE',
                                help='Introduzca el Hash que se desea comparar con el fichero
    1')

parser.add_argument('-m',dest = "MD5" , help='Si se activa se realizarán las pruebas con el
   algortimo MD5', action='store_true')

parser.add_argument('-s',dest = "SHA256", help='Si se activa se realizarán las pruebas con el
    algortimo SHA256', action='store_true')

args = parser.parse_args()

nombreFichero1 = args.fileIn1

nombreFichero2 = args.fileIn2

hashCodeIntroducido = args.hashCode

MD5Activo = args.MD5

SHA256Activo = args.SHA256
```

```

            nombreFichero1 = "hola.txt"

            nombreFichero2 = "hola.txt"

            hashCodeIntroducido = ""

            MD5Activo = True

            SHA256Activo = True


            if MD5Activo is False and SHA256Activo is False:
```

```
14              print ("Lo sentimos pero debe de escoger al menos un tipo del algoritmo Hash
    para trabajar")
15              print ("El programa finalizara")
16              exit (-1)
17
18          if nombreFichero1 is not None and nombreFichero2 is not None:
19              #Deseamos comparar los hash Code de cada uno de los ficheros para saber si
    son el mismo.
20              if MD5Activo is True:
21                  print ("Realizando comparación con el MD5")
22                  hashFile1 = generarHashCodeMD5(nombreFichero1)
23                  hashFile2 = generarHashCodeMD5(nombreFichero2)
24                  if hashFile1 == hashFile2:
25                      print ("[+] Los dos ficheros poseen el mismo hash code " + str(
    hashFile1))
26                  else:
27                      print ("[-] Los dos ficheros NO poseen el mismo hash code " )
28                      print ("Fichero " + nombreFichero1 + " posee el hash code " + str(
    hashFile1))
29                      print ("Fichero " + nombreFichero2 + " posee el hash code " + str(
    hashFile2))
30              if SHA256Activo is True:
31                  print ("Realizando comparación con el SHA-256")
32                  hashFile1 = generarHashCodeSHA256(nombreFichero1)
33                  hashFile2 = generarHashCodeSHA256(nombreFichero2)
34                  if hashFile1 == hashFile2:
35                      print ("[+] Los dos ficheros poseen el mismo hash code " + str(
    hashFile1))
36                  else:
37                      print ("[-] Los dos ficheros NO poseen el mismo hash code " )
38                      print ("Fichero " + nombreFichero1 + " posee el hash code " + str(
    hashFile1))
39                      print ("Fichero " + nombreFichero2 + " posee el hash code " + str(
    hashFile2))
40
41          elif nombreFichero1 is not None and hashCodeIntroducido is not None:
42              #En este caso lo que deseamos es compara el hash del fichero1 con el hash
    introducido
43              if MD5Activo is True:
44                  hashFile = generarHashCodeMD5(nombreFichero1)
45                  if hashFile == hashCodeIntroducido:
46                      print ("[+] El hash code introducido es el del fichero " + str(
    hashFile))
47                  else:
48                      print ("[-] El fichero no posee el mismo hash code que el que se ha
    introducido " )
49                      print ("Fichero " + nombreFichero1 + " posee el hash code" + str(
    hashFile) + "!= " + hashCodeIntroducido)
50              if SHA256Activo is True:
51                  hashFile = generarHashCodeSHA256(nombreFichero1)
52
53                  if hashFile == hashCodeIntroducido:
54                      print ("[+] El hash code introducido es el del fichero " + str(
    hashFile))
55                  else:
56                      print ("[-] El fichero no posee el mismo hash code que el que se ha
    introducido " )
57                      print ("Fichero " + nombreFichero1 + " posee el hash code" + str(
    hashFile) + "!= " + hashCodeIntroducido)
58          elif nombreFichero1 is not None:
59              # En este caso solo queremos mostrar los hash Code dependiendo de los tipos
60              if MD5Activo is True:
61                  print ("Hash generado por MD5")
62                  hashFileGenerado = generarHashCodeMD5(nombreFichero1)
63                  print (f"{hashFileGenerado = }")
64
65              if SHA256Activo is True:
66                  print ("Hash generado por SHA256")
67                  hashFileGenerado = generarHashCodeSHA256(nombreFichero1)
68                  print (f"{hashFileGenerado = }")
69
```

```
70              else:
71                  if nombreFichero2 is not None:
72                      print ("[ERROR] Solo se ha suministrado el nombreFichero2, lo sentimos,
    necesitamos el argumento fichero1 para comparar")
73
74                  if hashCodeIntroducido is not None:
75                      print ("[ERROR] Solo se ha suministrado el hashCode, lo sentimos,
    necesitamos el argumento fichero1 para comparar")
76
77
78
```

3.3. Autenticación de mensajes basado en Hash

En el caso de desear encriptar mediante la autenticación de mensajes, para ello utilizamos el objeto HMAC, en la cual deberemos de pasar la clave secreta, y el modo de encriptación.

Para ello podemos usar el siguiente fragmento:

```
1         def cifrarFichero (nombreFichero, objetoHash):
2             mensajeCifrado =""
3             try:
4                 fichero = open (nombreFichero,'r')
5                 for linea in fichero:
6                     linea = linea[:linea.find('\\')] #Con esto quitamos el barra n
7                     objetoHash.update(linea.encode())
8
9                 mensajeCifrado = objetoHash.hexdigest()
10            finally:
11                fichero.close()
12            return mensajeCifrado
13
14        def cifrarMensaje (mensaje, objetoHash):
15            objetoHash.update(mensaje.encode())
16            mensajeCifrado = objetoHash.hexdigest()
17            return mensajeCifrado
18
```

```
1             MD5Activo = True
2             SHA256Activo = True
3             mensajeDeseado = "Mensaje de prueba"
4             claveSecreta = "12345"
5
```

```
1
2             if MD5Activo is True:
3                 print ("Mediante MD5")
4                 objetoHMAC = HMAC.new(claveSecreta.encode(), digestmod=MD5)
5                 hashGenerado = cifrarMensaje(mensajeDeseado,objetoHMAC)
6
7                 print ("Valor hexadecimal -> " + hashGenerado)
8
9             if SHA256Activo is True:
10                print ("Mediante SHA256")
11                objetoHMAC = HMAC.new(claveSecreta.encode(), digestmod=SHA256)
12                hashGenerado = cifrarMensaje(mensajeDeseado,objetoHMAC)
13
14                print ("Valor hexadecimal -> " + hashGenerado)
15
```

Si deseáramos cifrar un fichero de texto entero, podemos utilizar la función **cifrarFichero**, a la cual le deberemos de pasar el nombre del fichero y el objeto HMAC.

4

Encriptación Simétrica

Con este cuaderno se pretende exponer los conceptos claves de la encriptación simétrica, la cual debemos de saber que consiste en aquella en la que encriptamos el mensaje mediante una clave secreta la cual deberá de ser compartida por el emisor y el receptor del mensaje.

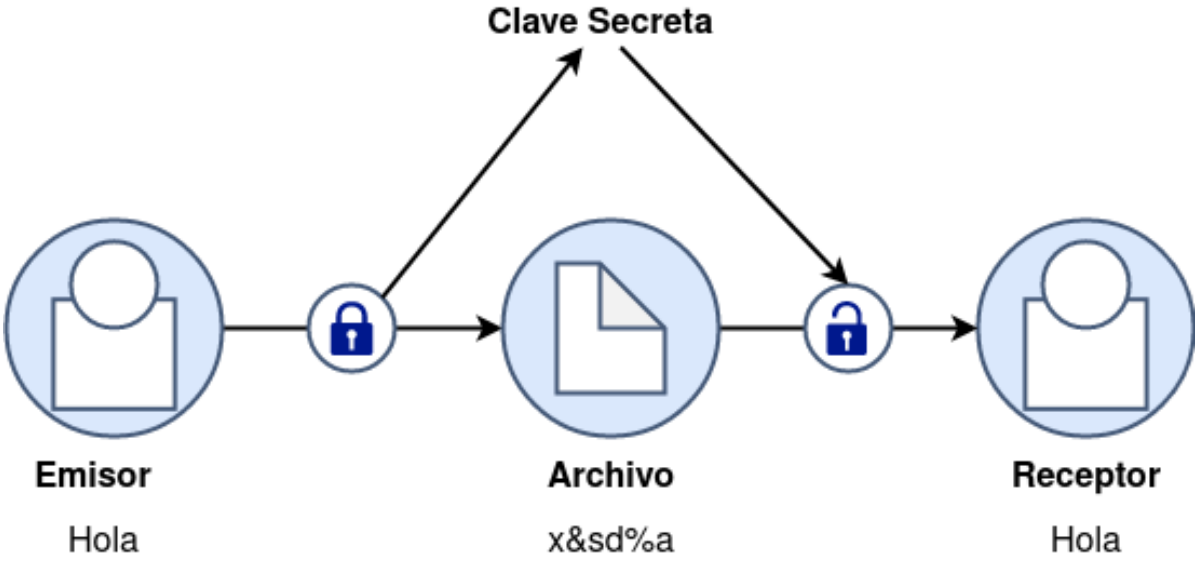

Figura 4.1: Encriptación Clave Simétrica

A la hora de trabajar con encriptación simétrica existen dos tipos de algoritmos aquellos en los cuales vamos cifrando el mensaje, generando bloques los cuales serán transmitidos y que para poder desencriptar el mensaje necesario descencriptar cada uno de los bloques transmitidos.

En el otro caso, tenemos los **Stream Ciphers**, en los cuales se realizará la encriptación bit a bit.

Para trabajar con este cuaderno será necesario tener instalado el paquete **PyCryptodome** de Python, por ello en caso de no tenerlo instálelo de la siguiente manera:

pip install pycryptodome

Una vez realizado esto podremos importa las funciones necesarias para realizar la encriptación simétrica de bloques.

4.1. Modos clásicos de operación para encriptación simétrica de bloque

A la hora de trabajar con los modos clásicos se nos propone utilizar el algoritmo **AES**, para comprender el funcionamiento de la encriptación.

Para ello, lo primero que haremos será importar la clase AES, y así poder trabajar con una instancia de esta clase. A esta clase le deberemos de pasar en el constructor la "'clave" y el método de encriptación a emplear.

```
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
```

Deberemos de indicar cual es el mensaje que deseamos encriptar, además de generar una clave de forma aleatoria:

```
mensaje = "Hola mundo"
mensajeBinario = mensaje.encode()
clave = get_random_bytes(16)
cipher = AES.new(clave, AES.MODE_CBC)
ct_bytes = cipher.encrypt(pad(mensajeBinario,AES.block_size))
```

De esta forma en ct_bytes tendremos el texto cifrado en binario, para poder obtener el texto en un formato UTF-8, podemos utilizar **b64encode**, para obtener ese valor en codificación **utf-8**.

```
iv = b64encode(cipher.iv).decode('utf-8')
ct = b64encode(ct_bytes).decode('utf-8')
```

Si mostramos ambas variables podemos ver tanto el vector de inicialización como el texto encriptado.

```
print (f"{iv=}")
print (f"{ct=}")
```

4.1.1. Desencriptación

Para desencriptar lo que deberemos de hacer será en primer lugar, decodificar el mensaje y el vector de inicialización, ya que lo tenemos en formato **Base64**. Para ello usaremos ***b64decode***.

Tras esto lo que haremos será crear nuestro objeto de la clase AES, el cual utilizaremos para desencriptar el mensaje.

```
ivCifrado = iv
cvCifrafo = ct
iv = b64decode(iv)
ct = b64decode(ct)
cipher = AES.new(clave, AES.MODE_CBC, iv)
```

Una vez creado el objeto, utilizaremos la función **decrypt**, a la cual le pasaremos el mensaje, y mediante la función **unpad**, descomprimiremos el block de AES. De esta forma el mensaje se encontrará guardado en la variable pt.

```
pt = unpad(cipher.decrypt(ct), AES.block_size)
print("Mensaje desencriptado: ", pt)
```

Una vez hecho esto ya tendríamos el texto descifrado. En caso de error nos saltaría la excepción de **KeyError**.

Si deseamos ejecutar todo este fragmento en conjunto sería:

```
try:
    iv = b64decode(ivCifrado)
    ct = b64decode(ctCifrado)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    pt = unpad(cipher.decrypt(ct), AES.block_size)
    print("The message was: ", pt)
except (ValueError, KeyError):
    print("Incorrect decryption")
```

4.1.2. Encriptación de un fichero

Si deseamos encriptar un fichero, como sabemos deberemos de ir leyendo línea a línea del fichero e ir encriptando el contenido de dicha línea.

En este caso se nos propone conocer el funcionamiento de los modos de encriptación **CBC y CTR**, los cuales podemos diferenciar con la siguiente explicación:

- El modo **CBC**, consiste en una encriptación por bloques, en la cual se tendrá que utilizar un IV aleatorio, cada vez que se encripte un mensaje. En este caso se necesitará rellenar los bloques a transmitir con un tamaño fijo para todos.
- El modo **CTR**, consiste en un modo de cifrado de flujo, en este caso lo que hace es mediante una secuencia pseudoaleatoria, independiente del texto, se obtendrán los diferentes IV para encriptar.

Una vez entendidos podemos trabajar con ellos, en primer lugar deberemos de importar los mismos módulos utilizados anteriormente.

```
CBCActivo = True
CTRActivo = False
```

Estas variables las utilizaremos para determinar que modo deseamos utilizar:

```
try:
    fichero = open (nombreFicheroEntrada,'r')
    ficheroEncriptado = open (nombreFicheroDestino,'w')
    if CBCActivo is True:
        cipher = AES.new(key, AES.MODE_CBC)
    elif CTRActivo is True:
        cipher = AES.new(key, AES.MODE_CTR)
    for linea in fichero:
        linea = linea[:linea.find('\\')]
        #Encriptamos el dato

        if CBCActivo is True:
            lineaEncriptadaBytes = cipher.encrypt(pad(linea, AES.block_size))
        elif CTRActivo is True:
            lineaEncriptadaBytes = cipher.encrypt(linea)

        lineaEncriptada = b64encode(lineaEncriptadaBytes).decode('utf-8')
        ficheroEncriptado.write(lineaEncriptada)

finally:
    fichero.close()
    ficheroEncriptado.close()
```

En primer lugar deberemos de abrir los ficheros, tanto el que deseamos encriptar como en el cual deseamos guardar. Posteriormente crearemos nuestro objeto AES, con el cual realizaremos la encriptación.

Posteriormente recorreremos el fichero, eliminando a cada línea el \n, posteriormente, encriptaremos el dato, dependiendo del modo Activo. Para el caso del \CBC deberemos de utilizar **pad**, para especificar el contenido con el tamaño de bloque.

Cosa que no ocurre con **CTR**.

Posteriormente codificaremos el mensaje encrptado en formato **utf-8**. Y escribiendo en el fichero destino.

Si deseamos desencriptar el fichero en primer lugar necesitaremos el IV utilizado y la clave. Para ello podemos mostrarlos por pantalla o guardarlos en un fichero json.

```
iv = b64encode(cipher.iv).decode('utf-8')
key = key.hex()
result = {}
result["iv"] = iv
result["key"] = key
print (iv)
print (key)

with open('data.json', 'w') as outfile:
    json.dump(result, outfile)
```

Desencriptación de un fichero

A la hora de desencriptar deberemos de conocer cual fue el modo utilizado para encriptar. Además de utilizar la clave y el IV, el cual cargaremos del fichero json.

```
    try:
        with open("data.json") as json_data:
            b64 = json.load(json_data)
            print(b64)
        iv = b64decode(b64['iv'])
        cipher = AES.new(key, AES.MODE_CBC, iv)
        try:
            fichero = open (nombreFicheroEntrada,'r')
            ficheroEncriptado = open (nombreFicheroDestino,'w')
            if CBCActivo is True:
                cipher = AES.new(key, AES.MODE_CBC)
            elif CTRActivo is True:
                cipher = AES.new(key, AES.MODE_CTR)
            for linea in fichero:
                linea = linea[:linea.find('\\')]
                #Encriptamos el dato
                linea = b64decode(linea)
                if CBCActivo is True:
                    lineaDesencriptadaBytes = cipher.decrypt(unpad(linea, AES.block_size))
                elif CTRActivo is True:
                    lineaDesencriptadaBytes = cipher.decrypt(linea)

                ficheroEncriptado.write(lineaDesencriptada)

        finally:
            fichero.close()
            ficheroEncriptado.close()


        print("The message was: ", pt)
    except (ValueError, KeyError):
        print("Incorrect decryption")
```

Como podemos ver se utiliza la función **decrypt** para desencriptar la información. Anteriormente debemos de decodificar el mensaje.

4.1.3. Encriptación Autentificada con Datos Asociados

En esta parte opcional se pretende exponer el modo **EAX** del algoritmo AES, el cual nos permitirá esta forma de encriptación. Para ello será necesario cambiar el modo que le pasamos a la hora de crear la instancia del objeto AES. Y posteriormente utilizar la función de encrypt_and_digest, para obtener el texto cifrado.

Para descifrar utilizaremos la función decrypt_and_verify.

El siguiente trozo de código nos permite encriptar el mensaje, para ello crearemos una clave de forma aleatoria, y a partir de ella crearemos la instancia del objeto AES con el modo EAX. Posteriormente utilizaremos la función **encrypt_and_digest**, como ya hemos indicado y finalmente lo que haremos será volcar la información al fichero.

```
    key = get_random_bytes(16)
    data="Hola mundo".encode("utf-8")
    cipher = AES.new(key, AES.MODE_EAX)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    file_out = open("encrypted.bin", "wb")
    [ file_out.write(x) for x in (cipher.nonce, tag, ciphertext) ]
    print(key.hex())
```

Para el caso de desencriptar sería de la siguiente forma:

```
    i = key.hex() #Para poder coger la clave correcta
    key= bytes.fromhex(i)
    file_in = open("encrypted.bin", "rb")
    nonce, tag, ciphertext = [ file_in.read(x) for x in (16, 16, -1) ]
    # let's assume that the key is somehow available again
    cipher = AES.new(key, AES.MODE_EAX, nonce)
    data = cipher.decrypt_and_verify(ciphertext, tag)
    print(data.decode())
```

Una vez visto todo esto ya podemos trabajar con la encriptación de un fichero completo. Para ello como siempre leeremos el fichero e iremos utilizando el objeto AES, para encriptar o desencriptar.

```
    nombreFicheroEntrada = "hola.txt"

    nombreFicheroDestino = "hola.bin"

    key = get_random_bytes(16)

    cipher = AES.new(key, AES.MODE_EAX)
```

```
    data = ""
    try:
        fichero = open (nombreFicheroEntrada,'r')
        ficheroEncriptado = open( nombreFicheroDestino + ".bin", "wb")

        for linea in fichero:
            #Encriptamos el dato
            data+=linea

        ciphertext, tag = cipher.encrypt_and_digest(data.encode())
        [ ficheroEncriptado.write(x) for x in (cipher.nonce, tag, ciphertext) ]

        print(key.hex())
    finally:
        fichero.close()
        ficheroEncriptado.close()
```

Para el caso de desencriptación sería de la misma forma:

```
    nombreFicheroEntrada = "hola.bin"

    nombreFicheroDestino = "holaDesencriptado.txt"

    i = key.hex()
    key= bytes.fromhex(i)

    cipher = AES.new(key, AES.MODE_EAX)

    #Debemos de leer todo el contenido del fichero y guardarlo en la variable ciphertext
    data = ""
    try:
        fichero = open (nombreFicheroEntrada,'rb')
        ficheroDesencriptado = open( nombreFicheroDestino, "w")

        nonce, tag, ciphertext = [ fichero.read(x) for x in (16, 16, -1) ]

        cipher = AES.new(key, AES.MODE_EAX, nonce)
        data = cipher.decrypt_and_verify(ciphertext, tag)
        print(data.decode())
        ficheroDesencriptado.write(data.decode())

    finally:
        fichero.close()
        ficheroDesencriptado.close()
```

Como podemos ver para encriptar los ficheros hemos recorrido los ficheros y almacenado el contenido en una variable y posteriormente encriptado este contenido. Esto podría dar problemas a la hora de que necesitaríamos almacenar toda la información del fichero en memoria para realizar este proceso. Siendo muy costoso a nivel de memoria.