Trabalho 3 Segurança Computacional

1st Eduardo Ferreira Marques Cavalcante - T02 - 202006368
Departamento de ciência da computação
Universidade de Brasília
Brasília, Brasil
202006368@aluno.unb.br

2nd Pedro Rodrigues Diógenes Macedo - T01 - 211042739
Departamento de ciência da computação)
Universidade de Brasília
Brasília, Brasil
211042739@aluno.unb.br

I. Introdução

No cenário atual, a segurança da informação desempenha um papel fundamental na proteção e confiança dos dados transmitidos digitalmente. A implementação de algoritmos de criptografia e assinatura digital torna-se uma prioridade para garantir a autenticidade das comunicações. Este relatório aborda a implementação de um gerador e verificador de assinaturas RSA em arquivos, utilizando a linguagem de programação Python e sua biblioteca *hashlib*.

O trabalho propõe a construção de um sistema capaz de gerar chaves RSA, cifrar e decifrar informações de forma assimétrica, além de criar e verificar assinaturas digitais para garantir a autenticidade dos dados transmitidos. Dividido em três partes distintas, o problema abrange os processos de geração de chaves(mínimo 1024 bits), cifração/decifração assimétrica RSA usando OAEP, cálculo de hashes da mensagem (hash SHA-3), assinatura digital, formatação do resultado e verificação, de acordo com a especificação proposta.

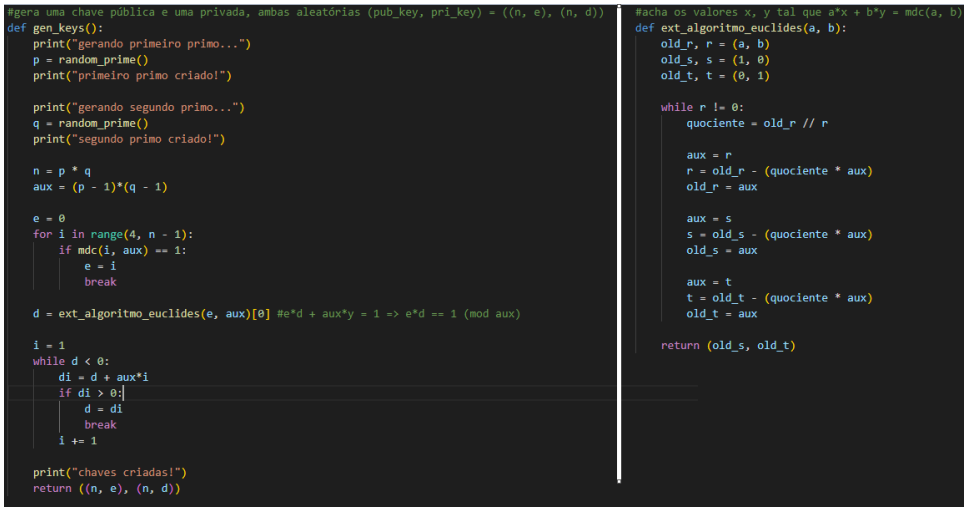
Utilizando a linguagem Python e sua biblioteca hashlib, este projeto busca apresentar uma solução robusta e eficiente para assegurar a confiabilidade das informações trocadas entre partes. No entanto, criamos a solução apenas geração de chaves com teste de primalidade (Miller-Rabin), cifração e decifração RSA, OAEP, formatação/parsing. Asolução pode ser encontrada no repositório [1].

II. Metodologia

A. Geração de chaves e cifra

Primeiramente devemos fazer a geração de chaves pública e privada $(pub\_key, pri\_key) = ((n, e), (n, d))$, para isso devemos gerar dois primos (p e q primos com no mínimo de 1024 bits). Esses primos foram gerados a partir do teste de primalidade Miller-Rabin. Após gerar esses primos, calculamos n como o produto desses primos. Encontramos então um número e e d que seguem da formula $e.d == 1(mod(p-1)(q-1))$. Para isso, encontramos primeiramente e (um primo em relação a $n-1$ que é $3 < e < n-1$), e depois utilizamos o algoritmo de Euclides para achar o d, pois $e.d + (p-1)(q-1).y = 1$. Correspondente à lógica em [2].

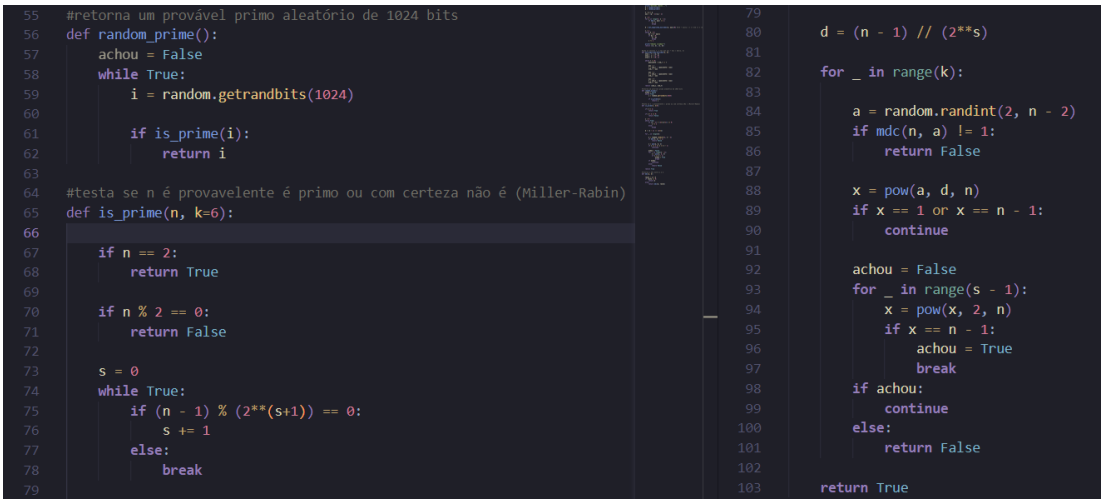
O algoritmo de Euclides encontra os valores x e y tais que $a * x + b * y = mdc(a, b)$ (o máximo divisor comum entre a e b). O algoritmo mantendo uma sequência de equações onde a cada passo, as variáveis old_r, old_s, e old_t representam os valores da iteração anterior e r, s e t representam os valores da iteração atual. A cada passo, são feitas atualizações nessas variáveis até que o $mdc(a, b)$ seja encontrado. O resultado retornado é o par de valores (x,y) que satisfazem a equação dada.



```python
#gera uma chave pública e uma privada, ambas aleatórias (pub_key, pri_key) = ((n, e), (n, d))
def gen_keys():
    print("gerando primeiro primo...")
    p = random_prime()
    print("primeiro primo criado!")

    print("gerando segundo primo...")
    q = random_prime()
    print("segundo primo criado!")

    n = p * q
    aux = (p - 1)*(q - 1)

    e = 0
    for i in range(4, n - 1):
        if mdc(i, aux) == 1:
            e = i
            break

    d = ext_algoritmo_euclides(e, aux)[0] #e*d + aux*y = 1 => e*d == 1 (mod aux)

    i = 1
    while d < 0:
        di = d + aux*i
        if di > 0:
            d = di
            break
        i += 1

    print("chaves criadas!")
    return ((n, e), (n, d))
```

```python
#acha os valores x, y tal que a*x + b*y = mdc(a, b)
def ext_algoritmo_euclides(a, b):
    old_r, r = (a, b)
    old_s, s = (1, 0)
    old_t, t = (0, 1)

    while r != 0:
        quociente = old_r // r

        aux = r
        r = old_r - (quociente * aux)
        old_r = aux

        aux = s
        s = old_s - (quociente * aux)
        old_s = aux

        aux = t
        t = old_t - (quociente * aux)
        old_t = aux

    return (old_s, old_t)
```

Figura 1. Criação das chaves á esquerda e Algoritmo de Euclides à direita.



```python
#retorna um provável primo aleatório de 1024 bits
def random_prime():
    achou = False
    while True:
        i = random.getrandbits(1024)

        if is_prime(i):
            return i

#testa se n é provavelente é primo ou com certeza não é (Miller-Rabin)
def is_prime(n, k=6):

    if n == 2:
        return True

    if n % 2 == 0:
        return False

    s = 0
    while True:
        if (n - 1) % (2**(s+1)) == 0:
            s += 1
        else:
            break
```

```python
    d = (n - 1) // (2**s)

    for _ in range(k):

        a = random.randint(2, n - 2)
        if mdc(n, a) != 1:
            return False

        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue

        achou = False
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                achou = True
                break
        if achou:
            continue
        else:
            return False

    return True
```

Figura 2. Geração dos números primos de 1024 bits.

Com as chaves criadas, podemos fazer a decifração(chave privada) com o resultado da cifração(chave pública) sobre a mensagem(lida do arquivo teste.txt).

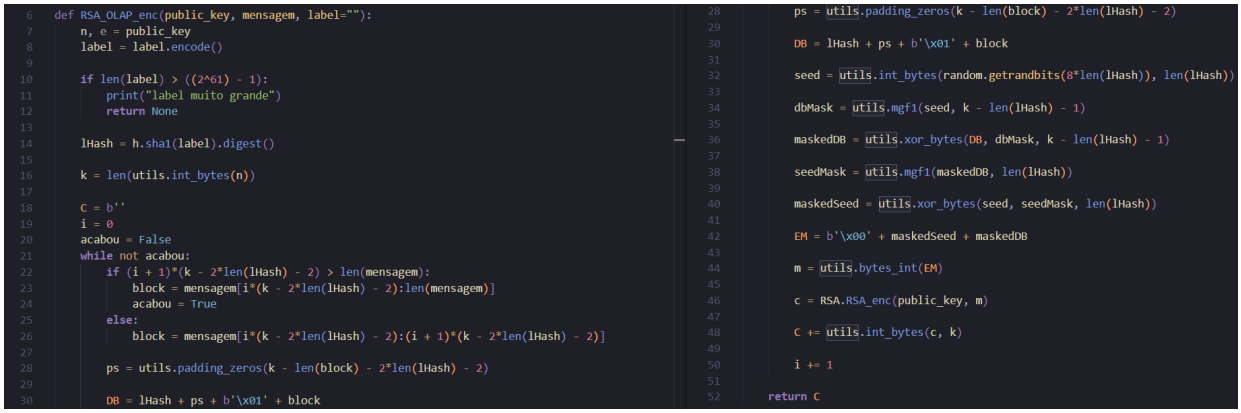
Rapidamente, falaremos sobre a cifração e decifração RSA que são bastante simples. Na cifração, utilizamos a chave pública (n, e). Primeiro verificamos se o tamanho da mensagem está dentro dos limites $(0 < tamanho < n-1)$ e a cifra é $mensagem^e$ (mod n). Já na decifração, utilizando a chave privada (n, d) verificamos também se o tamanho da cifra está dentro dos limites e a mensagem é $cifra^d$ (mod n). A implementação pode ser vista a seguir.

```python
#cifrador RSA, recebe mensagem (inteiro entre 0 e n-1) e uma chave pública (n, e)
def RSA_enc(pub_key, men: int):
    n, e = pub_key

    if not (0 < men and men < n - 1):
        print("representação da mensagem fora de alcance")

    c = pow(men, e, n)

    return c

#decifrador RSA, recebe cifra (inteiro entre 0 e n-1) e uma chave privada (n, d)
def RSA_dec(pri_key, cipher: int):
    n, d = pri_key

    if not (0 < cipher and cipher < n - 1):
        print("representação da cifra fora de alcance")

    m = pow(cipher, d, n)

    return m
```

Figura 3. Cifrador e decifrador RSA.

Para a cifração com chave pública, temos a função *RSA_OLAP_enc(public_key, mensage, label=)*:

1) Entrada:
   - $public\_key$: A chave pública RSA (n, e).
   - $message$: A mensagem a ser cifrada.
   - $label$ (opcional): Um rótulo associado à mensagem (pode ser vazio).
2) Processo::
   - Converte o $label$ para bytes.
   - Verifica se o tamanho do $label$ não excede um limite específico.
   - Calcula um hash SHA-1 do $label$ (lHash).
   - Determina k sendo o tamanho de n em bytes.
   - Quebra a mensagem em blocos de tamanho que tem um tamanho fixo máximo. Esses blocos são cifrados um por vez e depois concatenados para formar a cifra da mensagem toda.
   - Determina C tipo bytes vazio
   - Para cada bloco faça:
   - Realiza a operação de padding (preenchimento) com zeros para ajustar o tamanho o bloco.
   - Gera um conjunto de dados (DB) concatenando $lHash$, o padding, um byte fixo (0x01), e o bloco.
   - Gera um seed aleatório e máscaras ($dbMask$ e $seedMask$) usando o algoritmo MGF1.
   - Aplica as máscaras para obscurecer o DB e o seed.
   - Cria o bloco cifrada (EM) concatenando os bytes gerados.
   - Concatena ao final de C o EM
3) Saída:
   - Retorna a mensagem cifrada.



```python
def RSA_OLAP_enc(public_key, mensagem, label=""):
    n, e = public_key
    label = label.encode()

    if len(label) > ((2^61) - 1):
        print("label muito grande")
        return None

    lHash = h.sha1(label).digest()

    k = len(utils.int_bytes(n))

    C = b''
    i = 0
    acabou = False
    while not acabou:
        if (i + 1)*(k - 2*len(lHash) - 2) > len(mensagem):
            block = mensagem[i*(k - 2*len(lHash) - 2):len(mensagem)]
            acabou = True
        else:
            block = mensagem[i*(k - 2*len(lHash) - 2):(i + 1)*(k - 2*len(lHash) - 2)]

        ps = utils.padding_zeros(k - len(block) - 2*len(lHash) - 2)

        DB = lHash + ps + b'\x01' + block

        seed = utils.int_bytes(random.getrandbits(8*len(lHash)), len(lHash))

        dbMask = utils.mgf1(seed, k - len(lHash) - 1)

        maskedDB = utils.xor_bytes(DB, dbMask, k - len(lHash) - 1)

        seedMask = utils.mgf1(maskedDB, len(lHash))

        maskedSeed = utils.xor_bytes(seed, seedMask, len(lHash))

        EM = b'\x00' + maskedSeed + maskedDB

        m = utils.bytes_int(EM)

        c = RSA.RSA_enc(public_key, m)

        C += utils.int_bytes(c, k)

        i += 1

    return C
```

Figura 4. Função *RSA_OLAP_enc*

Para a decifração temos a função *RSA_OLAP_dec(private_key, cifra, label=)*:

1) Entrada:
   - $private\_key$: A chave privada RSA (n, d).
   - $cipher$: A mensagem encriptada a ser decifrada.
   - $label$ (opcional): O rótulo associado à mensagem.
2) Processo::
   - Converte o $label$ para bytes.
   - Verifica se o tamanho do $label$ não excede um limite específico.
   - Determina k sendo o tamanho de n em bytes.
   - Verifica se o tamanho da mensagem encriptada é múltipla de k.
   - Quebra a mensagem encriptada em blocos de tamanho k. Cada bloco será decifrado e depois concatenado para forma a mensagem original.
   - Para cada bloco faça:
   - Realiza a decifração RSA para obter o bloco encriptado.
   - Converte o bloco encriptado de volta para bytes.
   - Verifica se a estrutura do bloco encriptado está correto (começando com $0x00$).
   - Extrai o seed encriptado e o DB encriptado do bloco encriptado.
   - Utiliza as máscaras ($seedMask$ e $dbMask$) para desfazer a encriptação desses blocos.
   - Verifica se a estrutura e integridade dos blocos decifrados estão corretas.
   - Remove o padding e recupera o bloco original.
3) Saída:
   - Retorna a mensagem original decifrada.

```python
def RSA_OLAP_dec(private_key, cifra, label=""):
    n, d = private_key
    label = label.encode()

    if len(label) > ((2^61) - 1):
        print("erro de descriptografia\n")
        return None

    k = len(utils.int_bytes(n))

    acabou = False
    M = b''
    j = 0
    while not acabou:
        if (j + 1)*k == len(cifra):
            acabou = True
        if (j + 1)*k <= len(cifra):
            block = cifra[j*k: (j + 1)*k]
        else:
            print("erro de descriptografia\n")
            return None

        lHash = h.sha1(label).digest()

        if k < 2*len(lHash) + 2:
            print("erro de descriptografia\n")
            return None

        c = utils.bytes_int(block)

        m = RSA.RSA_dec(private_key, c)

        EM = utils.int_bytes(m, k)

        if EM[0] != 0:
            print("erro de descriptografia\n")
            return None

        maskedSeed = EM[1:1 + len(lHash)]

        maskedDB = EM[1 + len(lHash):]

        seedMask = utils.mgf1(maskedDB, len(lHash))

        seed = utils.xor_bytes(maskedSeed, seedMask, len(lHash))

        dbMask = utils.mgf1(seed, k - len(lHash) - 1)

        DB = utils.xor_bytes(maskedDB, dbMask, k - len(lHash) - 1)

        if DB[:len(lHash)] != lHash:
            print("erro de descriptografia\n")
            return None

        i = len(lHash)
        while True:
            if DB[i] == 0:
                i += 1
                continue
            elif DB[i] == 1:
                break
            else:
                print("erro de descriptografia\n")
                return None
        M += DB[i+1:]
        j += 1
    return M
```

Figura 5. Função *RSA_OLAP_dec*

B. *Assinatura e Verificação*

O processo de assinatura e verificação é também bastante simples. Para a assinatura temos:

1) Entrada:
   - $private\_key$: A chave privada RSA (n, d).
   - $mensagem$: A mensagem a ser assinada.
2) Processo::
   - Calcula o hash de $mensagem$ (utilizamos o SHA-3).
   - Encripta o resultado do hash com a $private\_key$ usando RSA_OLAP_enc.
   - Codifica o resultado usando a base 64.
3) Saída:
   - Retorna a mensagem assinada.

Agora para a verificação:

1) Entrada:
   - $public\_key$: A chave pública RSA (n, e).
   - $men\_assinada$: A mensagem assinada.
   - $men\_original$: A mensagem original.
2) Processo::
   - Calcula o hash de $men\_original$ (para verificar a assinatura).
   - Decodifica $men\_assinada$ com a base 64.
   - Decripta o resultado da decodificação com a $public\_key$ usando RSA_OLAP_dec.
   - Compara o hash calculado no primeiro passo com o resultado do último. Caso sejam iguais, é válido; caso o contrário, não é válido.
3) Saída:
   - Retorna um booleano identificando o resultado da verificação.

A implementação da assinatura e da verificação pode ser encontrado a seguir:

```python
# Recebe uma mensagem e assina ela com sua chave privada
def assinatura(mensagem, chave_pri):
    hashM = h.sha3_224(mensagem).digest()

    cifra_hashM = oaep.RSA_OLAP_enc(chave_pri, hashM)

    mensagem_assinada = base64.b64encode(cifra_hashM)

    return mensagem_assinada

# Recebe uma mensagem assinada, decodifica ela com a chave publica. Caso a mensagem
# gerada não for igual a original, retorna false. Caso o contrário, retorna true
def verificacao(assinatura, chave_pub, mensagem_original):
    hashM_original = h.sha3_224(mensagem_original).digest()

    cifra_hashM = base64.b64decode(assinatura)

    hashM = oaep.RSA_OLAP_dec(chave_pub, cifra_hashM)

    if hashM == hashM_original:
        return True
    else:
        return False
```

Figura 6. Função de assinatura e de verificação

III. Conclusões

Conseguimos implementar todas as partes do projeto com êxito e, vendo tudo funcionando como deveria, vimos como essas tecnologias de criptografia se mostram fortes e atuais diante do cenário de inúmeros sistemas de informação existentes. Obtivemos algumas dificuldades de encontrar documentação sobre o assunto, mas após acharmos, não tivemos muitas dificuldades para implementar.

Referências

[1] E. Ferreira, R. Pedro, "Projeto 2 de Segurança Computacional 2023.2," GitHub, Oct. 30, 2023. https://github.com/EduardoFMC/SC-AES/tree/main

[2] K. Moriarty, J. Subset, and A. Rusch, "Internet Engineering Task Force (IETF)," 2016. Disponível: https://www.rfc-editor.org/rfc/pdfrfc/rfc8017.txt.pdf