Signature Generator and Verifier

January 26, 2025

1 Gerador e Verificador de Assinaturas Virtuais

Geradores de assinaturas digitais são ferramentas as quais garantem a autenticidade e/ou a integridade de uma dada mensagem. Para isso, utiliza-se técnicas de criptografia, normalmente criptografia assimétrica, onde uma chave privada é empregada para assinar digitalmente um dado, e o hash da mensagem é incorporado na assinatura como uma forma de validação do conteúdo.

Já verificadores de assinaturas são responsáveis por confirmar a autenticidade e/ou a integridade de uma assinatura digital. Considerando um cenário de criptografia assimétrica como o dado acima, o verificador utiliza a chave pública correspondente do emissor para decifrar a mensagem cifrada, caso a mensagem realmente seja do emissor declaro, o hash deste texto em claro deve ser correspondente ao hash enviado pelo o emissor.

Assim, neste notebook iremos implementar um gerador e verificador de assinatura digitais de documentos, utilizando técnicas avançadas em segurança da computação. Para isso, o projeto consta com três módulos principais: geração de chaves, encriptação e decriptação, assinatura e verificação.

O primeiro módulo será responsável pela geração de chaves criptográficas com base no algoritmo RSA, onde essas chaves serão derivadas de números primos de 1024 bits. O segundo módulo dedica-se ao processo de encriptação e decriptação de mensagens utilizando o OAEP (Optimal Asymmetric Encryption Padding). O terceiro realiza o cálculo do hash da mensagem e formatação do resultado em BASE64. Por fim, o quarto módulo sumariza a verificação de assinaturas digitais com um exemplo prático.

1.1 Geração de Chaves

Primeiramente começaremos nosso projeto com a parte mais crucial nos sistemas modernos de criptografia, a geração das chaves. As chaves são responsáveis por garantir a confidencialidade e a integridade de qualquer algoritmo de criptografia moderno, então é necessário uma alta atenção para se evitar qualquer tipo de padrão ou rastreabilidade em sua geração. Para isso, usaremos duas bibliotecas que irão nos trazer a aleatoriedade necessária para a geração das Chaves.

[1]:
```
from random import getrandbits, randrange
```

O algoritmo que será responsável pela encriptação e decriptação do nosso projeto será o algoritmo de criptografia assimétrica RSA, amplamente reconhecido por sua segurança e aplicabilidade em sistemas modernos de criptografia assimétrica.

Dito isso, por ser assimétrico, iremos gerar duas chaves, uma pública e uma privada. A chave pública será composta por n=p×q, onde p e q são números primos com no mínimo 1024 bits e por "e", o qual será um número inteiro escolhido que satisfaça a propriedade 1< "e" < (n) e sendo

coprimo de (n) = (p−1)×(q−1). Já a chave privada será representada por “d”, calculado como “d” “e” ^{1} (mod (n)), sendo “d” o inverso modular de “e” em relação a (n).

Com isto, o primeiro passo será gerar os primos q e p. Faremos isso usando o teste de primalidade de Miller–Rabin, o que nos retornará dois primos diferentes entre si com uma alta probabilidade.

[2]:
```python
def is_prime(n: int, k=128) -> bool:
    """Teste de primalidade usando o algoritmo de Miller-Rabin."""
    s = 0
    r = n - 1
    while r & 1 == 0:
        s += 1
        r //= 2

    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, r, n)
        if x != 1 and x != n - 1:
            j = 1
            while j < s and x != n - 1:
                x = pow(x, 2, n)
                if x == 1:
                    return False
                j += 1
            if x != n - 1:
                return False
    return True

def get_prime():
    """Gera um número primo de 1024 bits."""
    while True:
        p = getrandbits(1024)
        p |= 1  # Garante que o número é ímpar
        p |= 1 << 1023  # Garante o tamanho correto (1024 bits)
        if is_prime(p):
            return p

p = get_prime()
q = get_prime()

while p == q:
    q = get_prime()
```

Com p e q gerados, iremos definir algumas funções auxiliares para calcularmos a operação mdc e os valores de n e phi.

[3]:
```python
def gcd(a, b):
    """Calcula o máximo divisor comum (GCD) usando o algoritmo de Euclides."""
```

```
    while b:
        a, b = b, a % b
    return a

def get_n(p, q):
    """Calcula o valor de n = p * q."""
    return p * q

def get_phi(p, q):
    """Calcula o valor de phi = (p - 1) * (q - 1)."""
    return (p - 1) * (q - 1)

n = get_n(p, q)
phi = get_phi(p, q)
```

Agora, iremos definir duas função que irão encapsular as equações de geração de “e” e “d”

[4]:
```
def choose_e(phi):
    """Escolhe um valor de 'e' que seja coprimo a 'phi'."""
    e = 2
    while e < phi and gcd(e, phi) != 1:
        e += 1
    return e

def get_d(e, phi):
    """Encotra o inverso modular de 'e'."""
    return pow(e, -1, phi)

e = choose_e(phi)
d = get_d(e, phi)
```

Pronto, com “n”, “e” e “d” estamos pronto para gerarmos nossas chaves com a função generate_keys();

[5]:
```
def generate_keys(n, e, d):
    """Gera as chaves pública e privada."""
    public_key = (e, n)
    private_key = (d, n)

    return public_key, private_key

pk, sk = generate_keys(n, e, d)

print(f"Chave Pública: {pk}. \n Chave privada: {sk}")
```

```
Chave Pública: (5, 24667051978200185482033153549338770698261146792563203937666163261330035287174060739130361086121726996407049465002783270391441311903405118651137988235812500335581447239110591208891397255406896613056257967438634227057894898764525245749141225594283683250931214380047533667745853145798130855953748231127 40
```

```
235260249069824051876111667422163397063635960441355538421642510880308595441831590
039960716461180941854219727692872656037099702793192998976048149836067416994409957
731459915079648166619692862925615902439493326376084971329663830028875869565822174
415074007463482040475236979537158678623739952792459892072943820445262528143 3).
 Chave privada: (148002311869201112892198921296032624189566880755379223625996979
567980211723044364434782166516730361978442296790016699622348647871420430711906827
929414875002013488683434663547253348383532441379678337547804631805362347369392587
151474494847353565702099505587286280285202006475118874788785135722489386764414
115613053397411909163003589094784824643256894085212346958606374633194194801182395
960201420084457194596201219747741384664540294645861862562978539500319229995752394
639228278951396239023987390823678162464081117856958972746608662927060439551238401
572722802397459389487048814108454001896826005187525804399047602640912096, 2466705
197820018548203315354933877069826114679256320393766616326133003528717406073913036
108612172699640704946500278327039144131190340511865113798823581250033558144723911
059120889139725540689661305625796743863422705789489876452524574914122559428368
325093121438004753366774585314579813085595374823112740235260249069824051876111667
422163397063635960441355538421642510880308595441831590399607164611809418542197276
928726560370997027931929989760481498360674169944099573145991507964816661969286292
561590243949332637608497132966383002887586956582217415074007463482040475236979537
158678623739952792459892072943820445262528143 3)
```

1.2 Encriptação e Decriptação

Agora, já em porte da chave pública e a privada, podemos avançar para o processo de encriptação e decriptação do algoritmo RSA. A encriptação será realizada com a chave privada (sk), seguindo a fórmula C = M^"e" mod n, onde M é a mensagem original, d é o expoente da chave privada e n é o produto dos números primos gerados. Já para a decriptação, utilizaremos a chave pública (pk), seguindo essa fórmula: M = C^"d", onde C é o texto cifrado e "e" é o expoente da chave pública.

1.2.1 RSA

Para encapsular a lógica dessas fórmulas, criaremos uma classe chamada RSA que irá gerenciar as operações de encriptação e decriptação.No método de inicialização da classe iremos passar as chaves pública e privada já definidas.

[6]:
```python
class RSA:
    def __init__(self):
        self.public_key, self.private_key = pk, sk

    def encrypt(self, message: int):
        """Criptografa uma mensagem usando a chave privada."""
        d, n = self.private_key
        result = pow(message, d, n)  # (m^d) mod n
        return str(result)

    def decrypt(self, message: int):
        """Descriptografa uma mensagem usando a chave pública."""
        e, n = self.public_key
```

```
        result = pow(message, e, n)  # (c^e) mod n
        return str(result)
```

1.2.2 OAEP

Mas, antes de aplicarmos diretamente o RSA na nossa mensagem a ser encriptada, primeiramente, iremos utilizar uma técnica de padding pseudo aleatória, OAEP (Optimal Asymmetric Encryption Padding), com o intuito de aumentar a segurança da nossa criptografia tornando mais difícil técnicas baseadas em análises matemáticas e/ou estruturais, as quais podem possuir alguma efetividade em mensagens com tamanho muito pequeno.

O algoritmo funciona mesclando a mensagem original com um valor pseudo aleatório r de tamanho de 64 bytes, esse r servirá para gerarmos x e y que serão a base da nossa nova mensagem com seu padding. Já o algoritmo de decriptação é determinismo, o qual realiza operações reversas para obter a mensagem original. Vale ressaltar que esse algoritmo não traz segurança a mensagem, qualquer um que tenha acesso a cifra será capaz de fazer o processo reverso, ele apenas serve como técnica de preenchimento de mensagens.

[7]:
```python
from hashlib import sha3_512
from os import urandom

class OAEP:
    k0 = 512
    k1 = 256

    def __init__(self):
        self.x = None
        self.y = None

    def sha3_512(self, data: bytes) -> bytes:
        """Aplica SHA3-512 e retorna o hash como bytes."""
        return sha3_512(data).digest()

    def encrypt(self, message: int) -> int:
        """Encriptação do algoritmo OAEP"""
        r = urandom(64)

        message = message << self.k1

        x = message ^ int.from_bytes(self.sha3_512(r))

        y = int.from_bytes(r) ^ int.from_bytes(self.sha3_512(x.to_bytes(128)))

        return (x << self.k0) | y

    def decrypt(self, ciphertext: int) -> int:
        """Decriptação do algoritmo OAEP"""
```

```
        y = ciphertext & ((1 << self.k0) - 1)
        x = ciphertext >> self.k0

        r = y ^ int.from_bytes(self.sha3_512(x.to_bytes(128)))

        pm = x ^ int.from_bytes(self.sha3_512(r.to_bytes(self.k0 // 8)))

        return pm >> self.k1
```

1.3 Assinatura, Transmissão e Verificação de Assinatura

Agora que já temos ferramentas o suficiente para realizarmos a geração das chaves, encriptação e decriptação, iremos nos preocupar com questionamentos fundamentais na nossa aplicação: Como podemos assinar a mensagem? Como podemos transmitir essa mensagem de uma maneira adequada? Como podemos verificar nossa assinatura?

1.3.1 Assinatura

Para assinar digitalmente nossa mensagem a ser transmitida, utilizaremos uma função de compactação de dados, mais especificamente a função SHA3-512. Uma função de compactação é um algoritmo que dado uma entrada de dados de qualquer tamanho, sua saída será de tamanho fixo, a qual é chamada de hash. Utilizaremos essa função de compactação, pois ela possui a propriedade de que a computação reversa do hash para a mensagem original é extremamente custosa computacionalmente.

Assim, a nossa estratégia de assinatura será transmitir nossa mensagem e o seu hash, porém este hash estará criptografado pelo algoritmo RSA. Com isso, ao realizar a decriptação utilizando a chave pública do emissor, o receptor poderá fazer o hash da mensagem recebida e comparar com o hash decriptado, e, caso eles sejam iguais, o emissor terá a garantia que a mensagem é realmente do seu emissor, já que apenas ele possui acesso a sua chave privada capaz de gerar a encriptação correta.

Nesta parte do nosso projeto, utilizaremos a biblioteca "hashlib" para o desenvolvimento da nossa função Hash, sendo a única parte do nosso projeto que utilizará código de terceiros.

Função Hash

[8]:
```
from hashlib import sha3_512
class sha3:
  @staticmethod
  def hash_512(message: str):
    if type(message) == int:
      message = str(message)
    return sha3_512(message.encode()).hexdigest()
```

1.3.2 Transmissão

Para a transmissão da nossa cifra, primeiramente, iremos converter a cifra para o formato Base64. Isso será feito, a Base64 é uma forma de codificação que transforma dados binários em uma sequência de caracteres ASCII, o que normalmente é um formato mais adequado a transmissão de dados.

Nossa implementação é um modelo simplificado da Base64, mas suficientemente robusto para o escopo deste projeto.

BASE64

[9]:
```
class base_convert:

  #converte para base 64
  @staticmethod
  def format(message: int):
    #Usa uma string de referencia
    base64_chars =␣
↪"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    res = ""
    #Enquanto a mensagem nao se tornar 0 realiza o metodo de divisoes sucessivas
    while message > 0:
      remainder = message % 64
      res = base64_chars[remainder] + res
      message = message // 64
    return res


  @staticmethod
  def parse(base64_message: str):
    #Tambem usa uma string de referebncia
    base64_chars =␣
↪"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    res = 0
    # procura o caracter encontrado na string e coloca e soma com o resultado␣
↪multiplicado por 64
    # essa multiplicacao por 64 fara com que os termos sejam multiplicados por␣
↪64 o numero correto de vezes
    for char in base64_message:
      res = res * 64 + base64_chars.index(char)
    return res
```

1.3.3 Verificação com Exemplo Prático

Agora, finalmente partiremos para a demonstração do nosso projeto. Nesta demonstração, geramos nosso plain text utilizando o lorem ipsum, um gerador randômico de texto.

O processo começa com a transformação do texto em números, utilizando a tabela ASCII para converter cada caractere em um valor numérico. Em seguida, calculamos o hash da mensagem para criar uma representação única e compacta do texto. Após isso, utilizamos o padding OAEP sobre o hash da mensagem para adicionar segurança à encriptação. Com o hash preparado, realizamos a sua criptografia utilizando a chave privada do algoritmo RSA. Por fim, para realizarmos a transmissão, concatenamos a mensagem ao hash criptografado e assim, colocamos o resultado da concatenação em Base64.

Já o processo de verificação é o reverso no processo de assinatura. Primeiramente, desfazemos o

formato Base64, separamos a mensagem do hash criptografado e realizamos a decriptação com a chave pública do emissor e retiramos o padding resultado do OAEP. Assim, ao final dessa série de processos, temos dois dados fundamentais, a mensagem e o hash do emissor, caso o hash do emissor seja compatível com o hash da mensagem, realmente essa mensagem é do emissor declarado, garantindo assim confiabilidade a origem dessa mensagem, caso não seja, concluímos que ou o hash ou a mensagem foram interceptadas e trocadas durante a nossa transmissão, logo, ambas devem ser invalidadas e descartadas.

[10]:
```
# Instanciação da classe RSA
cy = RSA()
oaep = OAEP()

plain_text = """
    Veniam sint in nulla eiusmod esse proident magna pariatur ea fugiat ipsum.␣
 ↪Ut cupidatat aliqua amet Lorem consequat
    amet eu anim. Id ad ut voluptate quis tempor nisi sunt esse consectetur.␣
 ↪Ullamco cupidatat sit commodo minim amet nulla
    sit. Tempor occaecat ad occaecat minim irure. Incididunt nostrud sunt ea␣
 ↪culpa reprehenderit esse sunt Lorem id ea et
    cillum tempor. Magna excepteur labore dolore Lorem esse do  adipisicing␣
 ↪aliquip culpa pariatur laboris deserunt consequat.
"""

plain_text = int(''.join([str(ord(c)) for c in plain_text]))

# Calculo do hash 256 da mensagem
h = int(sha3.hash_512(str(plain_text)), 16)

# Encryptação desse hash
e = oaep.encrypt(h)
e = cy.encrypt(int(e))

# Concatenação entre plain_text e o hash encriptado
message_to_encode = str(plain_text) + str(e)

# Conversão para a Base64
encoded_message = base_convert.format(int(message_to_encode))

print("Mensagem codificada em base64:", encoded_message)

# Parse da base64
decoded_message = str(base_convert.parse(encoded_message))

# Separação entre o plain_text e o hash encriptado
decoded_plain_text = decoded_message[:len(str(plain_text))]

encrypted_hash = decoded_message[len(str(plain_text)):]
```

```python
# Decryptação do hash encriptado
decrypted_hash = cy.decrypt(int(encrypted_hash))
decrypted_hash = oaep.decrypt(int(decrypted_hash))

# Comparação entre os hashs para verificação da assinatura
print("Hash original:", h)
print("Hash decodificado:", decrypted_hash)
print("Verificação de assinatura:", int(decrypted_hash) == h)
```

```
Mensagem codificada em base64: QH0aSsTH4hWJhAM4nH2uKZGuPujU7FL4vdyRTFO6CoVWHlsf4
hu+mxnVK5q2tnRuzSF8qHyPG7QTQ4xKYtws0niaZnghEkv7LUBIN+cfwfNv/OJJ9fh+zKBqiPkkXGyjx
gzjHIxPHEAyTDH2qRLPRpgZyLxAh/Sp1zj0RyLmuajRJCcJc2bL1djmQV6TM6bbmErXuIFU5Ibw8TzZv
IStNwIW8W9UXTmJeg/wCFiqE+s89unTHkfDp+cTRV+Ay0B76iqUF2WxFoue3nj82z7klIUKFlc+sZqhL
cU5p1sGCWsEn/9nk13QMCZ0xgUj7ZkwAfcX70XbA3phglycc4PQqnUNC95bHhNIsSc0HtGYubVTAd4gZ
ZKD9ny0fSmYyM9Ds1m+a1vWUjX2h/vNmnWU8bPDJ0TiapmOmrZrHQyRoHir+EeSUdcMHvs8EeUcpwUUs
/ImOTpN+y4989bUA02L3JQz+GqMIME4n6EsOISu4PZhFY9U3FOqJrnD8zFlzRrirj1z8Pss8/dm7qRI+
Wdk8T9nTf7HDIirD/C3tHHsH+UOuyWEAUf7p+jildqVrCkBYKEdqe5Yo2cf1AwiMBRpqsHlPiE1u0Vqk
Gc6VGSnmhftBGqZVHMMTnShosLteqi0mMTzg8pOZhWhoT9LtKBy5mGNNxMy/TI3EptNPOiny7ncQJXN/
piw67LkxSrqPgqp3J87BBmeg30SlNST9yu790hJ9vxHLCjL0p7TH/o9k+1iXnUEcoS/zIu+RCbIuUrdN
2pp7tGsRv2vSPn9UR7mryZ+AvAceF1ETAyK0c1Xe890WuuCoDteSn/cBNPKdJlo8WhN3yXy6ZlrJrG/l
7qxvR/EYBQzVFZYMYsMAEOgG9vcrrN277L/gcLvTPhYEKCiR+SL9ax9UR2+RKbc86u80W3HAHYE3ftu9
aeIoBj3NJqMxYuxaVWhF4F4JYJNhIMMytsAoV6YwObCiDyJqo6DUUuRs5tUtYREe6MqOKa40XXaYvIx5
Uwm4NHLnRunhkrVgNhEa9OJVXvK5b42SE60KYqyH7T3ey4ack9KZ2RBBiyJpX
Hash original: 1155051126737645498871128736145598639295399036438724922621922031120194494853405559452377866588951415855944916984931703532205365155166033836557800
8703094974
Hash decodificado: 1155051126737645498871128736145598639295399036438724922621922031120194494853405559452377866588951415855944916984931703532205365155166033836557800
78008703094974
Verificação de assinatura: True
```