CIC0201 - Segurança Computacional - 2020/2
Relatório de Implementação
Gerador/Verificador de Assinaturas

Helio Adson Oliveira Bernardo 16/0030081

Maio 2021

1 Objetivo

Este trabalho tem como objetivo implementar um gerador e verificador de assinaturas RSA em arquivos, que é um sistema criptográfico amplamente utilizado para transmissão segura de dados.

2 Introdução

Dado que a criptografia RSA é um algoritmo de criptografia assimétrico. Nesse caso, qualquer pessoa pode criptografar um dado, mas só é possível para alguém com a chave correta descriptografá-lo. O RSA funciona gerando uma chave pública e uma chave privada, as chaves pública e privada são geradas juntas e formam um par de chaves. Uma boa prática para deixar o RSA mais segura seria utilizar a função de hash SHA-3-512. O SHA-3 (Secure Hash Algorithm) é um algoritmo de hash seguro que utiliza uma construção de esponja, isto é, tem uma fase para "absorver"(processamento) a entrada e uma fase para "espremer"(gerar) a saída. Esse tipo de construção permite utilizar entradas de tamanhos arbitrários para gerar saídas de tamanhos arbitrários.

3 Implementação

Para realizar a implementação do trabalho, foi utilizada a linguagem Python e a biblioteca rsa. Métodos foram modularizados para desempenhar cada passo e funcionalidade do programa, sendo escolhida pelo usuário através de uma interface no terminal:

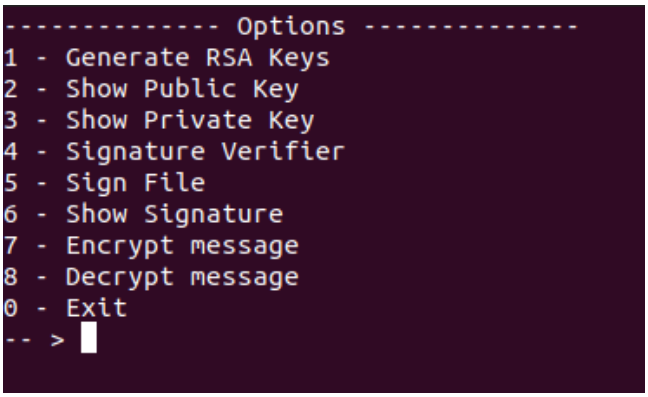

Figura 1: Interface de Usuário

Como descrito na Figura 1, o usuário pode gerar as chaves, mostrá-las, verificar assinatura de um arquivo e assiná-lo.Sendo assim, 4 arquivos txt são usados no trabalho para sua funcionalidade geral, sendo eles: *private_key.txt*, *public_key.txt*, *file.txt* e *signature.txt*, *encrypt.txt* e *decrypt.txt*. Sendo os arquivos de chave onde elas são gravadas/lidas, *signature* onde a assinatura fica, *file* sendo o arquivo a ser testado e por último os arquivos *encrypt.txt e decrypt.txt* para crifrar e decifrar mensagens de um arquivo passado ao programa.

4 Principais métodos desenvolvidos

Os principais métodos do trabalho serão descritos a seguir, sendo que os demais são simples (como saída de dados) ou apenas lógica da interface do usuário.

4.1 Criação de chaves públicas e privadas

```
def generate_rsa_key(self):
  print("Generating RSA Keys...")
  (public_key, private_key) =  rsa.newkeys(2048)

  try:
    with open("public_key.txt", "wb") as outfile:
      outfile.write(pub_pem.save_pkcs1('PEM'))
  except IOError:
    print('Error! Try again')

  try:
    with open("private_key.txt", "wb") as outfile2:
      outfile2.write(prv_pem.save_pkcs1('PEM'))
    print("Created Rsa Keys!")
  except IOError:
    print('Error! Try again')
```

Figura 2: Generate RSA Key

Como descrito na Figura 2, este método utiliza a biblioteca rsa para criar as duas chaves (pública e privada) e gravá-las nos arquivos.

4.2 Assinar arquivo

```
def sign(self, file):
    private_key_load = rsa.PrivateKey.load_pkcs1(self.__read_file('private_key.txt'))

    file = self.__read_file(file)

    sign_hash = rsa.compute_hash(file, 'SHA3-256')

    signature = rsa.sign(file, private_key_load, 'SHA3-256')

    try:
        with open("signature.txt", "wb") as outfile2:
            outfile2.write(signature)
        print("File has been signed!")
    except IOError:
        print('Error! Try again')
```

Figura 3: Sign File

Como descrito na Figura 3, este método assina o arquivo passado pelo usuário, usando o algoritmo de hash SHA-3 e a chave privada.

4.3 Verificar assinatura

```
def verify_rsa_signature(self, file):
    public_key_load = rsa.PublicKey.load_pkcs1(self.__read_file('public_key.txt'))

    file = self.__read_file(file)

    signature = self.__read_file('signature.txt')

    try:
        rsa.verify(file, signature, public_key_load)
        print('\n Valid signature! Document has not been modified.\n')
    except:
        print('\n Document has been modified or does not have a signature.\n')
```

Figura 4: Verify Signature

Como descrito na Figura 4, este método verifica a assiantura do arquivo passado pelo usuário, retornando se ele está válido ou não.

4.4 Encryptar Mensagem

```
def encrypt(self, file):
    file = self.__read_file(file)
    message = file
    public_key = rsa.PublicKey.load_pkcs1(self.__read_file('public_key.txt'))

    crypto = rsa.encrypt(message, public_key)
    try:
        with open("encrypted.txt", "wb") as outfile3:
            outfile3.write(crypto)
        print("Encrypted message!")
    except IOError:
        print('Error! Try again')
```

Figura 5: Encrypt

Como descrito na Figura 5, este método encrypta uma mensagem de um arquivo passado pelo usuário

4.5 Decryptar Mensagem

```
def decrypt(self):
    file = self.__read_file("encrypted.txt")
    private_key = rsa.PrivateKey.load_pkcs1(self.__read_file('private_key.txt'))

    message = rsa.decrypt(file, private_key)
    try:
        with open("decrypt.txt", "wb") as outfile3:
            outfile3.write(message)
        print("Decrypted message!")
    except IOError:
        print('Error! Try again')
```

Figura 6: Decrypt

Como descrito na Figura 5, este método decrypta a mensagem presente no arquivo *encrypt.txt*

5 Conclusão

No trabalho, foi possível gerar chaves RSA públicas e privadas e como usá-las assinar e verificar um arquivo. Com a limitação de utilizar a biblioteca rsa, foi possível obter o comportamento esperado no projeto.