# cryptopals

February 27, 2020

## 0.1 TAG CRIPTOGRAFIA - cryptopals.com

**Por: Yuri Medeiros da silva**

**Responsável: Sidney gris**

[0]:
```
# !/usr/bin/env python3.6
import textwrap
string = "49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736\
f6e6f7573206d757368726f6f6d"
```

Antes de começarmos temos que definidir algumas funções de validaçoes para a primeira challenge :

```
validator:  confere se a resposta final está correta
validator16to2: confere se a conversão de hexa pra binário está certa
```

[0]:
```
'''
https://cryptopals.com/sets/1/challenges/1
hex -> base64
hex:␣
 ↪49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e6f\
7573206d757368726f6f6d
output : SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
'''
def validator(output):
    b64string = "SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hy\
    b29t"
    if str(output) == str(b64string) :
        print("FFrom hex to b64 with success!")
        return True
    else:
        print("Failed!")
        print("out:" + str(output) + "\n")
        print("expected:" + str(b64string))
    return False
```

```
def validator16to2(output):
    binstring = "01001001001001110110110100100000011010110110100101101100011\
    11000110100101101110011001110010000001111001011011110111010101110010001\
    00001100010011100100110000101101001011011100010000001101100011010010110\
    11011001010010000001100001001000000111000001101111011010010111001101101\
    10110111001101111011101010111001100100000011011010111010101110011011010\
    01110010011011110110111101101101"
    if str(output) == str(binstring) :
        print("FFrom hex to bin with success!")
        return True
    else:
        print("Failed!")
        print("out:    " + str(output) + "\n")
        print("expected:    " + binstring)
    return False
```

A primeira challenge do cryptopals (https://cryptopals.com/sets/1/challenges/1) tem o nome "Convert hexConvert hex to base64 to base64". Basicamente recebemos como input: uma string em hexadecimal output: a string anterior em base64

Mas antes de partírmos pro código é necessário saber como *essas coisas* de base64 funcionam

### 0.1.1 o que é base64 ?

```
É uma forma de encoding usada principalmente para transmissão de
dados com propósito de manter a 'integridade'. Ou seja, aquilo que
enviamos é aquilo que chegará do outro lado.
```

Base64 está longe de ser um protocolo considerado seguro, qualquer um com acesso a um computador conseguiria decodificar a mensagem que ele encoda. Ainda mais, qualquer um que soubesse como o algorítmo funciona conseguiria decodificar a mensagem transmitida na mão - por mais que isso possa demorar algum tempo. Entretanto, ela continua sendo utilizado até hoje pois mantém a integridade dos dados transmitidos.

**Ok, mas porque nao enviar os bytes crus do que queremos transmitir ao invés de transformar em outra mensagem ?** Tranformar os bytes crus em 'letras' é importante porque, ao enviar uma imagem ou binario pra outra pessoa, dependendo do protocolo que esteja sendo utilizado podemos ter alguns conflitos entre o arquivo enviado e o protocolo (conflitos do tipo: caracteres inválidos, entre outros), ou seja, não teríamos certeza se o que enviamos chega da forma que queremos do outro lado.

A citação a baixo explicita muito bem o porque de usarmos base64, achei importante colocá-la aqui em sua forma original.

*To figuratively understand why Base64 was invented, imagine that during a phone call Alice wants to send an image to Bob. The first problem is that she cannot simply describe how the image looks, because Bob needs an exact copy. In this case, Alice may convert the image into the binary system and dictate to Bob the binary digits (bits), after that he will be able to convert them back to the original image. The second problem is that the tariffs for phone calls are too expensive and dictate each byte as 8 binary digits will last too long. To reduce costs, Alice and Bob agree to use a more efficient data transfer method by using a special alphabet, which replaces every "six digits" with one "letter".*

**Como esse encoding funciona ?** Bytes são comumente separados de 8 em 8 bits, ao utilizarmos base64 iremos repartir esses bytes de 6 em 6 bits, o que faz com que cada 3 bytes anteriores formem 4 novos bytes ($3_8 = 4_6$). Dado um conjunto de bits, por exemplo :

```
01111001 01110101 01110010 01101001

Dividimos de 6 em 6 bits
011110 010111 010101 110010 011010 01

Mas, nos deparamos com um problema, o ultimo conjunto nao está completo.
Para completarmos precisamos aplicar o que chamamos de padding (explicado
mais a baixo)
Agora completamos com 0
011110 010111 010101 110010 011010 010000

Agora convertemos cada um desses bytes em seu numero decimal correspondente e
depois consultamos sua posição na tabela do base64
30     23     21     50     26     16
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ [<-tabela]
começamos a contar do A, de forma que o caracter na posição 30 é o e, 23 = X

convertido teremos :
e      x      v       y      a     q    =   =
(os == serão explicados mais abaixo)
por fim, temos que  :
    yuri virou exvyaq==
```

**Padding** Como a ultima "casa" no exemplo nao está completa, precisamos completá-la e chamamos isso de padding. Para isso acrescentamos 0 (zeros) depois do ultimo bit até termos 6 bits. Mas e os '=' '==' que sempre aparecem ? Como em base64 cada 3 bytes viram 4($3_8 = 4_6 = 24$) precisamos que a cadeia encodada tenha tamanho de 24 bits, caso nao tenha completamos com '=', se faltar um byte de 6 completamos com um =, se faltar dois bytes de 6 completamos com 2 '==', se tivermos algo maior que 24 bits, completamos até o proximo multiplo de 24, 48,... (em alguns sistemas vemos implementações um pouco diferentes de como o padding com o = funciona, de fato, hoje em dia, algumas vezes ele é ignoradado, a 'antiga' necessidade dele nasceu pois ser multiplo de 2 facilita o acesso na memória e na paginação dos bits.

====

obs:

digo que varia pq em varios sites ao converter yuri pra base64 tenho o output "eXVyaQ==", mas utilizando o base64 do linux tenho só "eXVyaQ="

https://en.wikipedia.org/wiki/Base64#Output_padding

Essa é a primeira parte do nosso código que entrará em execução, dado a string em hexadecimal convertemos ela para binario.

### 0.1.2 O que é hexadecimal ?

É um sistema numerico que "conta" até o 16, e para isso ele utiliza, também, alguns simbolos.

```
1 2 3 4 5 6 7 8 9 A B C D E F
```

### 0.1.3 Porque utilizamos hexadecimal ?

Principalmente pelo fator compressão - apenas com dois digitos hexadecimais conseguimos expressar de 0-255, e, também, pela compatibilidade com "binário"/ base2, visto que 2^4 é 16, o ponto aqui é que :

(caso a imagem nao renderize: https://miro.medium.com/max/652/1*1NGffiR_VdV4F7DKSyJ70A.png)

Podemos ver que o binario e o decimal nunca se casam, já o binario e o hexa sim (eles se casam no 16 e no 255), ou seja, cada 4 bits do binario ( 1000 = 16 0x10) representa um bit no hexadecimal, e esse alinhamento nos ajuda a representar as coisas, endereços de memoria ..., melhorando a forma de acessos, paginacao ...

(https://medium.com/@savas/why-do-we-use-hexadecimal-d6d80b56f026 )

[0]:
```python
# 0000
def hex2bin(hex_str):
    int_ = int(hex_str,16)
    bin_str = ""
    # poderiamos usar a funcao a baixo, mas codamos na mao a conversao
    # de hexa -> bin -->> bin_str2 = str(bin(int_).replace("0b", "0"))

    while int_ > 0:
        bin_str = str(int_ % 2) + bin_str
        int_ = int_ >> 1
    bin_str =  "0"*(4 - len(bin_str)) + bin_str
    # por algum motivo ele tira o 0 da frente,
    # segundo um cara na internet isso também acontece com ruby

    bin_str =  "0" + bin_str

    return bin_str
```

[0]:
```python
def challenge1():
    binstr = hex2bin("49276d206b696c6c696e6720796f757220627261696e206c696b6520\
    6120706f69736f6e6f7573206d757368726f6f6d")
    ret = b64encode(binstr)
    validator(ret)
    print(ret)

challenge1()
```

```
FFrom hex to b64 with success!
SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
```

## 0.2 Challenge2

https://cryptopals.com/sets/1/challenges/2

Dado duas entradas de mesmo tamanho fazer o xor entre elas

```
str1 = 1c0111001f010100061a024b53535009181c
str2 = 686974207468652062756c6c277320657965
output = 746865206b696420646f6e277420706c6179
```

[0]:
```
def validatorChallenge2(output):
    expected = "746865206b696420646f6e277420706c6179"
    if str(output) == expected:
        print("Funcionou")
        return True
    else:
        print("==ERRO \n sua saída:"+ str(output) + "\n saida esperada:" +
              str(expected))
        return False
```

**O que é um XOR?** Xor é uma operação lógica que, dado duas entradas A e B, ela retorna o valor verdadeiro (1) caso uma entrada tenha o valor diferente da outra.

Tabela verdade da operação :

A | B | OUTPUT
0 | 1 | 1
0 | 0 | 0
1 | 1 | 0
1 | 0 | 1

**Porque essa operação é tão usada na criptografia ?** A primeira razão é que Xor é uma função que se aplicada nela mesmo ela volta pro valor original, ou seja, se tivermos uma string A (string) e uma chave B, e aplicamos o xor de B em A teremos a string encriptada C. Como xor é uma função que aplicada a ela mesmo ela retorna pro valor inicial, ao aplicarmos B xor C teremos A.

A segunda razão é que é uma funcao que não *perde informação*. O que isso quer dizer, se utilizassemos um 'and' na nossa string A com B, poderiamos fazer um bruteforce com uma sequencia de 1's, e saberiamos onde todos os 1s estao, porque na tabela logica do and, só da 1 se ambos os lados forem 1, com o OR os 0's seriam leakados (bruteforce nos 0s). Já com o XOR, para termos um output 1 podemos ter como entrada 0 ou 1 e para ter o output 0 também podemos ter duas entradas possíveis, dificultando um possível bruteforce.

https://stackoverflow.com/questions/1379952/why-is-xor-used-in-cryptography

[0]:
```
'''
essa funcao recebe duas 'strings binarias'
'''
def xor2xor(in1,in2):
    tam = len(in1) # tem o mesmo tamanho os dois
    ans = ""
    for i in range(0, len(in1)):
```

```
        # int pra converter de char/string pra int
        ans += str(int(in1[i]) ^ int(in2[i]))
    return ans
```

[0]:
```
str1 = "1c0111001f010100061a024b53535009181c"
str2 = "686974207468652062756c6c277320657965"

def challenge2():
    ans = ""
    for i in range(0,len(str1)):
        #
        v1 = hex2bin(str1[i])
        v2 = hex2bin(str2[i])
        ans += hex(int(xor2xor(v1,v2),2))

    ans = ans.replace('0x', '')
    validatorChallenge2(ans)
    return ans

print(challenge2())
#xored = int(xor2xor(vim,vim2),2)
```

```
Funcionou
746865206b696420646f6e277420706c6179
```

### 0.3 challenge 5

Para última challenge da TAG foi feita a challenge 5 ( https://cryptopals.com/sets/1/challenges/5 ) Basicamente ela faz uso dos últimos códigos e consiste em pegar uma string :

```
Burning 'em, if you ain't quick and nimble
I go crazy when I hear a cymbal
```

e aplicar uma chave pra encriptarmos ela com o xor. no caso a chave é *ICE*.
Como a chave é menos que a string completa, vamos aplicando ICE até o final repetidamente.

```
ICEICEICEICEIC...
letra por letra.
```

[0]:
```
str1 = b"Burning 'em, if you ain't quick and nimble\nI go crazy when I hear\
 a cymbal"
key  = b"ICE"

# zerofill deixa os dois bytes, o da chave e o da string do mesmo tamanho .
# ele coloca zeros na frente até terem o mesmo tamanho
def zerofill(st,l):
    return '0'*(l - len(st)) + st
```

```python
# esse xor2xor2 eh a mesma coisa que o xor2xor, eu so mudei ele porque estava
# tendo problemas e achei que poderia ser ele
# e,claro, ele usa list comprehension
# como gostei da solucao com listcomprehension resolvei deixar
def xor2xor2(in1,in2):
    '''
    tam = len(in1) # tem o mesmo tamanho os dois
    ans = ""
    for i in range(0, len(in1)):
        ans += str(int(in1[i]) ^ int(in2[i]))
    '''
    return ([int(in1[i]) ^ int(in2[i]) for i in range(len(in1))])
def ch5(s,key):
    i_key = 0 # index of key, max = 2... 0,1,2
    keyBin = []

    for byte in key :
        keyBin.append(format(byte,'b'))

    ans =  ''
    for byte in s:
        bs = format(byte,'b')
        if len(bs) < len(keyBin[i_key]):
            bs = zerofill(bs,len(keyBin[i_key]))

        xx2 = xor2xor2(bs,keyBin[i_key])
        bits_string=  ''.join(map(str, xx2))

        # esse if so funciona no caso desse exercicio
        # por alguma razao, como boa parte das coisas tentei fazer handmade,
        # por alguma razao ao trabalhar com os bits
        # o python ao inves de colocar 0x0b 0x0a e 0x0c, ele coloca apenas
        # 0xa 0xb 0xc.
        # e esses bits que ele comeu fizeram falta na  resposta final.
        # O codigo abaixo foi um remendo pra isso ai.

        value = hex(int(bits_string,2))
        carelist = ['0xa','0xb','0xc','0xd','0xe','0xf']
        if value in carelist:
            tmp = value.replace('0x', '')
            value = '0x0' + tmp

        ans += value
        # seta o index da key
        if (i_key ) == 2 :
            i_key = 0
```

```
        else :
            i_key += 1
    return ans

v = ch5(str1,key).replace('0x', '')
print(v)

expected = "0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324\
272765272a282b2f20430a652e2c652a3124333a653e2b2027630c692b20283165286326302e27\
282f"
assert(v == expected)

# A minha resposta tinha 3 bits de diferença da original, e eu passei boas 3
# horas tentando entender o porque
# basicamente, como eu fiz tudo handmade - ou quase tudo- o python ao inves de
# colocar  e 0x0b 0x0a e 0x0c, ele coloca apenas 0xa 0xb 0xc, e os 3 zeros que
# ele 'come' são os que estavam faltando. Por isso fiz a solucao (gambiarra) do
# carelist
```

```
0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a2622632427276527
2a282b2f20430a652e2c652a3124333a653e2b2027630c692b20283165286326302e27282f
```

[0]: