Relatório de Trabalho de Implementação

Segurança Computacional - Turma A

Hanniel Fernando Lopes Saldanha – 180111515

José Fortes Neto - 160128331

1 - Introdução

Amplamente utilizado para transmissões de dados de forma segura, o RSA foi um dos primeiros sistemas de criptografia contendo chave pública. Utilizando um sistema assimétrico com duas chaves, ele demonstrou ser um sistema seguro devido a seu sistema de fatoração dos números primos.

Este trabalho implementará uma versão do algoritmo para fins acadêmicos utilizando o algoritmo RSA disponível.

2 - Objetivo

Este trabalho tem como objetivo implementar um programa que deverá ser capaz de gerar e verificar assinaturas RSA. Deverá, assim, possuir uma geração de chaves com tamanho de 1024 bits, além de assinar mensagens e em seguida ser capaz de fazer a validação dela utilizando até mesmo cálculo de hashs no processo.

3 - Criptografia assimétrica

A criptografia assimétrica, utilizada no RSA, também conhecida como criptografia de chave pública, é baseada em 2 tipos de chaves de segurança — uma privada e a outra pública. Elas são usadas para cifrar mensagens e verificar a identidade de um usuário.

Resumidamente falando, a chave privada é usada para decifrar mensagens, enquanto a pública é utilizada para cifrar um conteúdo. Assim, qualquer pessoa que precisar enviar um conteúdo para alguém precisa apenas da chave pública do seu destinatário, que usa a chave privada para decifrar a mensagem.

Esse sistema simples garante a privacidade dos usuários e aumenta a confiabilidade de uma troca de dados.

4 - Implementação

A linguagem Python foi a escolhida para a implementação do trabalho, pela sua facilidade e praticidade com uso de bibliotecas conhecidas.

O ponto inicial do projeto é a função *“Main()”* que é responsável por chamar os métodos que irão gerar as chaves, assinar a mensagem e exibir os resultados na tela do usuário.

Temos os seguintes métodos:

4.1 - generate_RSA_keys():

É responsável por gerar as chaves pública e privada para a execução do algoritmo.

Esse método consiste em gerar dois números primos (p e q) com base no tamanho da chave esperada. Em seguida ela faz o cálculo do modulo RSA n, seguindo a definição do algoritmo onde n = p * q, e seguida utiliza a função totiente de Euller para encontrar um Phi(n), nomeado como t, onde t = (p-1) * (q-1).

O número aleatório **e** é calculado satisfazendo as condições de ser menor que t e maior que 1. E, por fim, é gerado o **d** para a chave privada, que consiste em usar a função "*modinv(e, t)*" para que possa calcular a chave resultante.

```
def generate_RSA_keys():
    #Numero primo 'p' de n bits
    p = get_prime(config['BITS']//2)

    #Numero primo 'q' de n bits diferente de 'p'
    while True:
        q = get_prime(config['BITS']//2)
        if q != p:
            break

    #RSA Modulus 'n'
    n = p * q

    t = ( p - 1 ) * ( q - 1 )

    for e in range(config['e'],t):
        if gcd(e,t)==1:
            break

    d = modinv(e, t)

    public_key = {'n' : n, 'e': e}
    private_key = {'n' : n, 'd': d}

    #print("Public Key: {}".format(public_key))
    #print("Private Key: {}\n".format(private_key))

    return (public_key, private_key)
```

4.2 - sign_message(mensagem, private_key):

Este método é responsável por assinar a mensagem fornecida utilizando a chave obtida. Ele converte a mensagem em base64 e em seguida a transforma em um hash de 512 bits que será assinado junto a sua chave.

```
def sign_message(mensagem, private_key):
    #Mensagem/informacao para ser assinada
    mensagem = b64encode(mensagem.encode())

    #Hash da mensagem de 512bits para caber na assinatura de 1024bits
    hash = int.from_bytes(sha3_512(mensagem).digest(), byteorder='big')

    #Assinatura com a chave privada
    assinatura = pow(hash, private_key['d'], private_key['n'])

    return (hash, assinatura)
```

4.3 - verify_signature (hash, assinatura, public_key)

Método responsável por obter o hash da mensagem original e fazer a verificação para validar a mensagem.

```
def verify_signature(hash, assinatura, public_key):
    #Retorno da assinatura para o hash da mensagem original para posterior comparacao e validacao da assinatura
    hash_assinatura = pow(assinatura, public_key['e'], public_key['n'])

    if hash == hash_assinatura:
        return True
    return False
```

4.5 – Ainda temos os seguintes métodos auxiliares implementados parar ajudar na modularizam do código.

- miller_rabin: Executa o teste de “primalidade” de Miller-Rabin.
- is_prime: Verifica se um determinado número é primo.
- get_prime: Método responsável por gerar um número primo aleatório.
- gcd: verificação *gcd* (e,t) = 1.
- extend_euclid: função que calcula máximo divisores comuns e fornece seus coeficientes.
- modinv: Função utilizada para calcular o inverso multiplicativo de ***e*** em (mod φ(n)).

5 - Resultados

Como apresentado na figura abaixo, os resultados constam a mensagem a ser criptografada, o hash da mensagem, a mensagem assinada e por último a validação da mensagem. É importante destacar que devido ao tamanho expressivo da chave (1024 bits) foi necessário, nos testes, reduzir o tamanho do valor de ***e*** para que o algoritmo pudesse ser executado em tempo hábil.

```
Mensagem para assinar: Mensagem secreta 1, 2, 3 !!!

Hash da mensagem: 0xa8c6fc24420af32913cbb3449bcaf135c1537660d921235be50a805274328484e79bd81915dfed472a9f31b23482869cfb6cae602fe591403105d89fdf0a593a

Assinatura: 0x2628456b77738d83afafa6dde1a90165e85405aa3dc7d8ccd2411e941c83acb68c669876d16912e8d0137614edcf4af0ff4e773ba523cfe6b0de48c3100ef0d752520a0a

Validade da assinatura: True
```

6 - Conclusão

Este trabalho apresentou uma implementação do método RSA, sendo de enorme aproveitamento e estimulando o aprendizado de forma prática. O trabalho implementou desde a assinatura a validação das mensagens assinadas cumprindo seus objetivos.