Trabalho 3 - RSA
Segurança Computacional - Turma 01
Gabriel Borges - 202006401
Maycon Fabio - 200059742

1. Introdução

O trabalho pode ser dividido em três partes: geração de chaves e cifra, assinatura e verificação. Na primeira parte foram gerados 2 números primos aleatórios de 1024 bits utilizando o teste de primalidade de Miller-Rabin para poder construir as chaves. Para a cifração e decifração foi utilizado o RSA juntamente com o OAEP para preenchimento. Na segunda e terceira parte, foram usadas as funções definidas na primeira para assinar a mensagem e verificar a assinatura.

2. Implementação

2.1. Geração de chaves

A função *is_prime* implementa o teste de primalidade de Miller-Rabin, que é um algoritmo probabilístico para verificar se um número é primo. As condições iniciais garantem que se n for 2 ou 3 o retorno é imediatamente True, enquanto se n for menor ou igual a 1 ou for par o retorno é imediatamente False. Para os demais casos, é feita a decomposição de n - 1 em 2^k*m com m sendo ímpar. Para aumentar a confiança na verificação, o teste é repetido 40 vezes. Para cada iteração, um número aleatório a é escolhido no intervalo [2, n - 2]. Em seguida, calcula-se b = a^m mod n. Se b for 1 ou n - 1, o teste é inconclusivo, e a iteração é repetida. Caso contrário, realiza-se mais k - 1 iterações, calculando b = b^2 mod n em cada uma. Se em alguma dessas iterações b for igual a n - 1, o número é considerado provavelmente primo para esta iteração. Se após todas as iterações, o número ainda é considerado provavelmente primo, a função retorna True. Caso contrário, retorna False.

A função *generate_prime* apenas gera dois números aleatórios de 1024 bits e utiliza a função *is_prime* para tentar garantir que eles serão primos. Assim que os números passam no teste de primalidade, eles são retornados.

A função *generate_keys* começa gerando dois números primos distintos, p e q, chamando a função generate_prime(). Calcula-se n como o produto dos dois primos (n = p * q) e o totiente de Euler phi como (p - 1) * (q - 1). Inicializam-se “e” e “d” com valores arbitrários, onde ambos são inicializados como 2. Enquanto o produto de d e e

não for congruente a 1 módulo phi, a função tenta gerar um novo par de chaves pública e privada. A variável “e” é escolhida aleatoriamente no intervalo [2, phi - 1] e deve ser coprimo a phi, ou seja, o MDC entre e e phi é 1. A variável d é então calculada como o inverso multiplicativo modular de e em relação a phi, usando o algoritmo estendido de Euclides. Um número aleatório no intervalo [1e2, 1e6] é adicionado a d. Isso é feito para tornar mais difícil de ser descoberto por meio de ataques de força bruta. O retorno é um dicionário contendo as chaves públicas e privadas do tipo Key, definido anteriormente no código.

2.2. Cifração e decifração

A função *mgf1* implementa a função geradora de máscara MGF1 usada para gerar uma sequência de bytes pseudo-aleatórios com base em uma semente. Os parâmetros de entrada são a semente a ser utilizada para gerar a máscara (seed), o comprimento desejado da máscara em bytes (mask_len) e a função de hash a ser utilizada que por padrão é SHA-1 (hash_function). A variável hlen é o tamanho do resultante da função de hash. A variável masks é inicializada como uma sequência vazia de bytes e counter é inicializada como zero. Enquanto o comprimento de masks for menor que mask_len, a função continua gerando máscaras. O retorno são os primeiros mask_len bytes gerados no processo.

A função *encode_oaep* utiliza por padrão a função de hash sha1, mas é possível escolher outra se desejado. A variável lhash é apenas um hash de string vazia utilizada para compor o bloco de dados e hlen é o tamanho em bytes da saída da função de hash. A variável k contém o tamanho em bytes de n (módulo resultante de p * q) e mlen é o comprimento da mensagem também bytes. A variável ps contém os bytes de preenchimento de tamanho k - mlen - 2 * hLen - 2. O bloco de dados representado por db é a concatenação de lhash + ps + b'\x01' + bytes do conteúdo da mensagem. A seed é um número aleatório do tamanho de hlen. A seed é utilizada na entrada da função *mgf1* para gerar a máscara de tamanho k - hlen - 1. Essa máscara representada por dbMask é utilizada na operação xor juntamente com db para gerar o maskedDB. O mesmo serve de entrada para a função mgf1 para gerar outra máscara só que de tamanho hlen, armazenada em seedMask. Da mesma forma seedMask é utilizado na operação xor com a seed gerando a maskedSeed. Finalmente, encoded irá armazenar os bytes de saída da função, provenientes da concatenação de b'\x00' + maskedSeed + maskedDB. O retorno é a string no formato base64 formada pelos bytes armazenados em encoded.

A função *decode_oaep* faz exatamente o processo inverso da função *encode_oaep,* sendo usada na decifração de mensagens codificadas justamente para revertê-las na mensagem original.

A função *rsa* recebe um texto de entrada (input) e a chave (key) para ser utilizado tanto no processo de cifração quanto no de decifração. O algoritmo em seu estado puro apenas realiza a potenciação modular entre o input, a chave k e o módulo n. Os bytes resultantes são formatados em base64 e retornados como uma string.

As funções *cipher* e *decipher* apenas aproveitam a implementação das funções descritas anteriormente. Enquanto *cipher* usa o resultante de *encoded_oaep* como entrada para o *rsa*, o decipher usa o resultante de *rsa* como entrada para o *decode_oaep*. Dessa forma uma é o inverso da outra tornando possível transformar a mensagem original em mensagem codificada, e vice versa. Para que o usuário possa criptografar e descriptografar mensagens de qualquer tamanho, a mensagem original é dividida em partes de 215 bytes e a mensagem codificada em partes de 344 bytes.

2.3. Assinatura e verificação

A função *sign* assina a mensagem original utilizando a função de hash *sha3_256*. O resultante do hash então é utilizado como entrada para *rsa,* gerando a assinatura criptografada que é retornada como string formatada em base64.

A função *verify* primeiro decodifica tanto o ciphertext quanto a signature utilizando as funções criadas. A mensagem decodificada pela função *decipher* passa pela função de hash novamente para poder ser comparada e armazenada em msghash. A assinatura após ser decodificada usando *rsa* puro é armazenada em signaturebytes, com o detalhe de que só os últimos 32 bytes importam, correspondentes ao seu tamanho original. Se todos os bytes de signaturebytes e msghash forem iguais, a verificação retorna True, caso contrário retorna False.

3. Referências

https://www.youtube.com/watch?v=qdylJqXCDGs
https://youtu.be/oOcTVTpUsPQ?si=OrjrnWusJr-VNP2C
https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding
https://pt.wikipedia.org/wiki/RSA_(sistema_criptográfico)