Trabalho de Implementação
Gerador/Verificador de Assinaturas

Ítalo Eduardo Dias Frota, 18/0019279
João Victor Cabral de Melo, 16/0127670

[1]Universidade de Brasília - Instituto de Ciências Exatas
Departamento de Ciência da Computação - CIC0201 - Segurança Computacional
2020.2 - Turma A - Professor João José Costa Gondim
Prédio CIC/EST - Campus Universitário Darcy Ribeiro
Asa Norte 70919-970 Brasília, DF

Resumo. *O projeto foi desenvolvido para a disciplina de Segurança Computacional da Universidade de Brasília - (UnB) ministrada no semestre 2020/2. O presente relatório tem como objetivo apresentar o problema proposto e a implementação da dupla para resolução do mesmo.*

1. Introdução

Com o crescimento da internet e a necessidade de transferência de dados com segurança, foram desenvolvidos algoritmos de criptografia para assegurar a integridade dos dados trocados, permitindo uma comunicação segura entre abas as partes. O algoritmo RSA foi descrito no ano de 1977 por Adi Shamir, Leonard Adleman e Ron Rivest.

Este tipo de algoritmo é chamado de criptografia assimétrica, onde há a utilização de duas chaves. A chave pública é utilizada para cifrar os dados enviados. A segunda chave é utilizada para descriptografia dos dados.

2. Procedimentos

2.1. Geração de Chaves com teste de primalidade (Miller-Rabin)

São gerados pares de chaves com uma chave privada e uma chave pública. A chave pública pode ser utilizada para cifrar dados arbitrários e a chave privada pode ser utilizada para decifrar dados cifrados pela chave pública correspondente. Esta estrutura composta por duas chaves difere da criptografia simétrica, onde apenas uma chave é utilizada para criptografar e descriptografar o texto, tornando-se inviável o seu uso na internet, pois a chave precisaria ser transferida junto com a mensagem criptografada, sendo facilmente descoberta por um atacante, que conseguiria ler os dados enviados.

Os passos implementados são os seguintes:

1. Escolha aleatória de dois números primos grandes P e Q, com tamanho mínimo de 1024 bits;
2. Cálculo de N, sendo a multiplicação de P e Q;
3. Função totiente de Euller, também chamada de phi, no N. Como P e Q são primos, phi é P -1 * Q-1 ;
4. Escolha de um número Aleatório E, que tem que satisfazer as condições: ser maior que 1 e menor que Phi(N), e também ser primo entre Phi(N);
5. A chave pública é composta pelo N e o E;
6. A chave privada é chamada de D e é precisa satisfazer o algoritmo: D * E mod Phi(N) == 1;

2.2. Cifração

Para cifrar uma mensagem precisamos dos números N e E que foram descritos anteriormente, pois eles são nossa chave pública, então utilizando uma potenciação modular entre N, E e a mensagem para conseguir gerar uma cifra. O processo de cifração ocorre da seguinte maneira:

Para cifrar, utiliza-se os números N e E, já descritos neste relatório. Com a chave pública preparada, é realizada uma potenciação modular entre N, E e a mensagem. Inicialmente, a mensagem é convertida em um número, então aplicamos**OAEP**, tornando o esquema probabilístico e portanto, aumentando o nível de segurança do RSA.

Então c é computado como a mensagem cifrada: $m^e \equiv c \pmod{n}$

2.3. Decifração

Para recuperar a mensagem da cifra, podemos utilizar a private key com expoente d calculando:

$c^d \equiv (m^e)^d \equiv m \pmod{n}$

onde m é a mensagem sem a cifra.

Então é removido o OAEP padding, fazendo com que a mensagem retorne ao seu formato original. Após a transformação, convertemos o número para string.

2.4. Assinatura

Para assinar uma mensagem, deve-se utilizar a chave privada. Primeiramente é produzido um hash da mensagem elevado a potência de d (mod n). A partir deste hash, é anexada a assinatura na mensagem. Este método facilita a posterior verificação, pois permite analisar se o hash corresponde ao hash da mensagem e se a própria bate com a mensagem recebida.

Isso funciona pelas regras de exponenciação:

$h = hash(m);$

$(h^e)^d = h^{ed} = h^{de} = (h^d)^e \equiv h \pmod{n}$

```python
def sign(message: bytes, privateKey: int, n: int):
    messageHash = sha3_256(message)
    signature = pow(os2ip(messageHash), privateKey, n)
    return i2osp(signature, 1024)
```

Figura 1. Implementação da função de assinatura

2.5. Verificação

Na verificação, utilizamos o mesmo algoritmo de hash em conjunção com a public key. Então, verificamos se o hash enviado e o gerado são os mesmos. Se ambos os dados corresponderem isso significa que a assinatura é válida.

```python
def verify(message: bytes, signature: bytes, publicKey: int, n: int):
    messageHash = sha3_256(message)
    verifying = pow(os2ip(signature), publicKey, n)
    verifying = i2osp(verifying, 32)
    if verifying == messageHash:
        return True
    else :
        return False
```

Figura 2. Implementação da função de verificação

2.6. Base64

Para a formatação do resultado, é necessário codificar os dados em Base64, para isso tomamos a seguinte fórmula:

1. Pegamos o valor ASCII de cada caractere da string
2. Calculamos o equivalente binário 8-bits dos valores ASCII
3. Convertemos os chunks 8-bits em chunks 6-bits através do reagrupamento dos digitos
4. Convertemos os grupos binários 6-bits para seus respectivos valores decimais
5. Por fim, com base numa tabela de codificação base64 indicamos os caracteres em base64 para cada um de seus valores decimais

Também foi implementada uma função de decodificação de dados em Base64, aqui, descobrimos o tamanho do padding e convertemos de 4 em 4 caracteres base64 de volta ao valor ASCII, por fim removemos o padding e conseguimos o dado original.

```python
def encodeBase64(s):

  i = 0
  base64 = ending = ''
  base64chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

  # Adiciona padding se a string não for divisível por 3
  pad = 3 - (len(s) % 3)
  if pad != 3:
    s += "A" * pad
    ending += '=' * pad

  # Itera através da string
  while i < len(s):
    b = 0

    #  De 3 em 3 caracteres, os converte para 4 chars base64
    for j in range(0,3,1):

      #  Pega o código ASCII do próximo caractere na linha
      n = ord(s[i])
      i += 1

      # Concatena os três caracteres juntos
      b += n << 8 * (2-j)

    # Converte os 3 chars para 4 chars base64
    base64 += base64chars[ (b >> 18) & 63 ]
    base64 += base64chars[ (b >> 12) & 63 ]
    base64 += base64chars[ (b >> 6) & 63 ]
    base64 += base64chars[ b & 63 ]

  # Adiciona o padding no final
  if pad != 3:
    base64 = base64[:-pad]
    base64 += ending

  return base64
```

```python
def decodeBase64(s):
  i = 0
  base64 = decoded = ''
  base64chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

  # Substitui o padding com caracteres "A" para o decoder processar a string e salva
  if s[-2:] == '==':
    s = s[0:-2] + "AA"
    padd = 2
  elif s[-1:] == '=':
    s = s[0:-1] + "A"
    padd = 1
  else:
    padd = 0

  #  Digere 4 caracteres por vez
  while i < len(s):
    d = 0
    for j in range(0,4,1):

      d += base64chars.index( s[i] ) << (18 - j * 6)
      i += 1

    # Converte os 4 chars de volta ao ASCII
    decoded += chr( (d >> 16 ) & 255 )
    decoded += chr( (d >> 8 ) & 255 )
    decoded += chr( d & 255 )

  # Remove padding
  decoded = decoded[0:len( decoded ) - padd]

  return decoded
```

Figura 3. Implementação das funções

2.7. Formatação

Para a formatação do resultado, geramos um arquivo .pem contendo a chave privada, a chave pública e a assinatura, todos codificados em Base64.

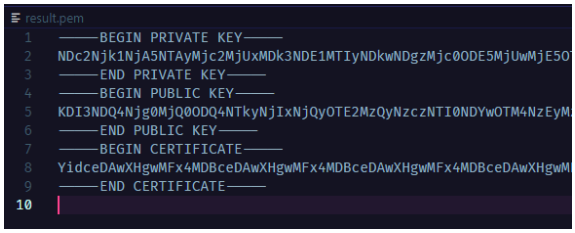

Figura 4. Exemplo de arquivo gerado na formatação

3. Conclusão

Foi possível realizar a implementação de todos os requisitos propostos no trabalho. A experiência se tornou mais desafiadora pela não permissão do uso de determinadas bibliotecas públicas que faciltariam a formulação da solução. Mas nos levou a de fato destrinchar o funcionamento esperado.

Dessa forma, o projeto demonstrou um impacto direto no entendimento da dupla a respeito dos conceitos abordados.

4. Bibliografia

1. ”RSA (cryptosystem)”, `https://en.wikipedia.org/wiki/RSA_(cryptosystem)` . Acesso em maio de 2021 .
2. ”Encoding and Decoding Base64 Strings in Python”, `https://stackabuse.com/encoding-and-decoding-base64-strings-in-python/` . Acesso em maio de 2021
3. ”Optimal asymmetric encryption padding”, `https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding` . Acesso em maio de 2021 .