

Faculdade de Tecnologia de Praia Grande - FATECPG

Entendendo e Implementando AES-256

Nome: Ana Beatriz Fernandes C. da Silva

Nome: Luis Fernando Gama de Oliveira

Turma: DSM 5

Professor: Alessandro Lima

Praia Grande, 2025

INTRODUÇÃO TEÓRICA

Fundamentos do AES (Advanced Encryption Standard):

O AES é um tipo de criptografia simétrica, o mesmo segredo (ou chave) é usado tanto para criptografar quanto para descriptografar uma mensagem. É muito usado hoje em dia por ser rápido e seguro, também está presente em várias aplicações, como em sites https, apps de mensagens e até em sistemas de armazenamento de dados.

A estrutura do AES é com blocos de 128 bits ou 16 bytes, o que significa que ele pega o texto e divide em partes desse tamanho para processar.

A diferença entre AES-128, AES-192 e AES-256 está no tamanho da chave, que é o segredo para criptografar, no caso da AES-256 a chave tem 256 bits (32 bytes) e o processo de criptografia passa por 14 etapas repetidas (rodadas) para embaralhar os dados e deixar praticamente impossível de descobrir o conteúdo sem a chave correta.

SubBytes, ShiftRows, MixColumns e AddRoundKey:

- **SubBytes**: cada byte do bloco é substituído por outro valor, seguindo uma tabela chamada S-box. Isso serve para embaralhar os dados e deixar tudo menos previsível.
- **ShiftRows**: as linhas do bloco de dados são deslocadas para a esquerda. A primeira linha não muda, a segunda anda um byte, a terceira anda dois e assim por diante. Isso espalha os dados e ajuda na segurança.
- **MixColumns**: os bytes de cada coluna são misturados entre si. Essa parte ajuda a espalhar ainda mais as informações, de forma que uma pequena mudança no texto original altera bastante o resultado final.
- **AddRoundKey**: o bloco é combinado com uma parte da chave (chamada subchave) usando uma operação matemática (XOR). É nesse momento que a chave realmente entra no processo e deixa o resultado dependente dela.

Essas etapas são repetidas várias vezes (14 rodadas no AES-256), e a cada rodada o texto vai ficando mais embaralhado, até virar o texto cifrado, que é o que ninguém consegue entender sem a chave.

Expansão de chave (Key Schedule):

A chave principal de 256 bits não é usada diretamente em todas as rodadas. O AES faz uma expansão de chave, ele cria várias “subchaves” a partir da chave original, uma para cada rodada do processo. Isso é feito com algumas transformações, como trocar bytes e rotacionar partes da chave, garantindo que cada rodada use uma subchave diferente. Então mesmo que alguém consiga descobrir parte do processo, ainda seria impossível descobrir a chave completa.

Modo de operação: GCM

Além do algoritmo em si, o AES precisa de um modo de operação para funcionar com mensagens maiores que 16 bytes. Existem vários modos, e eu escolhi o GCM (Galois/Counter Mode).

Foi escolhido esse modo porque ele é mais moderno e seguro. Ele não só criptografa os dados, mas também gera uma espécie de “assinatura”, que é chamada de tag, que serve para verificar se o conteúdo não foi alterado.

O GCM também usa um número aleatório chamado IV (vetor de inicialização) em cada cifragem, o que garante que duas mensagens iguais nunca vão gerar o mesmo resultado. Isso ajuda a manter a confidencialidade e também protege contra alterações nos dados.

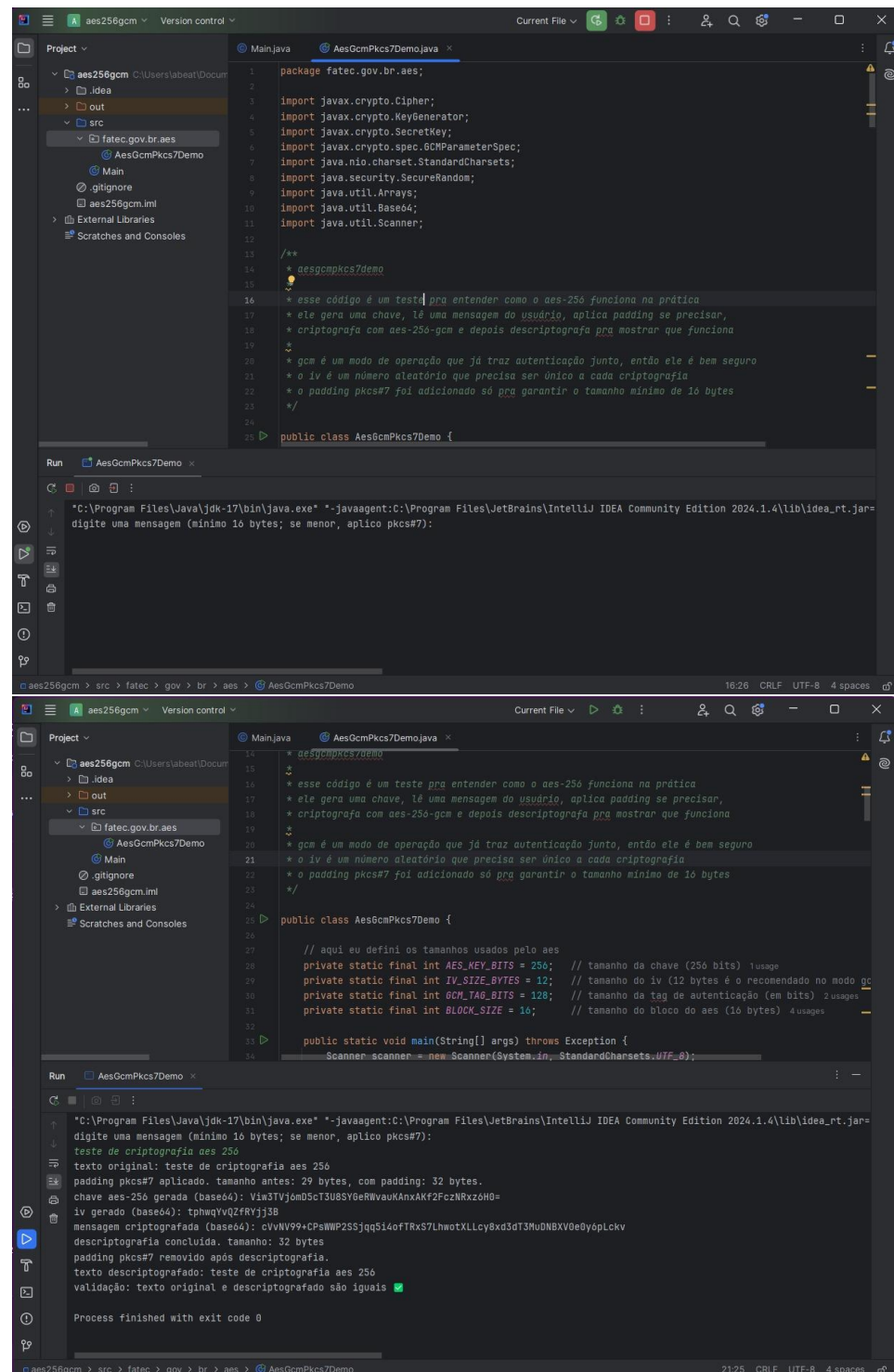
Resumindo, o AES-256 é uma forma muito segura de proteger informações. Ele usa várias etapas para embaralhar os dados, cria subchaves para cada rodada e com o modo GCM ainda garante que ninguém consiga alterar o conteúdo sem ser percebido.

EXEMPLO PRÁTICO

Nesta parte implementamos um pequeno programa em Java para entender como funciona a criptografia AES-256 no modo GCM.

O código gera uma chave, criptografa uma mensagem digitada e depois descriptografa pra validar se o processo foi bem-sucedido.

Repositório: <https://github.com/anabefernandes/AES-256-Encryption-Java/tree/main>



The image displays two screenshots of an IDE (IntelliJ IDEA) showing the development and execution of a Java program for AES-256 GCM encryption and decryption.

Top Screenshot: Shows the source code of the `AesGcmPkcs7Demo` class. The code imports necessary Java classes for AES, GCM, and random number generation. It includes a main method that prompts the user to enter a message (minimum 10 bytes) and a key (minimum 16 bytes). The code is currently commented out.

```
package fatec.gov.br.aes;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Base64;
import java.util.Scanner;

/**
 * esse código é um teste pra entender como o aes-256 funciona na prática
 * ele gera uma chave, lê uma mensagem do usuário, aplica padding se precisar,
 * criptografa com aes-256-gcm e depois descriptografa pra mostrar que funciona
 *
 * gcm é um modo de operação que já traz autenticação junto, então ele é bem seguro
 * o iv é um número aleatório que precisa ser único a cada criptografia
 * o padding pkcs#7 foi adicionado só pra garantir o tamanho mínimo de 16 bytes
 */

public class AesGcmPkcs7Demo {
```

Bottom Screenshot: Shows the same code, but with the main method uncommented. The output window displays the results of the execution, showing the original message, the generated key, the encrypted message, and the decrypted message, confirming that the process was successful.

```
Process finished with exit code 0
```

```

package fatec.gov.br.aes;

import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import java.nio.charset.StandardCharsets;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Base64;
import java.util.Scanner;

/**
 * aesgcmPkcs7demo
 *
 * esse código é um teste pra entender como o aes-256 funciona na
prática
 * ele gera uma chave, lê uma mensagem do usuário, aplica padding se
precisar,
 * criptografa com aes-256-gcm e depois descriptografa pra mostrar
que funciona
 *
 * gcm é um modo de operação que já traz autenticação junto, então
ele é bem seguro
 * o iv é um número aleatório que precisa ser único a cada
criptografia
 * o padding pkcs#7 foi adicionado só pra garantir o tamanho mínimo
de 16 bytes
 */

public class AesGcmPkcs7Demo {

    // aqui foi definido os tamanhos usados pelo aes
    private static final int AES_KEY_BITS = 256; // tamanho da
chave (256 bits)
    private static final int IV_SIZE_BYTES = 12; // tamanho do iv
(12 bytes é o recomendado no modo gcm)
    private static final int GCM_TAG_BITS = 128; // tamanho da tag
de autenticação (em bits)
    private static final int BLOCK_SIZE = 16; // tamanho do
bloco do aes (16 bytes)

    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in,
StandardCharsets.UTF_8);

        System.out.println("digite uma mensagem (mínimo 16 bytes; se
menor, aplico pkcs#7):");
        String input = scanner.nextLine();
        byte[] plaintext = input.getBytes(StandardCharsets.UTF_8);
        System.out.println("texto original: " + input);

        // aqui verifica se o texto precisa de padding pra completar
o tamanho do bloco
        // o padding pkcs#7 serve pra completar o último bloco quando
o texto não tem tamanho múltiplo de 16 bytes
        boolean padded = false;

```

```

        byte[] paddedPlaintext = plaintext;
        if (plaintext.length < BLOCK_SIZE || (plaintext.length %
BLOCK_SIZE) != 0) {
            paddedPlaintext = applyPkcs7Padding(plaintext,
BLOCK_SIZE);
            padded = true;
            System.out.println("padding pkcs#7 aplicado. tamanho
antes: " + plaintext.length +
                " bytes, com padding: " + paddedPlaintext.length
+ " bytes.");
        } else {
            System.out.println("não precisei aplicar padding, o
tamanho já está certo.");
        }
    }

```

```

        // gerar uma chave aes de 256 bits
        // essa chave é o segredo principal usado pra criptografar e
descriptografar
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(AES_KEY_BITS);
        SecretKey key = keyGen.generateKey();
        System.out.println("chave aes-256 gerada (base64): " +
Base64.getEncoder().encodeToString(key.getEncoded()));
        // obs: normalmente a chave nunca deve ser mostrada, aqui é
só pra teste mesmo
    }

```

```

        // agora gera o iv (vetor de inicialização), que é um número
aleatório usado junto com a chave
        // ele garante que, mesmo se criptografar a mesma mensagem
duas vezes, o resultado vai ser diferente
        byte[] iv = new byte[IV_SIZE_BYTES];
        SecureRandom sr = new SecureRandom();
        sr.nextBytes(iv);
        System.out.println("iv gerado (base64): " +
Base64.getEncoder().encodeToString(iv));
        // o iv não é segredo, mas tem que ser diferente a cada vez
que criptografar
    }

```

```

        // agora cria o objeto cipher, que é o que realmente faz a
criptografia
        // aqui uso o modo aes/gcm/nopadding (gcm = modo de operação
com autenticação)
        Cipher encryptCipher =
Cipher.getInstance("AES/GCM/NoPadding");
    }

```

```

        // o gcmparameterspec define o tamanho da tag de autenticação
e o iv que vai ser usado
        // a tag é usada pra garantir que o conteúdo não foi alterado
        GCMParameterSpec gcmSpecEnc = new
GCMParameterSpec(GCM_TAG_BITS, iv);
    }

```

```

        // inicializar o cipher no modo de criptografia, passando a
chave e o iv
        encryptCipher.init(Cipher.ENCRYPT_MODE, key, gcmSpecEnc);
    }

```

```

        // se quiser autenticar dados extras (que não são
criptografados), usaria updateAAD()
        // ex: encryptCipher.updateAAD("cabecalho".getBytes());
        // nesse caso não usamos, só deixamos o exemplo
    }

```

```

        // o aad é "additional authenticated data", garante
        // integridade mas não é secreto

        // aqui o cipher faz a criptografia de fato
        // o resultado final (ciphertext) já inclui a tag de
        // autenticação no final
        byte[] cipherBytes = encryptCipher.doFinal(paddedPlaintext);
        String cipherBase64 =
        Base64.getEncoder().encodeToString(cipherBytes);
        System.out.println("mensagem criptografada (base64): " +
        cipherBase64);

        // agora cria outro cipher pra descriptografar
        // ele precisa usar o mesmo modo, a mesma chave e o mesmo iv
        Cipher decryptCipher =
        Cipher.getInstance("AES/GCM/NoPadding");
        GCMParameterSpec gcmSpecDec = new
        GCMParameterSpec(GCM_TAG_BITS, iv);
        decryptCipher.init(Cipher.DECRYPT_MODE, key, gcmSpecDec);

        // se tivesse usado aad antes, precisaria repetir aqui pra
        // validar
        // decryptCipher.updateAAD(...)

        // aqui acontece a descriptografia, transformando o texto
        // criptografado de volta pro original
        byte[] decryptedPadded =
        decryptCipher.doFinal(Base64.getDecoder().decode(cipherBase64));
        System.out.println("descriptografia concluída. tamanho: " +
        decryptedPadded.length + " bytes");

        // se tinha padding antes, agora remove pra recuperar o texto
        // original
        byte[] decrypted;
        if (padded) {
            decrypted = removePkcs7Padding(decryptedPadded,
            BLOCK_SIZE);
            System.out.println("padding pkcs#7 removido após
            descriptografia.");
        } else {
            decrypted = decryptedPadded;
        }

        // converter os bytes descriptografados de volta pra string
        // legível
        String decryptedText = new String(decrypted,
        StandardCharsets.UTF_8);
        System.out.println("texto descriptografado: " +
        decryptedText);

        // aqui compara pra ver se o texto original e o
        // descriptografado são iguais
        if (Arrays.equals(plaintext, decrypted)) {
            System.out.println("validação: texto original e
            descriptografado são iguais ✓");
        } else {
            System.out.println("validação: texto original e
            descriptografado diferem ✗");
        }
    }

```

```

    }

    // essa função adiciona padding pkcs#7
    // ela serve pra completar o último bloco até chegar em 16 bytes
    private static byte[] applyPkcs7Padding(byte[] data, int
blockSize) {
        int padLen = blockSize - (data.length % blockSize);
        if (padLen == 0) padLen = blockSize;
        byte padByte = (byte) padLen;

        byte[] padded = Arrays.copyOf(data, data.length + padLen);
        for (int i = data.length; i < padded.length; i++) {
            padded[i] = padByte;
        }
        return padded;
    }

    // essa função remove o padding pkcs#7 depois da descriptografia
    // ela verifica os bytes extras e corta fora pra voltar o texto
original
    private static byte[] removePkcs7Padding(byte[] paddedData, int
blockSize) throws Exception {
        if (paddedData.length == 0 || paddedData.length %
blockSize != 0) {
            throw new Exception("tamanho inválido para remover
pkcs#7.");
        }

        int padLen = paddedData[paddedData.length - 1] & 0xFF;
        if (padLen < 1 || padLen > blockSize) {
            throw new Exception("padding pkcs#7 inválido.");
        }

        // aqui só confira se o padding é válido (não obrigatório,
mas é bom pra garantir)
        for (int i = paddedData.length - padLen; i <
paddedData.length; i++) {
            if (paddedData[i] != (byte) padLen) {
                throw new Exception("erro ao validar padding
pkcs#7.");
            }
        }

        // retorno o texto original sem o padding
        return Arrays.copyOf(paddedData, paddedData.length - padLen);
    }
}

```