Gerador/Verificador de Assinaturas

Universidade de Brasília - Segurança Computacional

Lucas Ramson Siefert, 222011543

July 2023

1 Introdução

Ao longo deste trabalho foi realizada a implementação de um gerador e verificador de assinaturas RSA em arquivos. O objetivo foi desenvolver um programa com funcionalidades que abrangessem a cifração e decifração AES, a geração de chaves e cifração RSA, bem como a assinatura RSA e sua verificação. Dividido em quatro partes, o trabalho abordou desde a geração de chaves com teste de primalidade até a formatação dos resultados e comparação dos hashes dos arquivos. O relatório apresenta os detalhes da implementação, os desafios enfrentados e as conclusões alcançadas. A implementação destas funcionalidades foi realizada na linguagem de programação Python.

2 Visão geral da implementação

Para melhor organização da implementação e afim de atingir todos os casos de uso solicitados, o código foi subdivido em três principais módulos de funcionamento: *AES.py*, *RSA.py* e *keyGen.py*.

Conforme suas nomeclaturas indicam, cada módulo é responsável por uma parte do funcionamento dos casos de uso, sendo o módulo *AES.py* responsável pela implementação da criptografia AES, bem como o modo de operação CTR, o módulo *RSA.py* responsável pela criptografia em RSA, *padding* com OAEP e pelas funções de geração e verificação de assinaturas e, por fim, o módulo *keyGen.py* é responsável pela geração de chaves para o uso dos outros módulos, contendo a geração de uma chave aleatória de 128 bits para o AES e de pares de números primos de 1024 bits para chaves do RSA.

2.1 AES

A criptografia AES foi implementada em essencialmente quatro etapas, operando em 16 bytes em forma de uma matrix 4x4: expansão da chave, adição inicial da chave da rodada (*AddRoundKey*), 9 rodadas de quatro operações (*SubBytes*, *ShiftRows*, *MixColumns* e *AddRoundKey*), e uma última rodada, que executa

apenas as três primeiras operações. A função *aes_encryption*() recebe a mensagem a ser criptografada e a chave para a criptografia em bytes, e retorna a mensagem cifrada também em bytes.

A expansão da chave consiste essencialmente na derivação das "*round keys*", chaves de cada rodada, a partir da chave original. As outras quatro operações podem ser vistas, de forma simples, como:

1. *SubBytes*: substituição não-linear de bytes de acordo com uma tabela de pesquisa;
2. *ShiftRows*: transposição das últimas três linhas de forma cíclica;
3. *MixColumns*: "mistura" das colunas, combinando os bytes de cada coluna;
4. *AddRoundKey*: operação XOR dos bytes com a chave da rodada

Para que a criptografia funcionasse em mensagens com mais de 16 bytes, foi implementada o funcionamento em CTR, que quebra a mensagem em blocos de 16 bytes e realiza uma operação XOR com um contador criptografado com a chave providenciada. O contador é incrementado para cada bloco de 16 bytes, sendo inicializado como uma matriz 4x4 de 16 bytes 0x00.

2.2 RSA

A cifra RSA é um algoritmo de criptografia assimétrica que utiliza dois tipos de chave: pública e privada. Primeiro, o destinatário do mensagem gera um par de chaves: uma chave pública para cifrar mensagens e uma chave privada para decifrá-las, sendo que cada uma dessas é uma dupla de valores, consistindo em um número primo e um módulo. Para cifrar uma mensagem, o remetente utiliza a chave pública do destinatário para elevar a mensagem a uma potência específica e, em seguida, calcula o módulo dessa potência. Para decifrar a mensagem, o destinatário utiliza sua chave privada para elevar a mensagem cifrada à potência apropriada e, novamente, calcula o módulo dessa potência.

Na implementação em python, isso é realizado de forma simples, com a função *apply_rsa*() recebendo uma mensagem e uma chave (um par de valores) e utilizando-os tanto para cifrar quanto para decifrar a mensagem, visto que a operação realizada é a mesma.

2.2.1 OAEP

Para melhorar a segurança ainda mais, é utilizado o esquema de preenchimento (ou *padding*) conhecido como OAEP, ou Optimal Asymmetric Encryption Padding. Antes da cifração, a mensagem é expandida para um tamanho compatível com o comprimento da chave RSA. Em seguida, uma função de hash é aplicada à mensagem expandida, resultando em um valor de hash. Esse valor é combinado com um valor aleatório chamado de "*pad*" por meio de uma operação XOR. Em seguida, outra função de hash é aplicada ao resultado do

XOR, gerando um novo valor de hash. Esse novo valor é então combinado com o "pad" original e a mensagem expandida por meio de outra operação de XOR.

Na implementação, as funções *oaep_encrypt*() e *oaep_decrypt*() são responsáveis, respectivamente, por cifrar e decifrar as mensagens através deste padrão.

2.2.2 Geração/Verificação de assinaturas

No mesmo arquivo, são também implementadas as funções de verificação e geração de assinaturas em RSA. Por especificação, a assinatura é também codificada em base64. De forma simples, o funcionamento da geração de assinaturas se dá pela aplicação de uma função de *hash* em uma mensagem e a cifração em RSA desta mensagem pelo remetente com sua chave privada, gerando a assinatura. Essa assinatura é enviada junto com a mensagem, e pode ser decifrada utilizando a chave pública do remetente, resultando no mesmo *hash* calculado anteriormente. Caso o *hash* resultante seja igual ao *hash* da mensagem recebida, a assinatura é validada.

Na implementação, a assinatura é gerada aplicando a função de hash $sha3\_256$, da biblioteca pública *hashlib* ao conteúdo da mensagem, seguida pela criptografia do valor de hash usando uma chave privada RSA. A assinatura é então convertida para BASE64 e retornada. A verificação é realizada calculando o valor de hash da mensagem recebida e decifrando a assinatura usando a chave pública correspondente. Se os valores de hash coincidirem, a assinatura é considerada válida. As funções $rsa\_sign()$ e $rsa\_verify()$ são responsáveis por esse processo.

3 keyGen

Por fim, o módulo de keyGen pode ser considerado o mais simples dentre os três. Este módulo é responsável tanto por gerar os pares de chaves públicas e privadas para o funcionamento da cifra RSA quanto a geração de chaves aleatórias de 128 bits para o funcionamento da cifra AES.

A função $aes\_keygen()$ é extremamente simples, retornando um text aleatório de 16 bytes em hexadecimal através da função $token\_hex()$ da biblioteca *secrets*.

A função $rsa\_keygen()$ é responsável por retornar dois pares de inteiros: uma chave pública e uma chave privada. Isso é feito através da chamada da função $generate\_prime\_number()$ para a geração dos valores de p e q, que são utilizados para calcular os valores de $n = p \times q$ e de $\phi = (p-1) \times (q-1)$. Em seguida são calculados os valores de e e d, sendo e um coprimo de ϕ, ou seja, o maior divisor comum entre estes é 1, e sendo d um inverso multiplicativo modular de $e\ mod(\phi)$, ou seja, $d \equiv e^{-1}(mod\phi(n))$. Estes valores são então retornados em dois pares: a chave pública contendo n e e, e a chave privada contendo n e d.

A função de geração de primos $generate\_prime\_number()$ funciona através da geração de números aleatórios de 1024 bits e sua posterior testagem através do método de Miller-Rabin para averiguar sua primalidade, pela função $is\_prime()$. O número gerado aleatoriamente tem inicialmente seus bits mais e menos sig-

nificantes setados como 1, para garantir que seja um número ímpar e que tenha, realmente 1024 bits significantes, e, então, sua testagem é feita.

A testagem é realizada na função *is_prime*(), que recebe o número a ser testado, e retorna um valor booleano expressando a primalidade do número. Na implementação, recebe-se um número "n" e escolhe-se um número aleatório "a" entre 2 e $(n-2)$. Em seguida, é realizada a seguinte operação modular: $a^d \equiv 1(modn)$, onde "d" é o maior divisor ímpar de $(n-1)$. Se a congruência não for satisfeita, significa que "n" é composto e, caso contrário, o algoritmo passa para a próxima etapa. Neste momento, são realizadas iterações da checagem $a^{(2^r * d)} \equiv -1(modn)$, para "r" variando de 0 até $s-1$, onde "s" é o maior inteiro tal que $(n-1)$ seja divisível por 2^s. Se algum desses testes falhar, ou seja, a congruência não for satisfeita, então o número "n" é composto. Caso contrário, ele é considerado como provavelmente primo com uma alta probabilidade, mas não com certeza.

References

[Nata] National Institute of Standards and Technology (NIST). *Advanced Encryption Standard (AES)*. `https://www.nist.gov/publications/advanced-encryption-standard-aes`. Accessed: 6/22/2023.

[Natb] National Institute of Standards and Technology (NIST). *Proceedings of the 23rd NISSC 2000 - Paper 905*. `https://csrc.nist.gov/csrc/media/publications/conference-paper/2000/10/19/proceedings-of-the-23rd-nissc-2000/documents/papers/905slide.pdf`. Accessed: 6/29/2023.

[Pon] Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS). *RSA-OAEP Specification*. `https://www.inf.pucrs.br/~calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf`. Accessed: 6/30/2023.