Codificação ArDVK-64

Introdução

Trata-se de um método de codificação textual baseado no conhecido algoritmo Base 64, com algumas mudanças. Assim como o Base 64, essa codificação pode ser utilizada para transmitir dados textuais comuns por meios de transmissão que lidam apenas com textos de caracteres simples, pois sua saida não possui espaços nem pontuações e é limitada em caracteres especiais.

É constituído por 64 caracteres ([A-Z],[a-z],[0-9], "/" e "+"), tal qual o algoritmo Base 64, mas esse conjunto de possibilidades não constitui uma tabela fixa de correspondência, podendo ser aleatorizado de acordo com o desejo do usuário. Como esse algoritmo não necessita de uma senha para codificar uma mensagem, o próprio conjunto de possilidades pode ser considerado uma espécie de credencial, visto que ele pode ser variável.

Codificação

Começamos definindo o nosso conjunto de possibilidades (pos[ ]), que será uma *string* de 64 elementos compostos por letras maiúsculas, minúsculas, dígitos e pelos caracteres especiais ‘/’ e ‘+’. O ideal é que esse conjunto seja aleatorizado pelo usuário codificador e depois seja passado ao usuário descodificador, mas para fins didáticos iremos utilizar os caracteres ordenados:

upper = ‘ABCDEFGHIJKLMNOPQRSTUVWXYZ’

lower = ‘abcdefghijklmnopqrstuvwxyz’

numbers = ‘0123456789’

special = ‘/+’

pos = ***upper*** + ***lower*** + ***number*** + ***special***

Com o conjunto de possibilidades definido, podemos receber a entrada:

Ola

A primeira coisa a fazer é codificar a entrada com os valores hexadecimais da tabela ASCII:

4f6c61

Agora convetemos cada caractere para o seu valor binário correspondente da tabela ASCII:

4	f	6	c	6	1
00110100	01100110	00110110	01100011	00110110	00110001

Tomando por ***n*** a quantidade de bits do texto em binário, temos que obedecer a condição:

$$n \bmod 6 = 0$$

Isto é, o tamanho da string binária deve ser divisível por 6. Se não for, adicionamos uma quantidade ***x*** de 0's (no final) até que essa condição se satisfaça. Se de inicío, n for divisível por 6, podemos continuar o processo sem realizar nenhuma mudança nos bits. Após isso, iremos dividir os bytes em grupos de 6 bits, para que os valores representados por eles após a conversão para decimal possa ter um valor máximo de 63 ($2^n - 1$):

001101	000110	011000	110110	011000	110011	011000	110001

Por fim, relizamos a conversão de cada sexteto para um valor decimal ***index***, que será o índice para o caractere codificado dentro do conjunto de possibilidades:

001101	000110	011000	110110	011000	110011	011000	110001
13	6	24	54	24	51	24	49
pos[13]	pos[6]	pos[24]	pos[54]	pos[24]	pos[51]	pos[24]	pos[49]

O último caractere que a saída recebe, na verdade é a quantidade de 0's que foi adicinada para que **n mod 6 = 0**. Caso a quantidade seja inicialmente divisível por 6, o último caractere da saída será '0', como acontece no nosso exemplo.

saida = pos[13] + pos[6] + pos[24] + ... + pos[49] + '0'

saida = NGY2YzYx0

Por fim, invertemos a saída para compor a saída final:

saida = **0xYzY2YGN**

ArDVK-64 vs. Base 64

Entrada	Ola
ArDVK-64	0xYzY2YGN
Base 64	T2xh

Decodificação

Começamos descartando o primeiro digíto da entrada, que na verdade é a quantidade de 0's que devemos retirar da sequência de bits para que ela volte à forma inicial. Depois, invertemos a *string*:

0xYzY2YGN
NGY2YzYx

Agora podemos procurar cada caractere da entrada em pos[ ] e codificar o índice ***index*** para binário, mas forçando 6 bits:

pos[13]	pos[6]	pos[24]	pos[54]	pos[24]	pos[51]	pos[24]	pos[49]
13	6	24	54	24	51	24	49
001101	000110	011000	110110	011000	110011	011000	110001

Caso o primeiro elemento da entrada codificada – o que foi retirado – seja **maior que 0**, devemos retirar essa quantidade de 0's do fim da sequência de bits. Nesse caso, como não houve a necessidade de nenhuma modificação desse tipo durante a codificação (**n mod 6 era 0**), não devemos retirar nenhum bit na hora da decodificação. Após verificar a necessidade de retirada de bits, podemos agrupar novamente entrada binária em bytes e **convertê-los para caracteteres com base na tabela ASCII**:

00110100	01100110	00110110	01100011	00110110	00110001
4	f	6	c	6	1

Por fim, basta converter o código hexadecimal obtido na coversão dos bytes de volta para *string*:

4f6c61 = Ola

Implementação em Python

```
#-----------ArDVK-64--------------
#---Autor: Érick Oliveira  -------
#---Data:  13/09/2018      -------
#---Copyright: Cripto S.A  -------
#--------------------------------

import binascii

global pos #Conjunto de possibilidades

#---Definição do conjunto de possibilidades----
upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lower = 'abcdefghijklmnopqrstuvwxyz'
numbers = '0123456789'
special = '/+'
pos = upper + lower + numbers + special
#-----------------------------------------

class ArDVK64():

      def to_bin(num): #Converte um número para binário (força 8 bits)
            temp = str(bin(num))
            binary = ''
            if num >= 47 and num <= 57:
                  binary += '0'
            if num == 43:
                  binary += '0'
            for bit in temp:
                  if bit != 'b':
                        binary += bit
            return binary

      def separate(binary, length): #Separa a entrada binária em blocos
de 'length' bits
            separated = ''
            cont = 0
            for bit in binary:
                  separated += bit
                  cont += 1
                  if cont == length:
                        separated += ' '
                        cont = 0
            return separated

      def to_dec(string): #Converte binario para decimal
            soma = 0
            exp = len(string)-1
            for i in range(0,len(string)):
                  if string[i] == '1':
                        soma += 2 ** exp
                  exp -= 1
```

```
            return soma

      def encode(ent): #Realiza a codificação
            binary = ''

            #convertendo para hexadecimal
            ent = ent.encode('utf-8')
            ent = ent.hex()

            for letter in ent:
                  aux = ord(letter) #Correspondente decimal ASCII
                  binary += ArDVK64.to_bin(aux)

            if len(binary) % 6 == 0: #Tamanho da entrada é divisível por
6
                  binary = ArDVK64.separate(binary, 6)
                  dif = 0 #Quantidade de zeros a serem acrescentados
            else: #Tamanho da entrada não é divisível por 6
                  aux = len(binary)
                  while aux % 6 != 0:
                        aux += 1
                  dif = aux-len(binary) #Quantidade de zeros a serem
acrescentados
                  for i in range(0,dif): #acrescenta a quantidade de
zeros pra ser divisível por 6
                        binary += '0'
                  binary = ArDVK64.separate(binary, 6)

            #colocando o valor em decimal de cada bloco em um vetor
            temp = ''
            vetor = []
            for bit in binary:
                  if bit != ' ':
                        temp += bit
                  else:
                        vetor.append(ArDVK64.to_dec(temp))
                        temp = ''

            #Saída recebe valor correspondente na tabela padrão para
cada valor do vetor
            saida = ''
            for i in vetor:
                  saida += pos[i]
            saida += str(dif) #Adiciona número de zeros adicionados
            saida = saida[::-1] #Invertendo saída

            return saida

      def reduxTo6(string): #reduz n bits para 6 bits
            hexa = ''
            if len(string) == 6:
                  return string
            if len(string) == 8:
```

```
                for bit in range(2,len(string)):
                hexa += string[bit]
          elif len(string) == 7:
                for bit in range(1,len(string)):
                      hexa += string[bit]
          if len(string) < 6:
                dif = 6-len(string)
                for i in range(0,dif):
                      hexa += '0'
                for bit in string:
                      hexa += bit
          return hexa

     def filter(text): #remove ' e b de uma string convertida
          filtered = ''
          for index in range(1,len(text)-1):
                if text[index] != "'":
                      filtered += text[index]
                else:
                      continue
          return filtered

     def decode(pre_ent): #Realiza a decodificação
          binary = ''
          ent = ''
          dif = int(pre_ent[0]) #Número de zeros acrescentados

          for index in range(1,len(pre_ent)): #Remove o primeiro
elemento
                ent += pre_ent[index]
          ent = ent[::-1]

          for char in ent:
                index = pos.find(char) #Acha a posicao de cada
caractere de ent em pos
                reducted = ArDVK64.reduxTo6(ArDVK64.to_bin(index))
#converte de volta para binário e reduz para 6 bits
                binary += reducted

          original = ''
          for i in range(0,len(binary)-dif): #retirando os zeros
acrescentados
                original += binary[i]

          plain_text = ''
          temp = ''
          cont = 0
          for bit in binary: #converte binario para string
                temp += bit
                cont += 1
                if cont == 8:
                      aux = ArDVK64.to_dec(temp)
                      plain_text += chr(aux)
```

```
                        cont = 0
                        temp = ''

            plain_text = str(binascii.unhexlify(plain_text))
            plain_text = ArDVK64.filter(plain_text)

            return plain_text

#Exemplo de Codificação/Decodificação:
encoded = ArDVK64.encode('Ola')
print(encoded)
decoded = ArDVK64.decode(encoded)
print(decoded)
```

Complexidade para Quebra

Considerando que um conjunto de possibilidades arbitrário seja usado como chave:

1 em 64! = **1 em**
126886932185884164103433389335161480802865516174545192198801894375214704230400000000000000 posibilidades

Computacionalmente inviável

Repositório Oficial

https://github.com/ErickOliveiraT/ArDVK-64