TP00_P2

March 14, 2021

Solução Para o Trabalho Prático 0

Problema 02:

1. Usamos uma XOF e um PRG para fazer um pad que consiste em 2^N palavras de 64 bits.
2. Como pedido usamos a password como seed para o PRG.
3. Para cifrar ou decifrar fazemos um XOR como no caso do One Time Pad.

Observação: Além do que foi mencionado para o problema 1, neste problema não há autenticação.

Funcionamento dos Agentes:

- Imports do que é necessário para correr os Agentes, salientando o uso da package cryptography para o SHAKE.

[1]:
```
import getpass
import socket
import math
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
```

O Receiver e o Emitter usam as mesmas funções para a comunicação entre eles.

- A primeira é a shake que é responsável por criar uma string longa com 2^N * 8 (bytes) de tamanho para servir de pad.

[2]:
```
def shake(self, size, password):
    digest = hashes.Hash(hashes.SHAKE256(size), default_backend())
    digest.update(password)
    return digest.finalize()
```

Tendo o pad maior esta função separa o pad num array de palavras (neste caso longs).

[3]:
```
def split(self, pad):
    n = 8
    len_pad = len(pad)
    x = [pad[i:i + n] for i in range(0, len_pad, n)]
    return x
```

A função xor faz XOR do pad com a mensagem. É de salientar a variável word que conta em que palavra é que vamos para nos certificarmos que não repetimos a mesma palavra duas vezes.

[4]:
```
def xor(self, pad, message):
    size = len(message)
    xored = bytearray(size)
    word = self.counter
    position = 0
    for i in range(size):
        xored[i] = pad[word][position] ^ message[i]
        position += 1
        if position == 8:
            word += 1
            position = 0
    return xored
```

Execução do Agente Emitter:

1. Inicialmente perguntamos qual é o valor desejado para o N, este valor tem de coincidir em ambos os agentes.
2. Depois disso calculamos o número total de palavras a gerar com a password que é pedida logo a seguir. Tendo N e a password usamos a shake para gerar o pad e a split para dividir o pad em palavras.
3. Agora, para cada mensagem inserida verificamos se ela é maior que o pad e a nossa decisão é que embora fosse possível criar tantos pads quanto necessários para acomodar a mensagem nós decidimos que seria melhor impor um limite máximo.
4. A seguir verificamos se o tamanho da mensagem faz com que seja necessário criar um pad novo, e se for, o resto por utilizar deste pad é descartado e geramos um pad novo.
5. Depois disto é só fazer o XOR do pad com a mensagem e enviá-la, atualizando o counter tendo em conta o tamanho da mensagem.

É de salientar o uso da math.ceil nestes casos que faz com que não usemos as mesmas palavras duas vezes, por exemplo se usarmos 32 bits de uma palavra não vamos voltar a usar essa palavra, mas também não vamos usar o resto dos 32 bits.

[ ]:
```
n = int(input("N?\n"))
lim = pow(2, n)
self.establish_connection()
password = self.requestPassWord()
pad = self.shake(lim * 8, password)
pad_words = self.split(pad)
while True:
    msg = self.ask_message()
    msg_size = len(msg)
    if msg_size < int(lim*8):
        try:
            if (self.counter + math.ceil(msg_size / 8)) > lim:
                password = self.requestPassWord()
                pad = self.shake(lim * 8, password)
```

```
                pad_words = self.split(pad)
                self.counter = 0
            ciphertext = self.xor(pad_words, msg)
            self.counter += math.ceil(msg_size / 8)
            if len(ciphertext) > 0:
                self.conn.send(ciphertext)
                print('Mensagem Enviada')
            else:
                self.finish()
        except:
            print("Erro na Comunicacao\n")
    else:
        print("Mensagem demasiado grande\n")
        self.finish()
```

Execução do Agente Receiver:

O Receiver funciona exatamente como o Emitter, mas em vez de se inserir uma mensagem nós recebêmo-la pela socket.

[ ]:
```
Self.establish_connection()
i = 0
while i < 1:
    n = int(input("N?\n"))
    lim = pow(2, n)
    adrr, emitter = self.conn.accept()
    print("Ligado a: ", emitter)
    i += 1
    password = self.requestPassWord()
    pad = self.shake(lim * 8, password)
    pad_words = self.split(pad)
    while True:
        msg = adrr.recv(lim*8)
        msg_size = len(msg)
        if (self.counter + math.ceil(msg_size/8)) > lim:
            password = self.requestPassWord()
            pad = self.shake(lim * 8, password)
            pad_words = self.split(pad)
            self.counter = 0
        if not msg:
            print('Nenhuma Mensagem Recebida!\n')
            break
        else:
            cleantext = self.xor(pad_words, msg)
            self.counter += math.ceil(msg_size/8)
            print(cleantext.decode())
    self.finish(self.conn)
```

Alínea 2.C:

Não achamos que seja necessário fazer um grande número de medições nem outros tipos de estudos à lá Performance Engineer, mas sim podemos olhar de um ponto de vista mais removido para os dois algoritmos com que estamos a trabalhar.

O gargalo do segundo algoritmo é gerar o pad, enquanto que o gargalo do primeiro pode ser o facto de que por mensagem temos de gerar uma chave, o código de autenticação para ela e cifrar o criptograma para o caso do Emitter ou então no caso do Receiver que tem o trabalho adicional de autenticar a chave que foi recebida.

Tendo isto podemos imaginar facilmente que num cenário em que temos um grande número de mensagens trocadas e um N grande, a situação em que o segundo algoritmo passa pelo gargalo é rara o suficiente para o segundo algoritmo ser mais rápido que o segundo.

A diferença é clara o suficiente para se poder demonstrar que o segundo algoritmo é mais rápido on demand, algo que pode ser feito facilmente na apresentação.