Universidad de Granada
E.T.S.I Informática y Telecomunicaciones
Grado en Ingeniería Informática (GII)

Memoria Práctica 5: Puzles Hash

Seguridad y Protección de Sistemas Informáticos

Curso 2017-2018

Arthur Rodríguez Nesterenko - DNI Y1680851W

2 de enero de 2018

Índice

1. Tarea 1: Para la función H, realizad, en el lenguaje de programación que queráis, una función que tome como entrada un texto y un número de bits b. Creará un id que concatene una cadena aleatoria de n bits con el texto. Pegará a ese id cadenas aleatorias x de n bits hasta lograr que H(id||x) tenga sus primeros b bits a cero. La salida será el id, la cadena x que haya proporcionado el hash requerido, el valor del hash y el número de intentos llevados a cabo hasta encontrar el valor x apropiado.

El objetivo de esta práctica consiste en realizar un **puzle hash** y estudiar la complejidad interna de estos a la hora del cálculo de preimagenes. Para ello se nos pide implementar una función que tome como entrada un **texto cualquiera** y un **número de bits b**, de forma que a partir de estos se realicen ciertos cálculos que permitan encontrar un determinado valor hash.

La función desarrollada para esta tarea se ha implementado en el lenguaje de programación **Python (version 3.6)**, atendiendo lo más rigurosamente posible a los requisitos solicitados, por lo que a continuación pasaremos a explicarla en detalle de forma que no quede ninguna duda de su funcionamiento de cara a las demás tareas de la práctica.

1.1. Script puzleHash.py

La cabecera del script que implementa la función utilizada en esta tarea incluye las librerías necesarias para generar números aleatorios, para utilizar funciones hash y para manejar cadenas hexadecimales; en nuestro caso se nos pide utilizar una función hash cuya **salida sea de al menos 256 bits**, por lo que utilizaremos la función **SHA-256**.

```
#Practica 5 SPSI: Puzles Hash
#Funcion de la tarea 1

#Librerias para numeros aleatorios, funciones Hash
#y manejo de cadenas hexadecimales
import random
import hashlib
import string
```

Los siguientes elementos que nos encontramos en el script son principalmente **variables globales**. Estas variables son principalmente: una **semilla aleatoria** para poder conseguir los mismos resultados si se repite el experimento, **el numero de bits** n que tendrá el nonce, una **cota superior de valor b** para los experimentos, un **texto de entrada** que servirá de argumento para nuestra función, así como una cadena que contiene los **16 digitos hexadecimales**, una **escala** y el **numero de bits** que produce la salida del digest; estos dos últimos para realizar conversiones intermedias de hexadecimal a binario.

```
#Variables globales a utilizar en la funcion
random.seed(18)
N = 64
maximum_b = 19
texto = """En un lugar de la Mancha, de cuyo nombre no quiero acordar-me,
no ha mucho tiempo que vivia un hidalgo de los de lanza en astillero,
adarga antigua, rocin flaco y galgo corredor."""
#Para la transformacion a binario y gestion de cadenas hexadecimales
hex_digits = string.hexdigits[0:16]
escala = 16
num_bits = 256
```

Inmediatamente después hemos creado una función `crear_nonce` que básicamente calcula una cadena hexadecimal que se utilizará para crear el **nonce** que se concatenará con el mensaje y además el valor **x** con el que se concatenará el **id**. Recibe como parámetro un tamaño `tam` que indica la longitud de la cadena de salida, teniendo en cuenta que la variable global **N** dicta la longitud en número de bits de la cadena en cuestión

```
#Funcion para crear el nonce
def crear_nonce(tam=8):

  #Creamos una cadena hexadecimal con tam caracteres
  #Los caracteres se eligen de forma aleatoria dentro del
  #conjunto de caracteres hexadecimales
  return ''.join(random.choice(hex_digits) for _ in range(tam))
```

Para controlar que $H(id||x)$ contenga al **inicio de la misma** una cantidad **b** de ceros determinada, nos valemos de una función `comprobar_ceros` que recibe como argumento un `hash` para contar la cantidad de ceros al inicio del mismo. Aquí entran en juego las variables globales `escala` y `num_bits` que sirven para transformar la cadena en hexadecimal a binario.

```
#Funcion para comprobar la cantidad de ceros al inicio del hash
def comprobar_ceros(hash):

  hash_bin = bin(int(hash, escala))[2:].zfill(num_bits)
  count = 0

  for i in range(0,len(hash_bin)):
    if(hash_bin[i] == '0'):
      count+=1
    else:
      break

  return count
```

Una vez hemos explicado todos los demás componentes, el último paso consiste en explicar la función `puzle_hash` que implementa todo el funcionamiento descrito en el enunciado de la tarea; recibe como argumentos de entrada un **mensaje** `msg` y un valor **b**.

El experimento se realiza 10 veces y en cada iteración se realiza el siguiente proceso: se calcula el **id** como la concatenación del **mensaje** y un **nonce** de tamaño **N**. A continuación se calcula una cadena **X** de la misma longitud que el nonce para posteriormente realizar la operacion $H(id||x)$, esto es, calcular el hash (mediante la función SHA-256) de la concatenación del **id y X**. Este último procedimiento se repite tantos **intentos** como sean necesarios hasta conseguir que el **hash calculado** tenga sus primeros **b** bits a cero. Como el experimento se realiza 10 veces, al final se muestra la media del número de intentos empleados. El trozo de código se puede ver a continuación:

```
def puzle_hash(msg, b):

intentos_totales = 0

for reps in range(0,10):
  #Creacion del nonce
  nonce = str(crear_nonce(int(N/4)))

  #Creacion del id, que es el msg concatenado con el nonce
  id = msg + nonce

  #Inicio del puzle hash: numero de intentos y condicion de parada
  intentos_actuales, encontrado = 0, False
  x,hash = "",""

  while(not encontrado):
    #Aumentamos el numero de intentos y creamos id||X
    intentos_actuales+=1
    x = str(crear_nonce(int(N/4)))
    test = id + x
    hash = str(hashlib.sha256(test.encode('utf-8')).hexdigest())
    encontrado = comprobar_ceros(hash) >= b

  intentos_totales += intentos_actuales

media_intentos = intentos_totales/10

print("------------------------------------------------------------")
print("Resultados Medios")
print("Valor B , Intentos : %i , %f" % (b, media_intentos))
print("------------------------------------------------------------")
```

Este script será llamado para cada uno de los valores **b** que elijamos para nuestro experimento, cuyos resultados se mostrarán en la siguiente tarea.

2. Tarea 2: Calculad una tabla/gráfica que vaya calculando el número de intentos para cada valor de b. Con el objeto de que los resultados eviten ciertos sesgos, para cada tamaño b realizad el experimento 10 veces y calculad la media del número de intentos.

El experimento se ha realizado dentro de un entorno pseudo-aleatorio, estableciendo un valor de semilla inicial de 18 para poder repetir el mismo y conseguir los mismos resultados. Tras 10 ejecuciones para cada posible valor de **b**, valores que van desde 1 hasta 18 (dado que la complejidad en tiempo aumenta conforme aumenta el valor de **b**), se ha realizado la media del número de intentos empleados para que se cumpla que $H(id||x)$ contenga al inicio un número **b** de ceros. Los resultados obtenidos se muestran en la tabla y gráfica a continuación:

Valor de b	Intentos (Media)	Valor de b	Intentos (Media)
1	1.3	10	1505.1
2	3.3	11	1606.2
3	7.9	12	3845.7
4	22	13	7608.2
5	26.1	14	21940.8
6	57.2	15	32776
7	158	16	58723
8	394.8	17	60020.8
9	1065.7	18	330555.7

Tabla 2.1: Resultado del experimento con la función de la tarea 1.

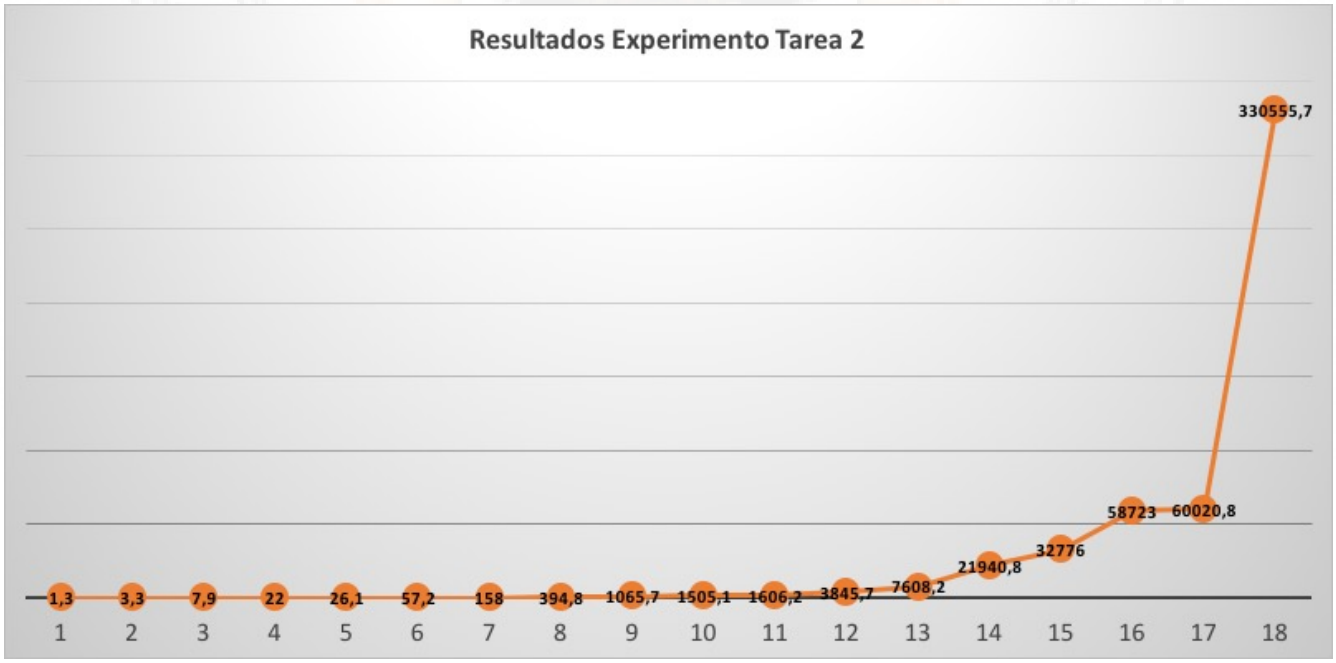

Figura 2.1: Resultados del experimento: valor de B frente al numero de intentos.

3. Tarea 3: Repetid la función anterior con el siguiente cambio: Se toma un primer valor aleatorio x y se va incrementando de 1 en 1 hasta obtener el hash requerido.

Para la resolución de ésta tarea, basta con modificar ligeramente la función de la tarea 1 para incorporar el cambio que se nos pide. En este caso, solo hace falta modificar la función principal `puzle_hash` de forma que el valor **x** que antes se creaba en cada iteración del bucle interno, ahora se cree **antes del bucle** y dentro del mismo únicamente lo que hará será **ir aumentando de 1 en 1 (suma bit a bit)**. Los cambios se pueden ver en el trozo de código a continuación.

```
def puzle_hash(msg, b):

intentos_totales = 0

for reps in range(0,10):
  #Creacion del nonce
  nonce = str(crear_nonce(int(N/4)))

  #Creacion del id, que es el msg concatenado con el nonce
  id = msg + nonce

  #Inicio del puzle hash: numero de intentos y condicion de parada
  intentos_actuales, encontrado = 0, False
  hash = ""

  #Se crea el valor x ANTES de empezar el bucle principal
  x = str(crear_nonce(int(N/4)))

  while(not encontrado):
    #Aumentamos el numero de intentos y creamos id||X
    intentos_actuales+=1

    #Ahora x = x+1, donde x es el nonce creado al antes de iniciar el bucle.
    x = hex(int(x,escala)+int('1',escala))[2:]

    test = id + x
    hash = str(hashlib.sha256(test.encode('utf-8')).hexdigest())
    encontrado = comprobar_ceros(hash) >= b

  intentos_totales += intentos_actuales

media_intentos = intentos_totales/10

print("------------------------------------------------------------")
print("Resultados Medios")
print("Valor B , Intentos : %i , %f" % (b, media_intentos))
print("------------------------------------------------------------")
```

4. Tarea 4: Calculad una nueva tabla/gráfica similar a la obtenida en el punto 2 pero con la función construida en 3.

Las tablas que se muestran a continuación reflejan los resultados del experimento con las modificaciones pertinentes. Cabe destacar que se han realizado bajo las mismas condiciones de aleatoriedad, fijando la semilla al mismo valor que en el experimento anterior, pero en este caso se han obtenido **mejoras notables** en cuanto a **número de intentos** para **los valores de b por encima de 8**, pero sobre todo en cuanto a **tiempo de ejecución**, con una reducción considerable para $b \in [1, 18]$, debido principalmente a que por cada intento no se realiza un cálculo aleatorio de la cadena, sino una simple suma. Mostramos los resultados a continuación.

Valor de b	Intentos (Media)	Valor de b	Intentos (Media)
1	2.2	10	943.9
2	2.9	11	1561.7
3	5.4	12	3196.5
4	17	13	7480.4
5	29.8	14	19041.2
6	58.7	15	26561.6
7	84.1	16	58200
8	225.5	17	77425.8
9	342.77	18	429114.5

Tabla 4.1: Resultado del experimento con la función de la tarea 3.

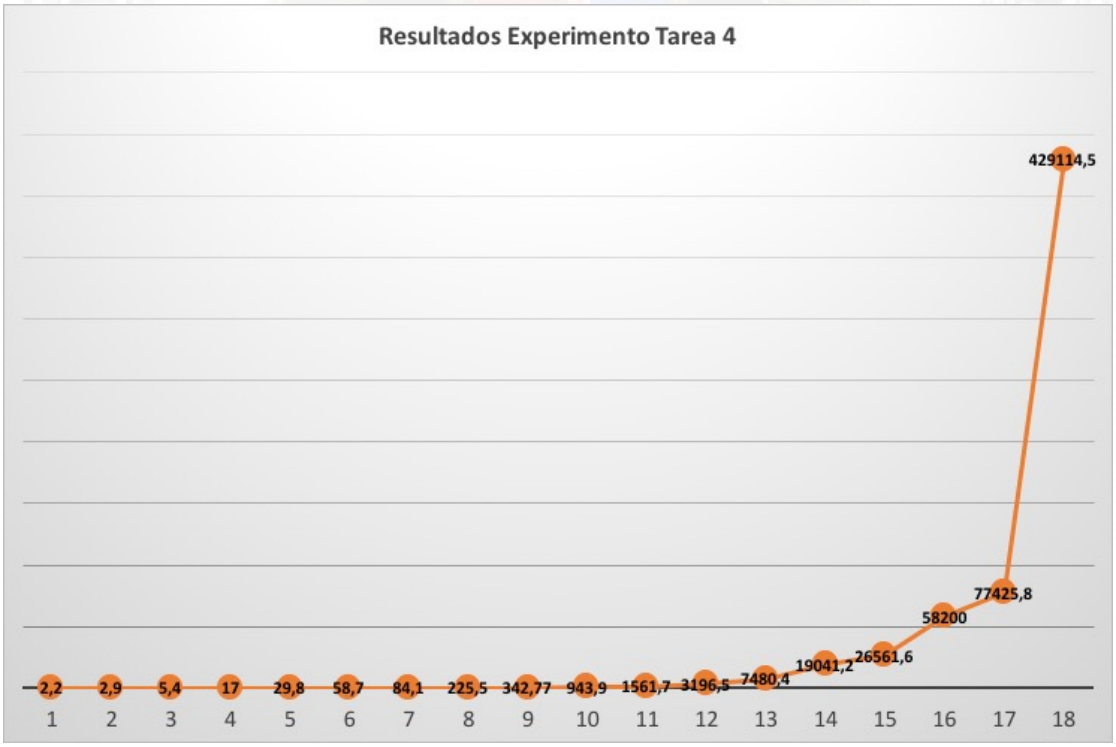

Figura 4.1: Resultados del experimento: valor de B frente al numero de intentos.

5. Anexo 1. Script del código de la tarea 1: puzleHash.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 16 17:19:27 2017
@author: Arthur18
"""

#Practica 5 SPSI: Puzles Hash
#Funcion de la tarea 1

#Librerias para numeros aleatorios, funciones Hash
#y manejo de cadenas hexadecimales
import random
import hashlib
import string

#Variables globales a utilizar en la funcion
random.seed(18)
N = 64
maximum_b = 19
texto = """En un lugar de la Mancha, de cuyo nombre no quiero acordar-me,
no ha mucho tiempo que vivia un hidalgo de los de lanza en astillero,
adarga antigua, rocin flaco y galgo corredor."""
#Para la transformacion a binario y gestion de cadenas hexadecimales
hex_digits = string.hexdigits[0:16]
escala = 16
num_bits = 256

#Funcion para crear el nonce
def crear_nonce(tam=8):

  #Creamos una cadena hexadecimal con tam caracteres
  #Los caracteres se eligen de forma aleatoria dentro del
  #conjunto de caracteres hexadecimales
  return ''.join(random.choice(hex_digits) for _ in range(tam))

#Funcion para comprobar la cantidad de ceros al inicio del hash
def comprobar_ceros(hash):

  hash_bin = bin(int(hash, escala))[2:].zfill(num_bits)
  count = 0

  for i in range(0,len(hash_bin)):
    if(hash_bin[i] == '0'):
      count+=1
    else:
      break
```

```
  return count

#Funcion que recibe como parametro un texto y un valor b
def puzle_hash(msg, b):

  intentos_totales = 0

  for reps in range(0,10):
    #Creacion del nonce
    nonce = str(crear_nonce(int(N/4)))

    #Creacion del id, que es el msg concatenado con el nonce
    id = msg + nonce

    #Inicio del puzle hash: numero de intentos y condicion de parada
    intentos_actuales, encontrado = 0, False
    x,hash = "",""

    while(not encontrado):
      #Aumentamos el numero de intentos y creamos id||X
      intentos_actuales+=1
      x = str(crear_nonce(int(N/4)))
      test = id + x
      hash = str(hashlib.sha256(test.encode('utf-8')).hexdigest())
      encontrado = comprobar_ceros(hash) >= b

    intentos_totales += intentos_actuales
    #  print("Intentos " + str(intentos_actuales))
    #  print("Intentos totales " + str(intentos_totales))
    #  print("_____________________________________________________________")
    #  print("Puzle Hash para B = " + str(b))
    #  print("Cadena X -> " + x)
    #  print("Hash -> " + hash)
    #  print("Intentos " + str(intentos_actuales))
    #  print("-------------------------------------------------------------")

  media_intentos = intentos_totales/10

  print("_____________________________________________________________")
  print("Resultados Medios")
  print("Valor B , Intentos : %i , %f" % (b, media_intentos))
  print("-------------------------------------------------------------")

#Ahora realizamos este procedimiento para valores de B que van desde
# 1 hasta el mayor valor posible
for valor_b in range(1, maximum_b):
  puzle_hash(texto, valor_b)
```

6. Anexo 2. Script del código de la tarea 3: puzleHashT3.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Dec 16 21:30:44 2017
@author: Arthur18
"""
#Practica 5 SPSI: Puzles Hash
#Funcion de la tarea 1

#Librerias para numeros aleatorios, funciones Hash
#y manejo de cadenas hexadecimales
import random
import hashlib
import string

#Variables globales a utilizar en la funcion
random.seed(18)
N = 64
maximum_b = 19
texto = """En un lugar de la Mancha, de cuyo nombre no quiero acordar-me,
no ha mucho tiempo que vivia un hidalgo de los de lanza en astillero,
adarga antigua, rocin flaco y galgo corredor."""
#Para la transformacion a binario y gestion de cadenas hexadecimales
hex_digits = string.hexdigits[0:16]
escala = 16
num_bits = 256

#Funcion para crear el nonce
def crear_nonce(tam=8):

  #Creamos una cadena hexadecimal con tam caracteres
  #Los caracteres se eligen de forma aleatoria dentro del
  #conjunto de caracteres hexadecimales
  return ''.join(random.choice(hex_digits) for _ in range(tam))

#Funcion para comprobar la cantidad de ceros al inicio del hash
def comprobar_ceros(hash):

  hash_bin = bin(int(hash, escala))[2:].zfill(num_bits)
  count = 0

  for i in range(0,len(hash_bin)):
    if(hash_bin[i] == '0'):
      count+=1
    else:
      break
```

```
  return count

#Funcion que recibe como parametro un texto y un valor b
def puzle_hash(msg, b):

  intentos_totales = 0

  for reps in range(0,10):
    #Creacion del nonce
    nonce = str(crear_nonce(int(N/4)))

    #Creacion del id, que es el msg concatenado con el nonce
    id = msg + nonce

    #Inicio del puzle hash: numero de intentos y condicion de parada
    intentos_actuales, encontrado = 0, False
    hash = ""

    #Se crea el valor x ANTES de empezar el bucle principal
    x = str(crear_nonce(int(N/4)))

    while(not encontrado):
      #Aumentamos el numero de intentos y creamos id||X
      intentos_actuales+=1

      #Ahora x = x+1, donde x es el nonce creado al antes de iniciar el bucle.
      x = hex(int(x,escala)+int('1',escala))[2:]
      test = id + x
      hash = str(hashlib.sha256(test.encode('utf-8')).hexdigest())
      encontrado = comprobar_ceros(hash) >= b

    intentos_totales += intentos_actuales
    #  print("Intentos " + str(intentos_actuales))
    #  print("Intentos totales " + str(intentos_totales))
    #  print("------------------------------------------------------------")
    #  print("Puzle Hash para B = " + str(b))
    #  print("Cadena X -> " + x)
    #  print("Hash -> " + hash)
    #  print("Intentos " + str(intentos_actuales))
    #  print("------------------------------------------------------------")

  media_intentos = intentos_totales/10
  print("------------------------------------------------------------")
  print("Resultados Medios")
  print("Valor B , Intentos : %i , %f" % (b, media_intentos))
  print("------------------------------------------------------------")

#Ahora realizamos este procedimiento para valores de B que van desde
# 1 hasta el mayor valor posible
for valor_b in range(1, maximum_b):
  puzle_hash(texto, valor_b)
```