CIC0201 - Segurança Computacional – 2022/1
Relatório trabalho 2

Exeução (Python3+):

python3 rsa.py

Descrição:

- **Gera um hash da mensagem original, decifra-a com rsa na base 64 e compara o hash da mensagem descriptografada com a original.**

- A **entrada** é o arquivo input.txt, a **saída** em out.txt imprime a comparação de hashes original/descriptografado e a **saída** e a mensagem descriptografada em desc.txt. Todas essas informações também são imprimidas no terminal.

- gera chaves RSA com teste de primalidade para execução (120 TRIAL_ROUNDS de iterações miller-rabin)
- Não foi possível implementar o AES nem o OAEP

Gerando as chaves RSA:

1. Generating the keys

1. Select two large prime numbers, x and y. The prime numbers need to be large so that they will be difficult for someone to figure out.
2. Calculate $n = x$ x y.
3. Calculate the ***totient*** function; $\phi(n) = (x-1)(y-1)$.
4. Select an integer e, such that e is ***co-prime*** to $\phi(n)$ and $1 < e < \phi(n)$. The pair of numbers (n, e) makes up the public key.

> **Note:** Two integers are co-prime if the only positive integer that divides them is 1.

5. Calculate d such that $e.d = 1 \; mod \; \phi(n)$.

d can be found using the ***extended euclidean algorithm***. The pair (n, d) makes up the private key.

E a chave[N,E] é pública e [N,D] é privada.

Geração de primos(https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/):
A parte fundamental da geração foi a uma tabela de primos conhecidos entre 2- 2000. Dessa forma: números randômicos são escolhidos, se forem divisíveis por aqueles da tabela- são descartados. Caso contrário, vão para o teste de miller-rabin.

```
7   # Primos pré gerados de 2-2000 (https://cis.temple.edu/~beigel/cis573/alizzy/prime-list.html)
8   primes_list = [
9       2,      3,      5,      7,     11,     13,     17,     19,     23,     29,
10      31,     37,     41,     43,     47,     53,     59,     61,     67,     71,
11      73,     79,     83,     89,     97,    101,    103,    107,    109,    113,
12      127,    131,    137,    139,    149,    151,    157,    163,    167,    173,
13      179,    181,    191,    193,    197,    199,    211,    223,    227,    229,
14      233,    239,    241,    251,    257,    263,    269,    271,    277,    281,
15      283,    293,    307,    311,    313,    317,    331,    337,    347,    349,
16      353,    359,    367,    373,    379,    383,    389,    397,    401,    409,
```

```
44  def get_possible_prime(n):
45      while True:
46          rand = random.getrandbits(n)
47          for prime in primes_list:
48              if rand % prime == 0:
49                  break
50          else: return rand
51
```

Prova de miller-rabin(120 iterações):
Foi adicionado um loop que não faz parte do algoritmo original,
pela velocidade, certa veracidade e praticidade adicional. A tabela de primos já havia sido implementada de qualquer modo, para aumentar a velocidade de encontrar possíveis primos antes do teste, então adicionou-se esse loop. Além disso, o resto do algoritmo segue o documento de referência.

```
53  def miller_rabin(n):
54      for prime in primes_list:
55          if n % prime == 0 or n!=int(n):
56              return False
57          elif n == prime:
58              return True
```

NOTA: Caso deseje testar os valores, edite o código do arquivo keygens.py e descomente as linhas 62/63. O teste de primo ocorre nesse caso com sympy.isprime().

```
62  #print_vars(P,Q,N,T,E,D)
63  #test_vars(P,Q,N,T,E,D)
```

RSA

Implementação: cifra-se a entrada do usuário

```
26   msg = read_input()
27   msg_hash = sha3(base64.b64encode(bytes(msg, 'utf-8')))
```

E utiliza-se a função ´hashlib.sha_256()´—

```
10   def sha3(msg):
11     hash = sha3_256(msg)
12     return hash.digest()
```

para verificar se a mensagem em claro(*msg*) coincide com a mensagem descriptografada(*dec*):

```
27   msg_hash = sha3(base64.b64encode(bytes(msg, 'utf-8')))

64   dec_hash = sha3(base64.b64encode(bytes(dec_str, 'utf-8')))
65   print("sha3 da mensagem original == sha3 da mensagem decodificada do rsa?")
66   print(msg_hash == dec_hash)
```

A criptografia/descriptografia foi a exponenciação das mensagens pela chaves pública e privadas, sem OAEP, em BASE64.

```
21   def enc(msg):
22     enc=[]
23     for i in range(0,len(msg)):
24       enc.append(pow(msg[i],PUBLIC[0],PUBLIC[1]))
25     return enc
26
27   def dec(c):
28       dec = []
29       for i in range(0,len(c)):
30           dec.append(pow(c[i],PRIVATE[0],PRIVATE[1]))
31       return dec
32
33   def dec_str(enc_b64):
34       dec = ''
35       for i in range (0,len(enc_b64)):
36           e =  pow(enc_b64[i],PRIVATE[0],PRIVATE[1])
37           dec += chr(e)
38       return dec
```

Ao fim do código, mostram-se as mensagens original e descriptografada, assim como o resultado de comparação de hashes das duas.

```
dec_hash = sha3(base64.b64encode(bytes(dec_str, 'utf-8')))
print("sha3 da mensagem original == sha3 da mensagem decodificada do rsa?")
print(msg_hash == dec_hash)


sha3 da mensagem original == sha3 da mensagem decodificada do rsa?
True
```