Aluno: José Victor Medeiros Thomé da Silva

Relatório da Tag de Criptografia

1. Introdução

Esse relatório tem como objetivo detalhar as soluções usadas nos três desafios propostos nessa tag. Essas soluções foram elaboradas usando a linguagem Python na versão 3 por conta de toda conveniência trazida pela mesma. No entanto, por motivos de aprendizagem, algumas operações foram feitas do início, isto é, sem usar uma função já pronta da linguagem, como foi o caso do desafio 1, por exemplo.

2. Desafio 1 – Converter para Base 64 um hexadecimal

Nesse desafio, a solução proposta é uma classe chamada Base_64 que possui dois métodos: `table_generator()` e `hexa_converter()`. Ao ser instanciada, é gerada automaticamente a tabela de conversão usada chamando o método `table_generator()`, como mostrado no algoritmo abaixo:

```
1. def __init__(self):
2.     ''' Inicializa o objeto tratador gerando a tabela
3.         da base 64 a ser usada nas operações. '''
4.
5.     self.table_generator()
```

O método `table_generator()` gera uma tabela de 64 caracteres ([a-z], [A-Z], [0,9], ’/’ e ‘+’) através da conversão de seus formatos decimais da tabela ascii para o formato caractere usando a função `chr()`.

```
1. def table_generator(self):
2.     ''' Gera a tabela de caracteres da base 64. '''
3.
4.     upper_a = 65
5.     upper_case_letters = [ chr(upper_a + i) for i in
       range(26) ]
6.
7.     lower_a = 97
8.     lower_case_letters = [ chr(lower_a + i) for i in
       range(26) ]
9.
10.    from_0_to_9 = [ str(i) for i in range(10) ]
11.    last_ones = ['+', '/']
12.
13.    self.char_table = upper_case_letters +
       lower_case_letters + from_0_to_9 + last_ones
```

O método `hexa_converter()` é onde ocorre o processo de conversão para base 64. Abaixo está o código do método:

```
1.  def hexa_converter(self, message):
2.    ''' Converte a mensagem passada para a base 64. '''
3.
4.    # Converte a mensagem passada como parâmetro para o
5.    # formato binário.
6.    msg_binary_format = '0' + "{0:08b}".format(int(message, 16))
7.
8.    # Calcula quantos valores binários de 6 algarimsmos podem
9.    # ser obtidos para mapeamento.
10.   bin_format_len = len(msg_binary_format)
11.   total_indexes = int(bin_format_len / 6)
12.
13.   # Insere na conversão os valores da tabela obtidos pelo
14.   # mapeamento da mensagem binária.
15.   base_64_msg = ''
16.   for i in range(total_indexes):
17.      table_index = int(msg_binary_format[6*i : 6*(i+1)], 2)
18.      base_64_msg=''.join([base_64_msg,self.char_table[table_in
       dex]])
19.
20.   return base_64_msg
```

Este método recebe como parâmetro uma string de uma mensagem no formato hexadecimal.

Essa mensagem primeiro é convertida para binário primeiro tornando a mesma um inteiro hexadecimal e passando para o método `format()`, que converte para o formato binário com a string "`{0:08b}`"; essa expressão transforma um número para o formato binário com 8 bits para cada algarismos, acrescentando 0s a esquerda caso seja necessário para completar essa quantidade de bits. No entanto, mesmo com esse formato de conversão foi necessário acrescentar "0" a frente dessa transformação no valor de `msg_binary_format` devido a ausência do mesmo na conversão do primeiro algarismo a direita, o que comprometeria o resultado final devido a esse caractere acabar tendo 7 bits no formato binário.

Como o objetivo é converter para Base 64, a mensagem convertida em binário precisa ser separada em seções de 6 bits para pegar os índices da tabela. Para isso, a variável `total_indexes` representa a quantidade dessas seções calculada ao dividir o tamanho da mensagem por 6.

Os valores da tabela necessários são obtidos extraindo seções de 6 bits da mensagem. usando as mesmas para mapeamento do índice do valor desejado e inserindo esse valor na string da nova mensagem. Essa operação é feita através de um loop no qual a quantidade de interações é o valor de `total_indexes`.

3. **Desafio 2 – Combinação de XOR entre dois buffers.**

Nessa solução a primeira coisa feita foi comparar o tamanho dos dois buffers, pois precisavam ter tamanhos para as operações serem feitas. Abaixo está o código da solução:

```
1. def fixed_xor(buff_1, buff_2):
2.   ''' Produz uma combinação XOR entre dois buffers '''
3.
4.   if(len(buff_1) != len(buff_2)):
5.     print("Os valores dos buffers não tem tamanho igual.")
6.     return -1
7.
8.   return str(hex(int(buff_1, 16) ^ int(buff_2, 16)))[2:]
```

Aqui, antes de fazer o XOR, as duas strings dos parâmetros são convertidas para inteiros no formato hexadecimal usando a função `int()` com o segundo parâmetro sendo 16. Como a operação XOR do operador `^` tem como retorno um inteiro decimal, a função `hex()` é usada para converter esse resultado para o formato hexadecimal antes de transformar em uma string com a função `str()`. O operador `[2:]` ao final de str() é para que o valor “0x” vindo por padrão no resultado de `hex()` não seja incluso na string de retorno da função `fixed_xor()`.

O operador ^ é o Bitwise Exclusive XOR, responsável por fazer o XOR bit a bit dos valores inteiros de buff_1 e buff_2. Como dito antes, o resultado dessa operação é um inteiro decimal, portanto é necessário fazer a conversão do mesmo para o formato inteiro desejado.

4. **Desafio 3 – Quebrando a Cifra XOR**

Nessa solução foi utilizada uma tabela de frequência de aparições dos caracteres da língua inglesa obtida no link https://en.wikipedia.org/wiki/Letter_frequency. A ideia usada foi realizar a decriptação usando como chave todas as letras dos intervalos [a-z] e [A-Z], e verificar qual delas gerava uma versão descriptografada da mensagem que somava mais pontos de frequência da tabela de letras.

```
5.   ENGLISH_FREQUENCY_TABLE = {
6.       'a': 0.08167, 'b': 0.01492, 'c': 0.02782, 'd': 0.04253,
7.       'e': 0.12702, 'f': 0.02228, 'g': 0.02015, 'h': 0.06094,
8.       'i': 0.06094, 'j': 0.00153, 'k': 0.00772, 'l': 0.04025,
9.       'm': 0.02406, 'n': 0.06749, 'o': 0.07507, 'p': 0.01929,
10.      'q': 0.00095, 'r': 0.05987, 's': 0.06327, 't': 0.09056,
11.      'u': 0.02758, 'v': 0.00978, 'w': 0.02360, 'x': 0.00150,
12.      'y': 0.01974, 'z': 0.00074, ' ': 0.16000
13.  }
```

Antes de fazer a decriptação, a entrada primeiro é traduzida para o formato utf-8, já que a mensagem de parâmetro está inicialmente em hexadecimal, e depois é feita a operação Bitwise Exclusive XOR em cada caractere com a chave passada, que está no formato inteiro. Abaixo está o código de decriptação:

```
1. def xor_decrypt_msg(key, hex_msg):
2.   '''Decripta a mensagem passada (em hexadecimal) pela cifra
   XOR usando a chave passada.'''
3.
4.    msg = bytes.fromhex(hex_msg) # Converte de hexadecimal
   para string bytes ascii.
5.   decrypted = b''
6.
7.   # Realiza XOR em cada uma das letras da mensagem
   encriptada.
8.    for letter in msg:
9.       decrypted += bytes([letter ^ key])
10.
11.   return decrypted
```

O cálculo da pontuação da frase decriptada é feita somando a frequência de cada letra da mesma que é obtida na tabela de frequência de letras do inglês, que nessa solução é um dicionário nos quais as chaves são as próprias letras do alfabeto minúsculas. Abaixo está o código no qual a soma é feita:

```
1. def decrypt_score(decrypted):
2.   '''Calcula a taxa de pontuação da string decriptada com
   uma frase em inglês.'''
3.
4.    return sum([ENGLISH_FREQUENCY_TABLE.get(chr(letter), 0)
   for letter in decrypted.lower()])
```

A procura da chave é feita por todas as letras de ‘a’ a ‘z’ primeiro com as minúsculas e depois com as maiúsculas. Após essa etapa é verificada qual das duas chaves obtidas na avaliação dos intervalos é a que possui a maior pontuação para que seja feito o retorno da mesma na função junto com a mensagem descriptografada.

```
1. def bigger_key_score(first_key, last_key, encrypted_hex):
2.    '''Calcula e avalia qual chave dentro do intervalo
3.       [first_key, last_key] tem maior taxa de pontuação.'''
4.
5.     bigger = -1
6.     winner = None
7.     winner_decr = ''
8.
9.     for key in range(first_key, last_key + 1):
10.       msg = xor_decrypt_msg(key, encrypted_hex)
11.       score = decrypt_score(msg)
12.       if(bigger < score):
13.          bigger = score
14.          winner = chr(key)
15.          winner_decr = msg
```

```
16.
17.     return bigger, winner, winner_decr

18.   def detect_key(encrypted_hex):
19.     ''' Procura a chave que decripta a string encriptada em
            hexadecimal pela cifra de XOR. A dedução é feita
            pela soma das frequências das letras em
            ENGLISH_FREQUENCY_TABLE.'''
20.
21.     # Verifica pelas letras minúsculas de 'a' a 'z'.
22.         l_score,      l_winner,      l_winner_decr      =
  bigger_key_score(ord('a'), ord('z'), encrypted_hex)
23.       # Verifica pelas letras maiúsculas de 'A' a 'Z'
24.         u_score,      u_winner,      u_winner_decr      =
  bigger_key_score(ord('A'), ord('Z'), encrypted_hex)
25.
26.     if(u_score > l_score):
27.         return u_winner, u_winner_decr.decode("utf-8")
28.
29.     return l_winner, l_winner_decr.decode("utf-8")
```