Documentação Seminário

Descrição dos algoritmos utilizados

RSA

- É um algoritmo de encriptação assimétrica, ou seja, utiliza uma chave pública e uma chave privada para o processo de encriptação/decriptação
- O texto em claro e o cifrado são inteiros entre 0 e n - 1, para algum n (geralmente 1024 bits)
    - O texto em claro é encriptado em blocos, onde cada um tem um valor binário menor que n
- Encriptação
    - $C = M^e \bmod n$
    - C = Texto encriptado
    - M = Mensagem em claro
    - e = Expoente público
    - n = Módulo público
- Decriptação
    - $M = C^d \bmod n$
    - C = Texto encriptado
    - M = Mensagem em claro
    - d = Expoente privado
    - n = Módulo público
- Para o cálculo dos valores acima (n, e, d) deve-se seguir os seguintes passos
    - Gerar dois números grandes primos (p e q)
    - Calcular o produto $n = p \times q$
    - Calcular o produto $\phi(n) = (p-1) \times (q-1)$
    - A partir desses valores, deve ser escolhido um valor “e” tal que “e” seja coprimo com $\phi(n)$ e $1 < e < \phi(n)$
    - A partir disso, o valor de d é calculado da seguinte forma $d \equiv e^{-1} \bmod n$
- Segurança do algoritmo
    - A segurança desses cálculos foi comprovada matematicamente e se baseia na dificuldade de fatorar números grandes compostos
    - O cálculo dos valores mencionados deve ser “trivial” para quem possui todas as informações, no entanto o cálculo inverso deve ser computacionalmente inviável (por meio de logaritmo discreto)

OAEP - Optimal Asymetric Encryption Padding

- O OAEP transforma uma mensagem em claro antes dela ser cifrada. Evitando que ela fique pequena ou com padrões previsíveis, garantindo que um mesmo texto produza textos cifrados diferentes e deixando o processo mais seguro
- Funcionamento do algoritmo
    - Divisão do bloco
        - A mensagem vira um bloco de dados (db) e uma seed. O bloco de dados é composto por um padding (ps) de bytes de 0x00, um delimitador 0x01 e a mensagem original
    - Geração de máscara (mfg1)

        - Produz uma máscara para o bloco de dados a partir da seed
        - Produz uma máscara para a seed a partir do bloco de dados
    - Aplicação das máscaras
        - Através de um XOR as respectivas máscaras são aplicadas
    - Combinação de blocos
        - Os resultados são combinados, ficando um byte inicial 0x00, o seed mascarado e o bloco de dados mascarado
- Segurança
    - A seed torna os textos cifrados imprevisíveis
    - O padding e a máscara removem padrões previsíveis na mensagem
    - A segurança depende da resistência à colião de hash

SHA-3 - Secure Hash Alorithm 3

- Uma função de hash produz uma saída de tamanho fixo e é determinística, ou seja, a mesma entrada gera sempre o mesmo hash
- Além disso, ela possui resistência a colisões, dessa forma é díficil encontrar duas entradas diferentes que gerem o mesmo hash
- Além desas características, o SHA-3 possui uma característica especial: uma construção matemática chamada função esponja
    - Os blocos da entrada são combinados gradativamente → Absorção
    - Ao processar a entrada por completo, o hash final é gerado em partes → Escoamento
- Algoritmo
    - O estado inicial é uma matriz de 5×5 onde cada posição armazena 64 bits
    - A entrada então é dividida em blocos de acordo com a taxa (r). No caso do algoritmo usado no trabalho (sha3_256) r = 1088 bits
    - Então cada bloco é combinado com a matriz através de operaões XOR e depois o estado é modificado pela função de permutação Keccak-f
        - Theta: adiciona redundância a cada coluna da matriz
        - Rho: rotaciona bits em cada posição da matriz
        - Pi: rearranja as posições dentro da matriz
        - Chi: Introuz não linearidade (aplicação de XOR entre bits da matriz)
        - Iota: adiciona uma constante evitando a simetria entre rodadas
    - Depois de processar todos os blocos, o estado interno gera o hash final, r bits de cada vez
- Segurança
    - Função Esponja: resiste contra ataques como colisões e pré-imagens
    - Alta entropia: combinando operações bit a bit e constantes específicas eliminam padrões
    - Adaptável a diferentes tamanhos de entrada e saída
    - Imune a ataques que explora a dependência entre blocos

Análise da implementação

Geração de chaves

```
def miller_rabin(n, d):
    a = randint(2, n - 2)
    x = pow(a, d, n)
```

```
    if x == 1 or x == n - 1:
        return True

    while d != n - 1:
        x = pow(x, 2, n)
        d *= 2

        if x == 1:
            return False
        if x == n - 1:
            return True

    return False

def isprime(n, k=5):

    if n == 2:
        return True

    if n < 2 or n % 2 == 0:
        return False

    d = n - 1
    while d % 2 == 0:
        d //= 2

    for _ in range(k):
        if not miller_rabin(n, d):
            return False

    return True

def gen_prime():
    while True:
        min_bits = 2 ** (NUM_BITS - 1)
        max_bits = 2 ** NUM_BITS - 1

        prime_number = randprime(min_bits, max_bits)

        if isprime(prime_number):
            return prime_number

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


def mod_inverse(a, m):
    m0, y, x = m, 0, 1

    if m == 1:
        return 0

    while a > 1:
        q = a // m
        m, a = a % m, m
        y, x = x - q * y, y
```

```
    return x + m0 if x < 0 else x


def generate_keys():
    p = gen_prime()
    q = gen_prime()
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537

    while gcd(e, phi) != 1:
        e = randrange(2, phi)

    d = mod_inverse(e, phi)

    return (e, n), (d, n)
```

- Para a geração dos números primos “p” e “q” foi utilizada uma função da biblioteca sympy chamada randprime
    - A função retorna um valor primo entre os valores passados no parâmetro. No caso, o valor deve estar entre 2^{1023} e $2^{1024} - 1$
- Escolha de “e”
    - Ao chamar a função gcd, checamos se “e” e “phi” são coprimos ao utilizar o Algoritmo de Euclides (princíoio de que o MDC não muda se o menor número for subtraído do maior
    - Caso esses números não sejam coprimos, um número aleatório entre 2 e $\phi(n)$ é selecionado até que essa propriedade seja verdadeira
- Checagem de primalidade
    - Para checar se o númeor gerado é primo, foi utilizado o teste de primalidade de Miller-Rabin

Criptografia / Decriptografia

```
def xor_bytes(a, b):
    return bytes([i ^ j for i, j in zip(a, b)])


def mgf1(seed, mask_len):
    hlen = sha3_256().digest_size

    output = b""

    for i in range(0, -(-mask_len // hlen)):
        c = i.to_bytes(4, 'big')
        output += sha3_256(seed + c).digest()

    return output[:mask_len]

def oaep_encode(message, n):
    mlen = len(message)
    pad = b'\x00' * (n - mlen - PADDING_LENGTH // 8 - 2)
    db = pad + b'\x01' + message

    seed = getrandbits(SEED_LENGTH).to_bytes(SEED_LENGTH // 8, 'big')
    db_mask = mgf1(seed, len(db))
    masked_db = xor_bytes(db, db_mask)
```

```
    seed_mask = mgf1(masked_db, SEED_LENGTH // 8)
    masked_seed = xor_bytes(seed, seed_mask)

    return b'\x00' + masked_seed + masked_db


def rsa_encrypt(plaintext, public_key):
    e, n = public_key
    k = (n.bit_length() + 7) // 8
    encoded_message = oaep_encode(plaintext, k)
    plaintext_int = int.from_bytes(encoded_message, 'big')
    ciphertext = pow(plaintext_int, e, n)

    return ciphertext


def oaep_decode(encoded_message, k):
    encoded_message = encoded_message[1:]

    masked_seed = encoded_message[:SEED_LENGTH // 8]
    masked_db = encoded_message[SEED_LENGTH // 8:]

    seed_mask = mgf1(masked_db, SEED_LENGTH // 8)
    seed = xor_bytes(masked_seed, seed_mask)

    db_mask = mgf1(seed, len(masked_db))
    db = xor_bytes(masked_db, db_mask)

    lhash_len = len(sha3_256().digest())
    ps_end = db.index(b'\x01', lhash_len)
    message = db[ps_end + 1:]

    return message


def rsa_decrypt(ciphertext, private_key):
    d, n = private_key
    k = (n.bit_length() + 7) // 8
    plaintext_int = pow(ciphertext, d, n)
    plaintext = plaintext_int.to_bytes(k, 'big')

    return oaep_decode(plaintext, k)
```

- Antes da encriptação ser feita, o texto é passado pela função de OAEP
    - O OAEP primeiro aplicará um padding de b’\x00’ e no final colocar b’\x01’ indicando que a mensagem vem logo após
    - Após isso, é gerado uma seed aleatória que será usada na função mfg1. Nela é criada uma máscara para o bloco de dados (db) e então é aplicada a partir de uma função XOR bit a bit
    - A partir do bloco de dados mascarado, é feita uma máscara para a seed, a qual é aplicada por meio de um XOR bit a bit também
- Após esse processo, a cifra RSA é aplicada em cima da mensagem encodificada
- O processo inverso é simples, ao aplicar a decifração do RSA, é chamada a funão de remoção do padding
    - Essa função remove o primeiro byte (b’\x00’)
    - Divide a mensagem em seed mascarada e bloco de dados mascarado

- Desmascara a seed a partir da função mfg1 e da máscara do bloco de dados
- Desmascara o bloco de dados a partir da função mfg1 e da seed
- Após isso, encontra o byte b’\x01’ e separa a mensagem verdadeira do preenchimento

Assinatura Digital

```
def calculate_hash(file_path):
    with open(file_path, 'rb') as f:
        while chunk := f.read(4096):
            sha3_256().update(chunk)

    return sha3_256().digest()


def sign_file(file_path, private_key):
    hash = calculate_hash(file_path)
    signature = rsa_encrypt(hash, private_key)
    formatted_signature = b64encode(signature.to_bytes((signature.bit_length() + 7) // 8, 'big')

    return formatted_signature


def verify_file(file_path, base64_signature, public_key):
    signature = int.from_bytes(b64decode(base64_signature), 'big')
    decrypted_hash = rsa_decrypt(signature, public_key)
    current_hash = calculate_hash(file_path)

    return decrypted_hash == current_hash
```

- Para a parte final, primeiro é lido o arquivo de blocos em blocos e é calculado o hash
    - Como foi comentado anteriormente, o sha3 é um algoritmo esponja, ou seja, aplica o hash a cada bloco para depois retornar o hash final. A analogia é feita como se fosse uma esponja acumulando água para depois soltar
    - Logo depois é feita uma assinatura digital. Isso é feito ao criptografar o hash calculado anteriormente. A partir dele e da chave privada do usuário é feito todo o processo comentado anteriormente na criptação. O resultado então é convertido para base64 para facilitar o armazenamento e o envio. Dessa forma, o resultado da criptografia, que é um númeor muito grande, é transformado um uma string legível
- A verificação da assinatura é feita com base no hash, no arquivo enviado e na chave pública
    - A assinatura recebida é decodificada (transformada de base64 para um número)
    - Esse valor é decriptografado utilizando a chave pública **daquele que enviou** o hash
    - O hash é calculado novamente com base no documento enviado
        - Se forem iguais, o documento é válido
        - Se forem diferentes, o documento pode ter sido alterado, tornando-o inválido

Algumas considerações

Como cada processo contribui para o aumento da segurnaça?

- O algoritmo RSA, como foi comentado anteriormente, tem sua segurnaça baseada na dificuldade de fatoração de números muito grandes. A partir da chave pública, foi matematicamente comprovado ser computacionalmente inviável realizar o processo inverso (logaritmo discreto) para descobrir a chave privada
- O OAEP adiciona mais uma camada de “aleatoriedade na mensagem”. Ao adicionar um padding, uma seed e mascarar a seed e o bloco de dados como um todo, o processo retira padrões que podem existir em mensagens e ainda altera o tamanho da mensagem a ser enviada. Dessa forma, fica ainda mais inviável reocnhecer padrões da mensagem codificada

    - Sem contar que dessa forma uma mesma mensagem pode gerar saídas diferentes
- Já o SHA3 tem uma proteção contra colisões muito eficiente, ou seja, é difícil encontrar duas mensagens que resultem no mesmo hash. Além disso, ele garante a integridade de um documento, dessa forma qualquer alteração que seja feita gera hashes completamente diferentes no resultado final
    - Curiosidade: a função esponja do SHA3-256 protege a mensagem contra atques de comprimento. A forma de funcionamento do SHA2, por exemplo, possibilitava que o atacante utilizasse a estrutura interna do hash após o processamento da mensagem para calcular outros hashes válidos. Por causa d aforma de absorção do SHA3 (como uma esponja), ele deixa de depender de um estado interno exposto

Por que o valor de “e” escolhido foi 65537?

- Esse é um valor comumente usado para um valor inicial “e”
- Caso ele não seja satisfeito, deve ser escolhido outro valor que satisfaça as condições de ser coprimo com $\phi(n)$ e estar entre 2 e $\phi(n)$

Quais são as vulnerabilidades do algoritmo implementado?

- Troca de chaves
    - Ao interceptar uma troca de chaves entre emissor e receptor, ele pode substituir as chaves públicas com chaves falsas, fazendo com que ambos criptografem mensagens com as chaves erradas
        - Solução: Uso de um Certificado Digital autenticado por uma Autoridade Certififcadora para validar a chave pública
- Ataque antes da assinatura
    - Ao modificar uma mensagem antes da sua assinatura, o atacante ainda consegue causar danos à integridade da informação
        - Solução: Uso do timestamp, ou seja, última modificação do arquivo
- Man in the middle
    - Um atacante pode tentar alterar a integridade da informação durante o processo de comunicação
        - Solução: Para evitar isso, basta verificar se o hash calculado corresponde à assinatura recebida

O tamanho da mensagem importa?

- Para o RSA puro, a mensagem deve ser menor que o tamanho do módulo
    - Ou seja, M < n; onde M é a mensagem e “n” o módulo (calculado por meio de $p \times q$)
    - O OAEP tem um grande impacto nisso, visto que ele reduz o espaço para a mensagem. Epsar dele aumentar a segurança significativamente, o seu preenchimento diminui a quantidade de informações que podem ser repassadas
        - Solução: Divisão da mensagem em blocos ou uso do esquema híbrido, com o uso de criptografia simétrica e assimétrica
- Na assinatura digital, o tamanho da mensagem também importa
    - Para isso, é possível utilizar funções de hashes compactos, como é o caso do SHA-256, reduzindo o tamanho do hash

O quão complexo é o algoritmo em questão de tempo?

- Geração de chaves
    - O processo consiste na geração de primos de 1024 bits e testá-los por meio do Teste da Primalidade de Miller-Rabin
        - Miller-Rabin tem uma complexidade de $O(k \log 2n)$, onde k é a precisão do algoritmo/número de iterações
    - Esse processo pode ser demorado na fase de teste de primalidade e na fase de escolha de “e”, visto que caso o número inicial não tenha as propriedades corretas, deve ser escolhido outro aleatoriamente entre um intervalo muito grande
    - Para números de 2048 e 4096 bits, esse tempo pode ser ainda maior

- Encriptação/Decriptação
    - Como o valor de “e” geralemente é significativamente menor, a criptografia é mais rápida
    - O processo inverso geralemente tem uma velocidade mais lenta, dado o tamanho de “d”
- Cálculo do hash
    - A função SHA3-256 possui uma complexidade linear onde N é o tamanho da entrada
    - Para arquivos muito grandes, ele pode levar mais tempo, mas no geral é considerado um algoritmo veloz

Código completo

```
'''
Implementação de gerador e verificador de assinaturas RSA

-> Parte 1: geração de chaves e cifra
    + Geração de chaves (p e q primos com no mínimo 1024  bits)
    + Cifração/decifração assimétrica RSA usando OAEP
-> Parte 2: assinatura
    + Cálculo de hashes da mensagem (SHA-3)
    + Assinatura da mensagem
    + Formatação do resultado (caracteres especiais e informações para verificação em BASE64)
-> Parte 3: verificação
    + Parsing do documento assinado e decifração da mensagem
    + Decifração da assinatura
    + Verificação da assinatura
'''
from sympy import randprime
from random import randrange, getrandbits, randint
from hashlib import sha3_256
from base64 import b64encode, b64decode

NUM_BITS = 1024
SEED_LENGTH = 256
PADDING_LENGTH = 256

def miller_rabin(n, d):
    a = randint(2, n - 2)
    x = pow(a, d, n)

    if x == 1 or x == n - 1:
        return True

    while d != n - 1:
        x = pow(x, 2, n)
        d *= 2

        if x == 1:
            return False
        if x == n - 1:
            return True

    return False

def isprime(n, k=5):

    if n == 2:
        return True
```

```
    if n < 2 or n % 2 == 0:
        return False

    d = n - 1
    while d % 2 == 0:
        d //= 2

    for _ in range(k):
        if not miller_rabin(n, d):
            return False

    return True




def gen_prime():
    while True:
        min_bits = 2 ** (NUM_BITS - 1)
        max_bits = 2 ** NUM_BITS - 1

        prime_number = randprime(min_bits, max_bits)

        if isprime(prime_number):
            return prime_number

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


def mod_inverse(a, m):
    m0, y, x = m, 0, 1

    if m == 1:
        return 0

    while a > 1:
        q = a // m
        m, a = a % m, m
        y, x = x - q * y, y

    return x + m0 if x < 0 else x


def generate_keys():
    p = gen_prime()
    q = gen_prime()
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537

    while gcd(e, phi) != 1:
        e = randrange(2, phi)

    d = mod_inverse(e, phi)
```

```
    return (e, n), (d, n)

def xor_bytes(a, b):
    return bytes([i ^ j for i, j in zip(a, b)])


def mgf1(seed, mask_len):
    hlen = sha3_256().digest_size

    output = b""

    for i in range(0, -(-mask_len // hlen)):
        c = i.to_bytes(4, 'big')
        output += sha3_256(seed + c).digest()

    return output[:mask_len]

def oaep_encode(message, n):
    mlen = len(message)
    pad = b'\x00' * (n - mlen - PADDING_LENGTH // 8 - 2)
    db = pad + b'\x01' + message

    seed = getrandbits(SEED_LENGTH).to_bytes(SEED_LENGTH // 8, 'big')
    db_mask = mgf1(seed, len(db))
    masked_db = xor_bytes(db, db_mask)

    seed_mask = mgf1(masked_db, SEED_LENGTH // 8)
    masked_seed = xor_bytes(seed, seed_mask)

    return b'\x00' + masked_seed + masked_db


def rsa_encrypt(plaintext, public_key):
    e, n = public_key
    k = (n.bit_length() + 7) // 8
    encoded_message = oaep_encode(plaintext, k)
    plaintext_int = int.from_bytes(encoded_message, 'big')
    ciphertext = pow(plaintext_int, e, n)

    return ciphertext


def oaep_decode(encoded_message, k):
    encoded_message = encoded_message[1:]

    masked_seed = encoded_message[:SEED_LENGTH // 8]
    masked_db = encoded_message[SEED_LENGTH // 8:]

    seed_mask = mgf1(masked_db, SEED_LENGTH // 8)
    seed = xor_bytes(masked_seed, seed_mask)

    db_mask = mgf1(seed, len(masked_db))
    db = xor_bytes(masked_db, db_mask)

    lhash_len = len(sha3_256().digest())
    ps_end = db.index(b'\x01', lhash_len)
    message = db[ps_end + 1:]
```

```
    return message


def rsa_decrypt(ciphertext, private_key):
    d, n = private_key
    k = (n.bit_length() + 7) // 8
    plaintext_int = pow(ciphertext, d, n)
    plaintext = plaintext_int.to_bytes(k, 'big')

    return oaep_decode(plaintext, k)


def calculate_hash(file_path):
    with open(file_path, 'rb') as f:
        while chunk := f.read(4096):
            sha3_256().update(chunk)

    return sha3_256().digest()


def sign_file(file_path, private_key):
    hash = calculate_hash(file_path)
    signature = rsa_encrypt(hash, private_key)
    formatted_signature = b64encode(signature.to_bytes((signature.bit_length() + 7) // 8, 'big')

    return formatted_signature


def verify_file(file_path, base64_signature, public_key):
    signature = int.from_bytes(b64decode(base64_signature), 'big')
    decrypted_hash = rsa_decrypt(signature, public_key)
    current_hash = calculate_hash(file_path)

    return decrypted_hash == current_hash


def main():
"""
IMPORTANTE:
Os arquivos abaixo foram arquivos que criei no mesmo diretório
para os testes das funções. Para testar o código, crie documentos
com os nomes abaixo ou troque o nome deles para algum documento
existente.
"""
    message_file = 'teste.txt'
    pdf_file = 'documentacao_seminario.pdf'

    message = 'Seminário de Segurança Computacional'

    public_key, private_key = generate_keys()

    cyphertext = rsa_encrypt(message.encode() , public_key)

    decrypted_message = rsa_decrypt(cyphertext, private_key).decode()

    file_signature = sign_file(message_file, private_key)
    pdf_file_signature = sign_file(pdf_file, private_key)
```

```
    verified = verify_file(message_file, file_signature, public_key)
    verified_pdf = verify_file(pdf_file, pdf_file_signature, public_key)

    print(f'###TRABALHO DE SEGURANÇA COMPUTACIONAL###\n\n')

    print(f'#INFORMAÇÕES INICIAIS##\n')
    print(f'Mensagem original: {message}\n')
    print(f'Chave Pública: {public_key}\n')
    print(f'Chave Privada: {private_key}\n')

    print(f'#CRIPTOGRAFIA#\n')
    print(f'Mensagem cifrada: {cyphertext}\n')

    print(f'#DECRIPTOGRAFIA#\n')
    print(f'Mensagem decifrada: {decrypted_message}\n')

    print(f'#ASSINATURA DIGITAL#\n')
    print(f'Assinatura digital: {file_signature}\n')
    print(f'Assinatura digital do PDF: {pdf_file_signature}\n')

    if (verified):
        print(f'Verificação da assinatura: verdadeira\n')
    else:
        print(f'Verificação da assinatura: falsa\n')

    if (verified_pdf):
        print(f'Verificação da assinatura do PDF: verdadeira\n')
    else:
        print(f'Verificação da assinatura do PDF: falsa\n')


if __name__ == '__main__':
    main()
```

Fontes

- Geração de primos aletórios: https://www.geeksforgeeks.org/how-to-generate-large-prime-numbers-for-rsa-algorithm/
- Biblioteca de hash: https://docs.python.org/3/library/hashlib.html
- OAEP: https://docs.python.org/3/library/hashlib.html
- RSA-OAEP: https://gist.github.com/ppoffice/e10e0a418d5dafdd5efe9495e962d3d2
- Miller test: https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/