Departamento de Ciência da Computação
Universidade de Brasília

Gerador/Verificador de Assinaturas

Nome: Gabriel Mendes Ciriatico Guimarães
Matrícula: 202033202

Teste de primalidade de Miller-Rabin

O teste de primalidade de Miller-Rabin permite saber se um número é possivelmente primo. O teste foi implementado seguindo o algoritmo[1] para o número n, testando k vezes (k padrão de 40):

- Escrever n-1 como (2^s)*d com d ímpar fatorando potências de 2 a partir de n-1;
- LOOP: repetir k vezes:
    - Escolher a aleatório entre 2 e n-2;
    - x = (a^d)*mod n
    - se (x == 1) ou (x == n-1):
        - Fazer LOOP
    - Para r entre 0 e s-1:
        - x = (x^2)*mod n;
        - se x == n-1:
            - Fazer LOOP
    - Retorna Falso
- Retorna Verdadeiro

Cifração e decifração RSA

Para o sistema Rivest-Shamir-Adleman (RSA), é necessário Inicialmente dois números primos p e q.

Para obter as chaves públicas e privadas, é necessário fazer:

1. Obter o valor n = p*q;
2. Obter o valor φ(n)= (p-1)*(q-1);
3. Obter um valor e tal que:
    a. 1 < e < φ(n);
    b. e é coprimo de n;
    c. e é coprimo de φ(n).
4. Obter d, em que d = (e^−1)*mod φ(n);
5. Retornar a chave pública (e, n);
6. Retornar a chave privada (d, n).

Com a chave pública, para cifrar basta fazer:
m^e * mod n, onde m é a sequência de bytes da mensagem original transformada em número.

Já para decifrar, com a chave privada basta fazer:
c^d * mod n, onde c é a sequência cifrada.

***Optimal Asymmetric Encryption Padding* (OAEP)**

O processo de *padding* (ou preenchimento, em português) é necessário no sistema RSA para evitar o determinismo do sistema, que pode torná-lo vulnerável. O *padding* é randômico, garantindo que uma mesma mensagem criptografada várias vezes tem sempre

[1] Algoritmo implementado (com pequenas modificações) a partir de:
https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Primality_Testing

uma saída diferente. Isso protege o sistema de *chosen-plaintext attacks*, onde um oponente pode obter textos cifrados para qualquer texto que quiser.

RSA sem *padding* é usado apenas com fins didáticos, já que dessa forma o sistema fica bastante vulnerável. Um *padding* frequentemente usado junto com RSA é o *Optimal Asymmetric Encryption Padding* (OAEP).

Para a implementação[2] do OAEP, seguiu-se o algoritmo definido pela RFC 8017. As siglas usadas significam:

- **MGF**: é a função de mascaramento (*mask generating function*, em inglês). Aqui, usa-se a MGF1;
- **Hash:** função de hash escolhida (aqui, para o *padding*, sha1);,
- **hLen:** tamanho em bytes da saída da função hash;
- **k:** tamanho do módulo n do RSA em bytes;
- **M:** mensagem para ser preenchida (*padded*), de no máximo $k - 2 \cdot hLen - 2$ bytes;
- **L:** rótulo opcional a ser associado à mensagem (aqui, não foi usado nenhum, portanto fica rótulo vazio);
- **PS:** é uma string de bytes de $k - mLen - 2 \cdot hLen - 2$ bytes nulos;
- $\oplus$**:** é a operação XOR.

Para cifrar, foi implementado em Python o seguinte algoritmo:

1. Passar o rótulo L na função de hash escolhida, obtendo LHash: lHash = Hash(L);
2. Gerar a string PS com $k - mLen - 2 \cdot hLen - 2$ bytes com valor 0x00;
3. Concatenar lHash, PS, o byte 0x01 e a mensagem M para obter o bloco de dados DB: DB = lHash || PS || 0x01 || M;
4. Gerar uma semente aleatória (*random seed*, em inglês) de tamanho hLen;
5. Usar a função de mascaramento para gerar uma máscara do tamanho certo para o bloco de dados DB: dbMask = MGF (seed, $k - hLen - 1$);
6. Mascarar o bloco de dados com a máscara gerada: maskedDB = DB $\oplus$ dbMask;
7. Usar a função de mascaramento para gerar uma máscara de tamanho hLen para a semente: seedMask = MGF (maskedDB, hLen);
8. Mascarar a semente com a máscara gerada: maskedSeed = seed $\oplus$ seedMask;
9. A mensagem preenchida é o byte 0x00 concatenado com a maskedSeed e o maskedDB: EM = 0x00 || maskedSeed || maskedDB.

Para decifrar, é necessário seguir o algoritmo:

1. Passar o rótulo L na função de hash: lHash = Hash (L);
2. Para reverter o passo 9 da cifração, é preciso separar a mensagem EM no byte 0x00, em maskedSeed (de tamanho hLen) e maskedDB: EM = 0x00 || maskedSeed || maskedDB;
3. Gerar o seedMask usado para mascarar a semente: seedMask = MGF (maskedDB, hLen);

[2] Algoritmo de cifração e decifração implementado (com pequenas modificações) seguindo: https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

4. Para reverter o passo 8 da cifração, é preciso recuperar a semente através da seedMask: seed = maskedSeed ⊕ seedMask;
5. Gerar dbMask, usada para mascarar o bloco de dados: dbMask = MGF(seed, k − hLen − 1);
6. Para reverter o passo 6 da cifração, recuperar o bloco de dados DB: DB = maskedDB ⊕ dbMask;
7. Para reverter o passo 3 da cifração, é preciso separar o bloco de dados em suas diferentes partes: DB = lHash′ || PS || 0x01 || M.

É preciso checar as seguintes condições. Se alguma delas não for satisfeita, então o *padding* é inválido:

a. lHash' é igual a lHash;
b. PS tem apenas bytes de 0x00;
c. PS e M estão separados pelo byte 0x01 byte;
d. O primeiro byte de EM é 0x00.

***Mask Generation Function 1* (MGF1)**

Uma função de mascaramento (ou *mask generation function*, em inglês) é uma primitiva criptográfica semelhante a uma função de hash, com a diferença de que o tamanho da saída é variável. São funções completamente determinísticas - uma mesma entrada resulta sempre em uma mesma saída. No caso do *padding*, é útil porque um valor de hash fixo tornaria o padding mais vulnerável.

Para o *padding*, a MGF1 foi escolhida. Para entender o algoritmo utilizado na implementação, comecemos definindo os símbolos usados:

- Hash: função de hash (no caso, sha1);
- Z: semente que gera a máscara, uma string de octeto (entrada);
- l: tamanho desejado para a máscara, em octetos e no máximo de 2^32 (hLen) (entrada);
- mask: a máscara, uma string de octeto de tamanho l (saída).

O algoritmo para a implementação[3]:

1. Se l > 2^32 (hLen), tamanho maior do que o permitido e lança erro;
2. Seja T uma string de octeto vazia;
3. Para counter de 0 a ⌈lhLen⌉ − 1, faça:
    a. Converta counter para uma string de octeto C de tamanho 4;
    b. Concatenar o hash da semente Z e C para o string de octeto T: T = T ∥ Hash (Z ∥ C).
4. Retorna os primeiros l octetos de T como a máscara string de octeto.

Assinatura e verificação

[3] Algoritmo implementado (com pequenas modificações) seguindo: https://en.wikipedia.org/wiki/Mask_generation_function

Para a assinatura[4] do documento, foi utilizada a função de hash sha3-256. Para assinar o documento, passa-se pelas etapas:

1. Passa os dados de entrada não-cifrados pela função de hash;
2. Encripta a saída da função de hash com o algoritmo RSA, usando a chave pública fornecida.

Para checar a assinatura, é feito o processo de:

1. Passa os dados da mensagem decifrada pela função de hash;
2. Decifra a assinatura cifrada em RSA usando a chave privada;
3. Confere se o valor decifrado é igual à saída da função hash da mensagem decifrada. Se o valor for igual, então a assinatura está correta, senão não está.

[4] Algoritmo implementado (com pequenas modificações) seguindo: https://www.geeksforgeeks.org/rsa-and-digital-signatures/