Gerador e verificador de assinaturas RSA em arquivos

Camila Frealdo Fraga - 17/0007561
José Roberto Interaminense Soares - 19/0130008

Dep. de Ciência da Computação - Universidade de Brasília (UnB)

1. Introdução

O algoritmo RSA (Rivest–Shamir–Adleman) é um sistema de criptografia amplamente utilizado na transmissão de dados, destacando-se especialmente em operações de assinatura digital e verificação. Este relatório se propõe a explorar a implementação de um Gerador e Verificador de Assinaturas RSA em arquivos, incorporando também o esquema de padding OAEP (Optimal Asymmetric Encryption Padding) para aprimorar a segurança do processo.

A assinatura digital desempenha um papel crucial na verificação da autenticidade de dados eletrônicos, assegurando que a origem e integridade da informação sejam preservadas durante a transmissão ou armazenamento. O algoritmo RSA, baseado em chaves assimétricas, oferece uma solução eficaz para esse propósito.

Além disso, a inclusão do esquema de padding OAEP otimiza a segurança da implementação, tornando-a resistente a determinados ataques criptográficos. O OAEP não apenas adiciona uma camada adicional de aleatoriedade à mensagem antes de criptografá-la, mas também garante a integridade do processo de criptografia, minimizando as vulnerabilidades associadas a ataques do tipo "chosen ciphertext.", entre outros tipos.

2. Metodologia

2.1 Geração de Chaves

Para a geração de chaves, foi feita uma função que recebe o número de bits desejados e gera dois primos **p** e **q** aleatórios. O número de bits foi definido no início do programa como 1024, que é um valor suficientemente grande mas que não compromete a eficiência do programa. A função de geração de chaves foi feita baseada na referência [5] e pode ser vista a seguir:

```
number_of_bits = 1024
def key_generation():
    p = generate_probable_prime(number_of_bits)
    q = generate_probable_prime(number_of_bits)
    n = p*q
    phi = (p-1)*(q-1)
    e = calculate_e(phi)
    d = calculate_d(e, phi)
    return [n, e, d]
def generate_probable_prime(number_of_bits):
    while True:
        n = random_odd_value(number_of_bits)
```

```
        if (miller_rabin(n, 40)):
            return n
```

O teste de Miller Rabin recebe um número e uma quantidade de rodadas, e 40 é o número ideal, onde a probabilidade de erro é negligenciável. Portanto, com os parâmetros '**n**', '**e**' e '**d**', temos a chave pública e privada. A primeira operação do programa, caso o usuário deseje, é gerar uma chave nova, em Keys.txt.

2.2. Rsa - OAEP

O próximo passo foi fazer o RSA-OAEP. Para isto, foi consultado a referência [4]. A função que calcula o hash utiliza a biblioteca hashlib do Python. O número aleatório que garante confiabilidade é gerado com a biblioteca random. MGF representa a função máscara.

```
def rsa_oaep(n, e, M, L, k):
    h_len = hashlib.sha1().digest_size                #hash
    lHash = hashlib.sha1(L).digest()
    PS = b'\x00' * (k - len(M) - 2*h_len - 2)
    DB = lHash + PS + b'\x01' + M               # DB = lHash || PS || 0x01 || M

    seed = os.urandom(h_len)
    db_mask = MGF(seed, k - h_len - 1)
    masked_db = bytes(a ^ b for a, b in zip(DB, db_mask))
    seed_mask = MGF(masked_db, h_len)
    masked_seed = bytes(a ^ b for a, b in zip(seed, seed_mask))
    EM = b'\x00' + masked_seed + masked_db
    return EM
```

Em resumo, a saída é a concatenação de 3 elementos:
1- Byte de verificação (00)
2- 'Xor' entre número aleatório (seed) e MGF(hash do rótulo + padding de zeros + byte de verificação interno + mensagem).
3- 'Xor' entre MGF(valor 2) e seed.

Tal função é utilizada na parte de codificar, onde a mensagem é quebrada em blocos (caso seja muito grande) e cada bloco gera um conjunto de bytes, que é escrito no arquivo cypher.txt, caso o usuário deseje cifrar um arquivo. A parte de decifração é reversa e é semelhante à cifração.

2.3. Assinatura e Verificação

A assinatura e verificação são opções que o usuário pode escolher. O programa recebe um arquivo de texto, e escreve no final a assinatura, no formato:
Assinado: Assinatura.

Para isto, é necessário obter as chaves de codificação, encontradas no arquivo Keys.txt. Se a aplicação fosse feita para uma empresa onde clientes teriam acesso, o usuário não teria permissão para acessar tal arquivo. Como o programa foi pensado para administradores do sistema, o arquivo é acessível. A parte do programa responsável por isto pode ser vista a seguir:

```
[public_key, private_key] = readKeyFile('chaves.txt')
print ("Gerando assinatura...")
hash_file = hashlib.sha3_256(file_read.encode('utf-8')).hexdigest()
C = encode(hash_file, public_key[0], public_key[1])
C = base64.b64encode(C)
assinatura = '\nAssinado: \n' + C.decode() # Escreve a assinatura no arquivo
file = open(file_name, 'a')
file.write(assinatura)
file.close()
print ("Assinatura gerada com sucesso!")
```

Esta parte é bastante simples, primeiro é gerado o hash do arquivo, que então é cifrado, e em seguida é codificado em base64, para ser escrito no artigo. E a parte de verificação de assinatura é a seguinte:

```
[public_key, private_key] = readKeyFile('chaves.txt')
print ("Verificando assinatura...")
assinatura = file_read.split('\n')[-1].split(': ')[-1] # Pega a assinatura
assinatura = base64.b64decode(assinatura)
# Decodifica a mensagem usando RSA-OAEP
try: M = decode(public_key[0], private_key, assinatura, b'')
    except:
        print ("Assinatura inválida!")
        continue
file_read = file_read.split('\n')[:-2] # retira últimas duas linhas do arquivo
hash_file = hashlib.sha3_256('\n'.join(file_read).encode('utf-8')).hexdigest()

if M.decode('utf-8') == hash_file:
    print ("Assinatura válida!")
else:
    print ("Assinatura inválida!")
```

Para verificar a assinatura, a última linha é decodificada (de base64 para bytes), e é utilizada a função de decode na assinatura. Após isso, é calculado o hash do arquivo (sem as linhas de assinatura) e é comparado então com a decodificação. Se o resultado for o mesmo então aquela assinatura foi gerada pelo código com as chaves públicas, caso contrário, a assinatura é inválida.

3. Conclusão

Em síntese, a implementação do Gerador e Verificador de Assinaturas RSA em arquivos, integrando o esquema de padding OAEP, representa uma solução abrangente e segura para a autenticação de dados eletrônicos. O uso do algoritmo RSA, com sua estrutura de chave pública e privada, aliado ao OAEP, fortalece a segurança do processo ao introduzir camadas adicionais de aleatoriedade e integridade.

A maior dificuldade encontrada foi em relação ao OAEP, que não tem uma documentação clara e concisa. Apesar disso, o grupo conseguiu superar as dificuldades e através da pesquisa conseguiu desenvolver o projeto. O trabalho

poderia ser incrementado caso a assinatura fosse realizada em outros tipo de documento, e as chaves fossem estáticas, garantido mais facilidade na hora de verificação, porém não era o escopo do projeto. Portanto, o entendimento sobre criptografia foi alcançado e o trabalho foi devidamente concluído.

Referências

1. [Wikipedia - RSA]
2. [Vídeo - OAEP]
3. [Wikipedia - Função MGF]
4. [Guia - OAEP]
5. [Vídeo - Gerando Números Primos]