Implementação de Criptografia RSA com Assinatura Digital

Lucas Alves Rodrigues
200022784

Jean Bueno Karia
21105529

Github link
https://github.com/Jibabk/SegCompSeminario

1 Introdução teórica

A criptografia assimétrica, como o RSA, é importante para a segurança dos dados. Este trabalho implementa a geração de chaves, criptografia, descriptografia e assinatura digital com RSA, usando o OAEP para aumentar a segurança. A assinatura digital garante que a mensagem seja autêntica e não tenha sido alterada, criando um resumo (hash) da mensagem e criptografando com a chave privada. O programa cobre a geração das chaves, a assinatura e a verificação, usando RSA, OAEP, SHA-256 e Base64.

1.1 SHA3-256

O SHA3-256 é uma função que transforma uma entrada de qualquer tamanho em uma saída de tamanho fixo. Essa saída é chamada de valor hash ou resumo. O SHA3-256 pode ser combinado com RSA e OAEP para criar sistemas de assinatura digital seguros e eficientes, garantindo que as mensagens não sejam adulteradas e possam ser verificadas de forma confiável.

1.2 OAEP

O OAEP é um esquema de preenchimento que fortalece a segurança do algoritmo RSA ao evitar ataques baseados em análise de plaintext . Ele adiciona aleatoriedade à mensagem antes da criptografia, garantindo que o mesmo texto claro nunca gere o mesmo texto cifrado. O OAEP utiliza uma função de geração de máscara (MGF1) para transformar a mensagem original em um formato seguro antes da aplicação do RSA.

1.3 Assinatura RSA

O RSA é um dos algoritmos de criptografia assimétrica mais conhecidos e utilizados no mundo. Ele se baseia na dificuldade de fatoração de números primos grandes, tornando a quebra da criptografia inviável sem o conhecimento da chave privada. A assinatura RSA funciona criptografando o hash da mensagem com a chave privada do remetente. O receptor então, usa a chave pública para verificar se a mensagem não foi alterada e veio do remetente certo.

1.4 O Programa

O programa é dividido em três partes:

- Parte I: Geração das chaves pública e privada, com primalidade verificada por Miller-Rabin.
- Parte II: Criação e formatação da assinatura digital com a chave privada, utilizando o hash da mensagem.
- Parte III: Verificação da assinatura, usando a chave pública para validar o hash e garantir a integridade da mensagem.

2 Geração das chaves pública e privada

A geração das chaves pública e privada no RSA envolve alguns passos importantes que garantem a segurança do processo.

2.1 Gerar dois números primos grandes

Feito pela função `getPrime()`, que gera um número aleatório de 1024 bits e verifica sua primalidade usando o teste de Miller-Rabin:

Listing 1: Geração de números primos

```
def getPrime():
    while True:
        print("Calculando primo...")
        prime = secrets.randbits(1024)
        if miller_rabin(prime, 40):
            return prime
```

2.1.1 Teste de Miller-Rabin

O teste de Miller-Rabin é um método probabilístico para verificar se um número é primo. Ele repete um teste de "testemunha" k vezes, reduzindo a chance de falso positivo. Cada rodada possui probabilidade de erro $\leq 25\%$. Com 40 iterações, a probabilidade de falso positivo é considerada segura ($\leq 2^{-80}$).

Listing 2: Implementação do Miller-Rabin

```
def miller_rabin(n, k):
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

2.1.2 Função nextprime

Depois de gerar um primo de 1024 bits através da função acima, é utilizada a função nextprime() da biblioteca sympy para se gerar o segundo numero primo

Listing 3: Geração do segundo primo

```
from sympy import nextprime
...
prime= getPrime()
prime2= nextprime(prime)
```

2.2 Geração das Chaves

No RSA, a geração das chaves envolve os seguintes passos:

1. Escolha de dois números primos grandes p e q.
2. Cálculo do módulo RSA: n = p * q.
3. Cálculo da função totiente de Euler: ϕ = (p - 1) * (q - 1).
4. Escolha do expoente público e, que deve ser um número coprimo à ϕ
5. Cálculo do expoente privado d, tal que d $\equiv e^{-1}(mod\phi)$.
6. A chave pública é composta por (n, e), enquanto a chave privada é (n, d).

A segurança do RSA está na dificuldade de fatorar n e obter p e q, tornando inviável descobrir d sem conhecê-los.

2.2.1 Chave pública

Para a chave pública, no código é calculado ϕ e escolhido um número aleatório $1 > e > \phi - 1$ que possua um maior divisor comum igual a 1 com ϕ

Listing 4: Obtenção da chave pública

```
def getPublicKey(prime, prime2):
    pi_n = (prime-1)*(prime2-1)
    while True:
        print("Calculando␣chave␣publica...")
        publicKey = randint(2, pi_n-1)
        if math.gcd(publicKey, pi_n) == 1:
            return publicKey
```

2.2.2 Chave privada

Já a chave privada é obtida através do cálculo: $\equiv e^{-1}(mod\phi)$.

Listing 5: Obtenção da chave privada

```
def getPrivateKey(publicKey, prime, prime2):
    phi = (prime-1)*(prime2-1)
    return pow(publicKey, -1, phi)
```

3 Assinaturas

3.1 Cálculo de hashes da mensagem em claro

A mensagem em claro é processada pela função de hash SHA-3-256, que gera um resumo criptográfico de 256 bits (32 bytes).,

Listing 6: Obtenção do hash da mensagem

```
    hash_sha3_256 = hashlib.sha3_256(message).digest()
```

3.1.1 Por que SHA-3?

O SHA-3 (Keccak), estabelecido pelo padrão `NIST FIPS 202`, oferece:

- Resistência comprovada contra ataques de colisão e pré-imagem
- Eficiência contra vulnerabilidades conhecidas em funções hash tradicionais

3.1.2 Saída Fixa

A função garante:

- Tamanho constante de `256 bits` para qualquer entrada
- Uniformidade necessária para o processo de assinatura digital

3.2 Encodificação do OAEP

1. Hash do rótulo (l_hash): Calcula-se o hash de um rótulo opcional associado à mensagem. Que em nosso programa é vazio.
2. Formação do bloco de dados (DB): O l_hash é concatenado com um preenchimento PS, um byte 0x01 e a mensagem original.
3. Geração de um seed aleatório: Um valor aleatório é gerado para ofuscar o bloco de dados.
4. Aplicação da função MGF1: A função MGF1 é utilizada para gerar uma máscara para seed e DB, e então é utilizado uma função xor para mascarar-los.
5. Construção do bloco final: O bloco final é composto por um byte 0x00, o seed mascarado e o bloco de dados mascarado.

Listing 7: Encoding OAEP

```
def oaep_encode(mensagem, n, e):
    k = (n.bit_length() + 7) // 8
    if len(mensagem) > k - 2 * hashlib.sha3_256().digest_size - 2:
        raise ValueError("Mensagem muito longa")

    l_hash = hashlib.sha3_256(b"").digest()
    l_hash_len = len(l_hash)

    #zero octets padding
    ps = b"\x00" * (k - len(mensagem) - 2 * l_hash_len - 2)

    #Concatenate
    db = l_hash + ps + b"\x01" + mensagem

    # Generate a random octet string seed of length hLen
    seed = os.urandom(l_hash_len)

    #mask generation function
    db_mask = mgf1(seed, k - l_hash_len - 1)
    masked_db = bytes(a ^ b for a, b in zip(db, db_mask)) #xor
    seed_mask = mgf1(masked_db, l_hash_len)
    masked_seed = bytes(a ^ b for a, b in zip(seed, seed_mask)) #xor

    return b"\x00" + masked_seed + masked_db
```

3.3 Processo de Assinatura Digital e Formatação do resultado

Após a geração do hash com SHA-3 e mascaramento por meio do OAEP, a assinatura digital é implementada conforme o seguinte fluxo:

3.3.1 Fluxo para a assinatura assimétrica

Primeiramente, é lido do arquivo "message"a mensagem a ser assinada, juntamente com "n", "privateKey"e "publicKey"gerados previamente pelo programa "Keys.py".

Listing 8: Inputs da função de assinatura.

```
# 1. Carregar mensagem e chaves
message = ler_arquivo("message").encode('utf-8')
n = int(ler_arquivo('N'))
privateKey = int(ler_arquivo('privateKey'))
publicKey = int(ler_arquivo('publicKey'))
```

Então é calculado o hash dessa mensagem utilizando o sha3-256, e logo em seguida esse hash é passado para a função de codificação o OAEP para somente então o resultado dessa função ser codificado utilizando o RSA, e por fim codificado em base64. Finalmente o resultado é salvo no arquivo "output.txt".

Listing 9: Fluxo da função de assinatura.

```
# 2. Calcular hash SHA3-256
hash_sha3_256 = hashlib.sha3_256(message).digest()

# 3. Converter hash para inteiro
hash_int = int(hash_sha3_256.hex(),16)

oaepEncode = oaep_encode(hash_sha3_256, n, privateKey)


oaepEncode_int = os2ip(oaepEncode)



# 4. Aplicar assinatura RSA: S = (hash^d) mod n
assinatura = pow(oaepEncode_int, privateKey, n)


# 5. Codificar resultado em Base64
assinaturaI2OSP = i2osp(assinatura, (n.bit_length() + 7) // 8)


assinatura_b64 = base64.b64encode(assinaturaI2OSP).decode('utf-8')



with open("output.txt", 'w') as arquivo:
    arquivo.write(assinatura_b64)

print("Hash encriptado com sucesso")
```

4 Verificação de Assinatura Digital

O processo de verificação segue três etapas principais para validar a autenticidade e integridade da mensagem.

4.1 Parsing do Documento Assinado

Primeiramente se é calculado o hash sha3-256 da mensagem texto original

Listing 10: Leitura e decodificação da assinatura.

```
message = ler_arquivo("message").encode('utf-8')
hash_sha3_256 = hashlib.sha3_256(message).digest()
```

4.2 Decifração da Assinatura

Em seguida é lido o hash em base64 criptografado gerado pelo arquivo "hash.py", juntamente com a chave pública e N. E então é chamado a função de decodificação do OAEP, que também realiza a descriptografia do RSA.

Listing 11: Decriptografia do hash assinado.

```
oaepEncoded64 = ler_arquivo("output.txt").encode('utf-8')
oaepEncoded = base64_para_bytes(oaepEncoded64)
n = int(ler_arquivo('N'))
publicKey = int(ler_arquivo('publicKey'))

oaepDecoded = oaep_decode(oaepEncoded, n, publicKey)
```

4.2.1 Função de decodificação do OAEP

Na decodificação, os passos são o inverso da codificação:

1. Descriptografação com RSA: O texto cifrado é convertido novamente para um bloco formatado usando a formula $C^e mod\ n$.
2. Recuperação do seed: A máscara é removida usando 'MGF1'.
3. Recuperação do bloco de dados: O seed original é usado para remover a máscara aplicada ao bloco de dados.
4. Validação do l_hash e extração da mensagem: A integridade do preenchimento é verificada antes da extração da mensagem original.

Listing 12: Decodificação do OAEP.

```
def oaep_decode(c, n, d):
    k = (n.bit_length() + 7) // 8

    if k < 2 * hashlib.sha3_256().digest_size + 2:
        raise ValueError("Decodificacao falhou")


    c_int = os2ip(c)

    if c_int >= n:
        raise ValueError("Decodificacao falhou")

    m_int = pow(c_int, d, n)


    m = i2osp(m_int, k)

    l_hash = hashlib.sha3_256(b"").digest()
    l_hash_len = len(l_hash)


```

```
    masked_seed = m[1:l_hash_len + 1]


    masked_db = m[l_hash_len + 1:]


    seed_mask = mgf1(masked_db, l_hash_len)
    seed = xor(masked_seed, seed_mask)
    db_mask = mgf1(seed, k - l_hash_len - 1)
    db = xor(masked_db, db_mask)

    l_hash_prime = db[:l_hash_len]



    if l_hash_prime != l_hash:
       raise ValueError("Decodificacao falhou")
    i = l_hash_len
    while i < len(db):
        if db[i] == 1:
            i += 1
            break
        elif db[i] != 0:
            raise ValueError("Decodificacao falhou")
        i += 1
    return db[i:]
```

4.3 Verificação de Integridade

Por fim, o Hash calculado da mensagem original e o hash obtido pela decodificação do OAEP são comparados. Caso iguais, temos a confirmação da assinatura, caso contrário não podemos garantir a autoridade nem a integridade da mensagem.

Listing 13: Comparação dos hashes.

```
print("Hash original:   ", hash_sha3_256)

print("Hash decriptado: ", oaepDecoded)

if oaepDecoded == hash_sha3_256:
    print("Hashes iguais")
else:
    print("Hashes diferentes")
```

5 Funções auxiliares

Por fim, durante toda a implementação da função de assinatura, se foram utilizadas funções cruciais para o tratamento de dados entre bytes, inteiros e base64.

Listing 14: Funções de formatação

```

def os2ip(x: bytes) -> int:
    return int.from_bytes(x, byteorder='big')

def i2osp(x: int, xlen: int) -> bytes:
    return x.to_bytes(xlen, byteorder='big')

def xor(data: bytes, mask: bytes) -> bytes:
    '''Byte-by-byte XOR of two byte arrays'''
```

```
    masked = b''
    ldata = len(data)
    lmask = len(mask)
    for i in range(max(ldata, lmask)):
        if i < ldata and i < lmask:
            masked += (data[i] ^ mask[i]).to_bytes(1, byteorder='big')
        elif i < ldata:
            masked += data[i].to_bytes(1, byteorder='big')
        else:
            break
    return masked

def int_para_base64(numero):
    numero_bytes = numero.to_bytes((numero.bit_length() + 7) // 8, byteorder='
        ↪ big')
    numero_base64 = base64.b64encode(numero_bytes).decode('utf-8')
    return numero_base64

def base64_para_int(base64_str):
    numero_bytes = base64.b64decode(base64_str)
    return int.from_bytes(numero_bytes, byteorder='big')
```

6 Conclusão

A combinação de RSA, OAEP e SHA3-256 oferece um alto nível de segurança para comunicações e assinaturas digitais. O OAEP impede ataques de texto cifrado escolhido, o RSA permite a troca segura de informações e o SHA3-256 assegura a integridade dos dados. Essa abordagem é amplamente utilizada em sistemas modernos para garantir a privacidade e autenticidade das informações. De forma geral esse projeto permitiu aos alunos aprofundar seus conhecimentos em criptografia, compreendendo na prática o funcionamento de algoritmos como RSA, OAEP e SHA3-256. Além disso, possibilitou o desenvolvimento de habilidades essenciais para a área de segurança da informação, incluindo a geração e manipulação de chaves criptográficas, aplicação de funções hash e técnicas de preenchimento seguro.