Universidade de Brasília - UnB
Departamento de Ciência da Computação
Disciplina: Segurança Computacional - CIC0201
Professor: Dr. João José Costa Gondim
Período: 2023/2 - Turma: 2

Alunos: Eder de Amaral Amorim - Matrícula: 170140636
Tais Alves Oliveira - Matrícula: 190117176

Trabalho de Implementação 3: Gerador/Verificador de Assinaturas

Descritivo

O presente trabalho abordou o desenvolvimento de um gerador/verificador de assinaturas RSA em arquivos. O trabalho foi feito usando a linguagem Python 3 e consiste em três partes, como segue abaixo:

Parte I: Geração de chaves e cifra

- Módulo: **main.py**

Ponto de entrada do programa. Ele coordena a execução, chamando funções de cada módulo conforme necessário. Nele o usuário pode adicionar um caminho para um arquivo a ser cifrado, tomando o cuidado para não ser um arquivo grande. E executa o processo de leitura, transformação em base64, geração de chaves, criptografia e descriptografia e mostra o conteúdo do documento após todo processo.

- Módulo: **RSA.py**
  - ***inverso_multiplicativo( )***

Faz o cálculo do inverso multiplicativo modular de um número. A função trabalha com o máximo divisor comum (mdc) por meio da chamada da função

mdc_extendido( ). Caso o mdc seja diferente de 1, então não existe inverso multiplicativo e caso o mdc seja igual a 1, então possui inverso multiplicativo.

- ***generate_keys( )***

Função responsável pela geração das chaves públicas e privadas. A chave é definida com tamanho 1024 bits para então gerar dois números primos (‘*p’* e ‘*q’*) nesse tamanho, por meio da função *gerar_primo( ).*

Após isso, a função faz o produto de ‘*p’* e ‘*q’* e calcula o valor de *phi* por meio da função *valor_phi( ).* Posteriormente a isso, faz-se a seleção de um expoente ‘*e’*, por meio da função *expoente_publico( )* e logo após realiza o cálculo do inverso multiplicativo *‘d’* com a função *inverso_multiplicativo( ).*

As chaves são criadas como sendo: chaves públicas *‘n’* e *‘e’,* chaves privadas *‘n’* e *‘d’.* Finalmente faz-se a impressão das chaves públicas e privadas e o retorno das mesmas ao fim da função.

- ***cifrar_RSA( )***

Função responsável por fazer a operação de cifragem usando o algoritmo RSA. Recebe dois parâmetros: *‘m’* (mensagem a ser cifrada) e ‘chave’ (chave pública *‘n’, ‘e’*). A função realiza então a operação modular exponencial *pow( )* para realizar a cifragem.

- ***decifrar_RSA( )***

Faz a decifragem usando o algoritmo RSA. Recebe dois parâmetros: ‘c’ (mensagem cifrada que será decifrada) e ‘chave’ (chave pública ‘n’, ‘d’). A função realiza então a operação modular exponencial pow( ) para realizar a decifragem. Por fim a função retorna a mensagem original.

- Módulo: **OAEP.py**

Este arquivo contém a implementação do preenchimento da mensagem, assim como a remoção.

- ***HexastringToCharacters( )***

função que transforma um texto hexadecimal em caracteres. Utilizado para facilitar a expansão no momento da máscara. Escolhemos o padrão ‘latin-1’ por haver mais quantidade de caracteres.

- ***MGF( )***

A função MGF expande a semente em um valor maior, em cada iteração ela anexa um contador de 4 bytes (com o ultimo byte incrementado de cada iteração) e calcula um novo hash que é concatenado ao final do valor acumulado, usando um deslocamento á esquerda seguido por um OR bit a bit. Ao final aplica-se um deslocamento a direita no resultado, para que o valor seja divisível por 8, representando 1 bit.

- ***aplicar_oaep( )***

Esta função aplica o OAEP, através da mensagem a ser criptografada e de um par de chaves públicas, além de uma hash aleatória estática.

Ela primeiro calcula o tamanho do preenchimento necessário, e verifica se o tamanho da mensagem é muito longo, se não for, ela adiciona a hash ao início da função assim como a quantidade de “00” necessários.

Depois ela faz um um XOR entre o pad resultante da operação anterior e o a hash criada mascarada, e depois aplica o mascaramento de novo e faz outro XOR com a hash resultando na cifra em questão.

- ***remover_oaep( )***

Essa função remove o preenchimento da mensagem entregando ela decifrada utilizando a chave privada, para tal. Seu funcionamento é o inverso da função anterior. Foram mantidos as mesmas variáveis para melhor entendimento.

Parte II: Assinatura

- Módulo: **assinatura.py**

Este arquivo contém a implementação da assinatura.

1. Cálculo de hashes da mensagem em claro (função de hash SHA-3)

- ***calcular_hash_sha3( )***

Faz uso da biblioteca hashlib para calcular o hash SHA-3-256 da mensagem de entrada e retorna seu valor hash, que é posteriormente utilizado no processo de assinatura.

2. Cálculo de hashes da mensagem em claro utilizando a função de hash SHA-3:
   - ***assinar_mensagem( )***

Função para assinatura da mensagem. Faz a cifração do hash da mensagem, calcula o hash SHA-3-256, cifra hash usando a chave privada RSA (*'d'*) e retorna a assinatura formatada em base64.

3. Formatação do resultado (caracteres especiais e informações para verificação em BASE64)
   - ***assinar_mensagem( )***

A função retorna o resultado (hash decifrado) formatado em BASE64 por meio do uso da biblioteca base64.

Parte III: Verificação

- Módulo: **assinatura.py**

1. Decifração da assinatura (decifração do hash)
   - ***decifrar_assinatura( )***

A partir da assinatura codificada, a função realiza a decifração por meio da decodificação da base64 e utilizando a chave pública 'n', 'e'. O resultado é então convertido em um número inteiro e usado no algoritmo RSA para sua decifração.

2. Verificação (cálculo e comparação do hash do arquivo)
   - ***verificar_assinatura( )***

Faz a verificação da assinatura comparando o hash da mensagem original, que foi calculado pela função *assinar_mensagem( )* com o hash decifrado. Caso os hashes sejam iguais, então a assinatura é válida, caso contrário a assinatura é inválida.