Mateus Elias de Macedo – 222011561

Andrey Calaça Resende - 180062433

O primeiro objetivo do programa é obter uma mensagem criptografada com RSA. Entretanto, para isso poder ser realizado, várias outras etapas precisam acontecer antes.

Primeiramente, 2 números primos ‘p’ e ‘q’ devem ser gerados para a criação das chaves do RSA. Para isso acontecer, se gera um número aleatório de 1024 bits e se confere se ele é primo utilizando o teste de primalidade Miller-Rabin “MRprim(n,k)”.

Esse teste se baseia em encontrar ‘d’ e ‘s’ tal que $2^s d$= n-1 e, a partir de uma base aleatória ‘a’, conferir a sequência $a^{d*2^k} mod(n)$, para k = 1,2,3...s , nessa sequência, se um valor for diferente de 1 mod(n) ou -1 mod(n) (que é igual a n -1 mod(n)), ou se um valor que não seja o primeiro for igual a -1 mod(n), o número é composto. Sabemos disso pois, pelo pequeno teorema de Fermat $a^{n-1} \equiv 1\ mod(n)$ e que as únicas raízes de 1 mod(n) são 1 e -1, logo, como a sequência é composta das raízes do valor anterior e seu último valor é igual a 1 mod(n), todos os valores são iguais a 1, exceto potencialmente o primeiro.

O teste de Miller-Rabin não determina se o número é primo, apenas se ele é composto, para esse motivo, a implementação do programa realiza esse teste k (por padrão 100) vezes, para ‘a’s diversos, minimizando a probabilidade de um erro ser cometido na avaliação.

```python
def genprime():     # Gera um número ale
    check = 0
    p = secrets.randbits(size)
    while (p < 2) or (p % 2 == 0):
        p = secrets.randbits(size)
    count = 0
    while check == 0:
        count += 1
        check = MRprim(p,rnds)
        if check == 1:
            break
        p = secrets.randbits(size)
        while (p < 2) or (p % 2 == 0):
            p = secrets.randbits(size)
    return p
```

```python
def MRprim(n,k):    # Confere se é provavelmente prim
    n1 = n -1
    s = 0
    while (n1 % 2 == 0):    # Dividindo por 2 até enc
        s += 1
        n1 = n1 // 2
    d = n1
    for i in range(k):
        a = secrets.randbelow(n-4) + 2  # Pegando núm

        x = pow(a,d,n)
        y = 0
        for j in range(s):
            y = pow(x,2,n)
            if (y == 1) and (x != 1) and (x != n-1):
                return 0    # Composto
            x = y
        if y != 1:
            return 0    # Composto
    return 1    # Provavelmente primo
```

```python
p = genprime()
q = genprime()  # Gera p e q primos
n,e,d = genrsakeys(p,q) # Obtém as chaves usando p e q
public_key = (n,e)
private_key = (n,d)
```

Após gerar ‘p’ e ‘q’ primo utilizando esse teste, se gera as chaves do rsa, definindo e como 65537 como é padrão em muitas aplicações desse método e utilizando a função de Carmichael de n (λ(n)) que, por p e que serem primos é igual ao

menor múltiplo comum de p-1 e q-1. Então é realizado o algoritmo de Euclides expandido em ‘e’ e ‘λ(n)’ e ‘d’ será o coeficiente de ‘e’. Assim, obtém-se a chave pública (n,e) e privada (n,d).

```python
def genrsakeys(p,q):    # Gera as chaves usando p,q e o algoritmo de euclid
    n = p * q
    lcm = ((p-1)*(q-1))//gcdeuclid((p-1),(q-1))
    if lcm < 0:
        lcm = - lcm
    e = 65537
    euc =  euclidexp(e,lcm)
    if (((euc[1] * e) + (euc[2] * lcm)) != 1):
        print("Error in finding d")
    d = euc[1]
    return n,e,d
```

```python
def euclidexp(a,b): # Algoritmo de Euclid Expandido
    rs = sorted([a,b],reverse=True)
    ss = [1,0]
    ts = [0,1]
    r = 1
    t = 0
    s = 0
    q = 0
    pos = 1
    while r > 0:
        q = (rs[(pos-1)%2]//rs[pos])
        r = rs[(pos-1)%2] - q * rs[pos]
        s = ss[(pos-1)%2] - q * ss[pos]
        t = ts[(pos-1)%2] - q * ts[pos]
        pos = (pos + 1) % 2
        rs[pos] = r
        ss[pos] = s
        ts[pos] = t
    r = rs[(pos-1)%2]
    s = ss[(pos-1)%2]
    t = ts[(pos-1)%2]
    if a > b:
        return [r,s,t]
    else:
        return [r,t,s]
```

Com as chaves prontas, o programa pode pegar como input uma mensagem do usuário, que é então dividida em segmentos de tamanho $k - 2 * hlen - 2$, onde k é o tamanho em bytes do n usado no rsa (que nesse caso é 2048 bits ou 256 bytes) e hlen o tamanho em bytes do valor de retorno da função hash utilizada (nesse caso 32). É utilizado esse como tamanho máximo, pois ainda será feito o padding da mensagem com o OAEP. Após o padding, a mensagem pode ser facilmente criptografada utilizando a fórmula $m^e mod(n)$ ou assinada substituindo ‘e’ por ‘d’ para cada pedaço de mensagem ‘m’, e decifrado do mesmo jeito, utilizando um pedaço criptografado como ‘m’ e utilizando ‘d’ se tinha usado ‘e’ na criptografia e vice-versa.

```python
tempM = input()     # Pega a mensagem
hlen = 32
k = 2048    # Comprimento doo produto de 2 números de 1024 bits
mlen = k//8 -2*hlen -2 # mlen é o comprimento máximo de M para um OAEP de um RSA com n de ta
Mlist = []
Msize = len(tempM)

if  Msize  > mlen:      # Se a mensagem é grande demais,separa em partes menores
    i = 1
    while i * mlen < Msize:
        newM = []
        for j in range(mlen):
            newM.append(tempM[j + (i-1)*mlen])
        Mlist.append(''.join(newM)) # Convertendo a lista e string e colocando na lista
        i+= 1
    base = (i-1)*mlen
    if base != Msize:   # Pegando o resto dos caractéres que não estão em um bloco completo
        limit = Msize - base
        newM = []
        for j in range(limit):
            newM.append(tempM[j + base])
        Mlist.append(''.join(newM))
else:
    Mlist.append(tempM)
```

```python
Clist = []
for M in Mlist:
    PM = OAEP(MGF,HASH,hlen,k//8,M,'')      # Obtendo a mensagem com padding
    C = RSA(PM,public_key[0],public_key[1]) # Obtendo mensagem criptografada
    Clist.append(PM)        # Colocando na lista de Mensagens criptografadas
    dM = RSA(C,private_key[0],private_key[1])   # Decriptografando a mensagem
    deM = deOAEP(MGF,HASH,hlen,k//8,dM,'')  # Removendo o padding
    print(M)
    print()
    print(deM)  # Imprimindo ambas as mensagens para conferir que são iguais (
return Mlist,Clist
```

```python
def RSA(pm,n,e):
    c = pow(int.from_bytes(pm),e,n) # Funciona para a  criptografia e decriptografia
    return c.to_bytes(256,'big')
```

O padding OAEP foi feito utilizando o hash SHA-3_256 e MGF (Função Geradora de Máscara) MGF1, sendo que esta foi implementada diretamente no código, enquanto o SHA fez uso de biblioteca hashlib. O padding envolve pegar o Hash de uma label qualquer ‘L’, deixada como vazia por conveniência, e concatená-lo a uma sequência de bytes 0, seguida de um byte 1 separador, seguido da mensagem,

tal que o tamanho desse bloco seja $k - hlen - 1$. Após isso, pode-se gerar uma 'seed' aleatória de tamanho hlen e uma máscara para o bloco usando-a, seguido de uma máscara para essa semente usando a máscara do bloco. Por fim, o OAEP retorna a concatenação de um byte 0 com a máscara da semente com a máscara do bloco, com um tamanho total de k bits.

O desempacotamento desse padding é feito isolando essas 3 partes e fazendo o processo inverso de cada etapa quando possível. O byte 1 do bloco é utilizado para poder nessa etapa separar o padding de 0s da mensagem em si, enquanto o hash da label é comparado com o hash feito na decifração para confirmar a integridade da mensagem.

```python
def OAEP(MGF,HASH,hlen,k,M,L): # Toma como argumento a função MGF
    lhash = HASH(L)

    mlen = len(M)
    PS = bytes(k - mlen - 2*hlen - 2)
    DB = lhash + PS + bytes([1]) + bytes(M,'utf-8')
    seed = secrets.randbits(8*hlen).to_bytes(hlen,'big')
    dbMask = MGF(seed,k-hlen-1)
    maskDB = bytes(a ^ b for a, b in zip(dbMask, DB))
    seedMask = MGF(maskDB,hlen)
    maskseed = bytes(a ^ b for a, b in zip(seedMask, seed ))
    return bytes(1) + maskseed + maskDB # Tem tamanho K
```

```python
def deOAEP(MGF,HASH,hlen,k,EM,L):
    lhash = HASH(L)
    mlen = k -2*hlen -2
    maskseed = EM[1:hlen+1]
    maskDB = EM[hlen+1:]
    seedMask = MGF(maskDB,hlen)
    seed = bytes(a ^ b for a, b in zip(seedMask, maskseed ))
    DBmask = MGF(seed,k-hlen-1)
    DB = bytes(a ^ b for a, b in zip(maskDB, DBmask))
    pslen = -1
    i = hlen
    crntbyte = b'0'
    while crntbyte != 1:    # Encontrando PSlen
        crntbyte = DB[i]
        i += 1
        pslen += 1
    lhash2,PS2,pad,M = DB[:hlen],DB[hlen:hlen+pslen], DB[hlen+pslen], DB[hlen+pslen+1:]
    PS = bytes(pslen)
    if lhash != lhash2 or PS != PS2 or pad != 1:    # Checando se os hashes são diferentes
        print(lhash != lhash2,PS != PS2, pad != 1)
        print("Erro: Hash não confere")
        return -1
    else:
        return M
```

O hash precisa ser gerado para que a assinatura possa ser aplicada, então a função de geração do hash é utilizada e o resultado é convertido para o formato de inteiro para que a potenciação possa ser feita.

A assinatura é feita utilizando a chave privada gerada anteriormente, utilizando a mesma forma que outras mensagens são assinadas. A função também converte a assinatura para o formato de byte e retorna a assinatura já formatada em base64.

```python
def signMsg(msg, private_key):
    hash_bytes = hashlib.sha3_256(msg.encode('utf-8')).digest()
    n, d = private_key
    padmsg = OAEP(MGF,HASH,32,256,hash_bytes,'')
    sign_byte = RSA(padmsg,n,d)
    assinatura_b64 = base64.b64encode(sign_byte).decode('utf-8') #Converte para base64

    return assinatura_b64
```

A verificação é feita a partir da comparação entre os bytes finais do hash decifrado com os bytes do hash da mensagem, para isso é utilizada a chave pública que decifra a assinatura

```python
def verifySign(msg, b64, public_key): #Verifica se a assinatura está correta
    sign_bytes = base64.b64decode(b64) #Transforma a string em base64 para byte
    n, e = public_key

    decbytes = RSA(sign_bytes,n,e)
    hash_bytes  = deOAEP(MGF,HASH,32,256,decbytes,'')

    hash_msg = hashlib.sha3_256(msg.encode('utf-8')).digest() #Calcula o hash da mensagem em claro

    return hash_bytes[-len(hash_msg):] == hash_msg #Compara o hash decifrado com o hash da mensagem olhando os últimos bytes.
```

O retorno consiste apenas de um bool que informa se a assinatura é válida, caso os bytes sejam iguais, ou inválida caso sejam diferentes.

Link do repositório: https://github.com/Mattelis/SC---P2/tree/main