Trabalho de Implementação
Gerador/Verificador de Assinaturas

Nicole de Oliveira Sena *

Universidade de Brasília, 28 de agosto de 2024

Resumo

Este relatório descreve a implementação de um gerador e verificador de assinaturas RSA em arquivos. Será dividido em 3 etapas: Geração de chaves e cifra, Assinatura e Verificação.

Palavras-chave: Assinatura RSA · OAEP ·

1 Introdução

RSA (Rivest-Shamir-Adleman) é um dos primeiros sistemas de criptografia de chave pública e é amplamente utilizado para transmissão segura de dados. Neste sistema de criptografia, a chave de encriptação é pública e é diferente da chave de decriptação que é secreta (privada).[1]

No RSA, esta assimetria é baseada na dificuldade prática da fatorização do produto de dois números primos grandes, o "problema de fatoração".

As etapas dessa implementação serão as seguintes:

- Etapa I: Geração de chaves e cifra
  - Geração de chaves (p e q primos com no mínimo de 1024 bits)
  - Cifração/decifração assimétrica RSA usando OAEP.
- Etapa II: Assinatura
  - Cálculo de hashes da mensagem em claro (função de hash SHA-3)
  - Assinatura da mensagem (cifração do hash da mensagem)
  - Formatação do resultado (caracteres especiais e informações para verificação em BASE64)
- Etapa III: Verificação:
  - Parsing do documento assinado e decifração da mensagem (de acordo com a formatação usada, no caso BASE64)
  - Decifração da assinatura (decifração do hash)
  - Verificação (cálculo e comparação do hash do arquivo)

2 Geração de Chaves

Nesse primeiro momento, serão gerados 2 numeros primos "p"e "q"com no mínimo 1024 bits, que irão passar por uma verificação utilizando o teste de primalidade Miller-Rabin.

Miller-Rabin é um teste probabilístico da primitividade de um dado número n. Se um número n não passar pelo teste, n com certeza é um número composto (ou seja, não-primo), o algoritmo utilizado na implementação deste trabalho foi retirado de Ayrx [**?**]).

Seguindo a implementação, foi utilizado "p"e "q"para gerar "n"(n=p*q), o que gera também, com ajuda de f(n) = (p-1)(q-1) os números ”e” e ”d”. Por fim, "d" e ”n” geram a chave privada e ”e” e ”n” para geram a chave publica.

```
        def gera_primos():
            n = random.getrandbits(1024)
            if miller_rabin(n,40) == True:
                return n
            return gera_primos()
```

A função encontraInversoMod(a, m) calcula o inverso modular de a em relação a m, que é o número x tal que a multiplicação de a por x, modulo m, resulta em 1. Para que isso seja possível, a e m devem ser coprimos, ou seja, o Máximo Divisor Comum (MDC) entre eles deve ser 1. O algoritmo utilizado é o Algoritmo de Euclides Estendido, que resolve uma equação linear relacionada ao problema. Quando o loop termina, o valor encontrado é ajustado para garantir que esteja no intervalo correto.

```
        def encontra_inverso_mod(a, m):
        if math.gcd(a, m) != 1:
            return None

        u1, u2, u3 = 1, 0, a
        v1, v2, v3 = 0, 1, m
        while v3 != 0:
            q = u3 // v3
            v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2,
                v3
        return u1 % m
```

*190114860@aluno.unb.br

No processo de geração de chaves RSA, n é o produto de dois números primos distintos, p e q. O valor de n é utilizado como módulo para as operações de cifração e decifração. A função totiente de n, denotada como fn, é calculada a partir de p e q. O expoente público e é escolhido de forma que seja coprimo com fn e dentro de um intervalo específico. O expoente privado d é encontrado como o inverso modular de e em relação a fn. A chave pública RSA é composta por (e, n), e a chave privada é composta por (d, n). Estas chaves são usadas para cifrar e decifrar mensagens, e a segurança do sistema RSA depende da dificuldade de fatorar o número n.

```
n = p*q
fn = (p-1)*(q-1)
e=0

while (math.gcd(fn, e) != 1):
    e = random.randrange(2, fn)

d = encontra_inverso_mod(e, fn)

chave_privada = (d, n)
chave_publica = (e, n)
```

3 Cifração/decifração

Neste momento, será implementada a cifração e decifração assimétrica RSA usando OAEP (Optimal Asymmetric Encryption Padding). OAEP é um esquema de padding que melhora a segurança da cifração RSA, adicionando uma camada extra de proteção.

Primeiramente, o código gera uma semente aleatória. Este valor é utilizado para criar um hash da semente, que é gerado pela função sha256. A semente é convertida para bytes e usada para criar uma digestão criptográfica que será utilizada no processo de cifração e decifração.

Na função CodificaOaepRsa, a mensagem é verificada quanto ao comprimento para garantir que não exceda o tamanho permitido pela chave RSA. Se a mensagem for muito longa, uma exceção é levantada. Em seguida, a mensagem é preenchida (padding) com bytes nulos até o tamanho apropriado e XOR é aplicado com o hash da semente para garantir a segurança adicional. A mensagem resultante é então convertida para um número inteiro e cifrada usando a chave pública e e o módulo n.

```
    def codifica_oaep_rsa(message, n, k):

message_length = len(message)
if message_length > n - k:
    raise ValueError("Menssagem muito longa para tamanho da chave")

hash_seed = sha256(seed.to_bytes((seed.bit_length() + 7) // 8, byteorder='big')).
    digest()

padded_message = message + b'\x00' * (n - message_length - k)
padded_message = bytearray(padded_message)
for i in range(len(padded_message)):
    padded_message[i] ^= hash_seed[i % len(hash_seed)]

return pow(int.from_bytes(padded_message, byteorder='big'), e, n)
```

A função cedodificaOaepRsa realiza o processo inverso: decifra o texto cifrado usando a chave privada d e o módulo n. A mensagem decifrada é então convertida de volta para bytes e o mesmo hash da semente é usado para remover o padding aplicado durante a cifração. Finalmente, os bytes nulos adicionados no padding são removidos, restaurando a mensagem original.

A função DedodificaOaepRsa é responsável por decifrar uma mensagem criptografada usando RSA com padding OAEP. Inicialmente, ela decifra o texto cifrado (ciphertext) usando a chave privada (d) e o módulo (n), convertendo o resultado para uma sequência de bytes que representa a mensagem preenchida. Em seguida, a função remove o padding aplicando uma operação XOR entre os bytes da mensagem e o hash da semente utilizado durante a cifração. O hash da semente é gerado através da função sha256 aplicada à semente convertida em bytes. Após desfazer o padding, a função retorna a mensagem original, removendo os bytes nulos adicionados no processo de padding.

```
    def cedodifica_oap_rsa(ciphertext, d, n, k):

padded_message = pow(ciphertext, d, n)
padded_message = padded_message.to_bytes((n.bit_length() + 7) // 8, byteorder='big
    ')

hash_seed = sha256(seed.to_bytes((seed.bit_length() + 7) // 8, byteorder='big')).
    digest()
original_message = bytearray(padded_message)
for i in range(len(original_message)):
    original_message[i] ^= hash_seed[i % len(hash_seed)]

return original_message.rstrip(b'\x00')
```

4 Assinatura

Neste momento, abordaremos a seção responsável pela criação da assinatura digital. Este processo inclui calcular o hash da mensagem original usando a função SHA-3, assinar o hash da mensagem também através da cifração OAEP, e formatar o resultado final em BASE64, que inclui caracteres especiais e informações necessárias para a verificação. Primeiro, calculamos o hash SHA-3 da mensagem e, em seguida, aplicamos a cifração OAEP sobre o hash gerado. Usamos duas funções determinísticas, G e H, para criar uma máscara. Para obter a máscara de DB, concatenamos "PS", o byte 0x01 e a mensagem "M", e aplicamos a operação XOR junto com a função "G". Finalmente, formatamos o resultado em BASE64.

5 VERIFICAÇÃO

Na fase final do projeto, o processo envolve analisar o documento assinado e decifrar a mensagem de acordo com a codificação utilizada, que neste caso é BASE64. Em seguida, realizamos a decifração da assinatura, o que significa recuperar o hash da mensagem. A etapa seguinte é a verificação, onde calculamos e comparamos o hash obtido com o hash do arquivo original. Para isso, utilizamos funções de decifração que convertem a mensagem cifrada em um número inteiro com a mesma quantidade de bytes do módulo da chave privada. Com a mensagem cifrada representada como um número inteiro, procedemos elevando este número à potência "d"e aplicando o módulo "n". Esse processo nos permite recuperar o hash SHA-3 da mensagem original.

REFERÊNCIAS

[1] Wikipédia. *RSA (sistema criptográfico)* - Wikipédia, a enciclopédia livre. Disponível em: https://pt.wikipedia.org/wiki/RSA_(sistema_criptogr%C3%A1fico). Acesso em: 30/08/2024

[2] GitHub Gist. *Python implementation of the Miller-Rabin Primality Test* Disponível em: https://gist.github.com/Ayrx/5884790. Acesso em: 30/08/2024