Utilizando Criptografia Simétrica em Java

Veja neste artigo como funcionam os diferentes algoritmos que fazem parte desse tipo de criptografia. Além disso, veremos diversos exemplos práticos de como funciona a criptografia simétrica em Java.

Esse tipo de criptografia também é chamado de criptografia de chave única, criptografia de chave privada, criptografia de chave compartilhada, criptografia de chave secreta ou criptografia de chave convencional. Vale ressaltar que nem sempre temos apenas uma chave aqui, porém a outra chave é fortemente baseada na primeira. Dessa forma, podemos ter uma chave para cifragem e uma chave para decifragem que seja diferente da primeira, mas baseada naquela. Como um exemplo prático podemos imaginar que a primeira chave poderia ser "roma" e a segunda "amor", ou seja, o contrário da primeira, mas facilmente deduzível.

Normalmente também temos que a chave de cifragem é igual a chave de decifragem.

A transformação é dada caractere por caractere ou bit a bit. A execução é mais rápida se comparada a criptografia de chave assimétrica.

Utilizando a criptografia simétrica temos como garantia a confidencialidade e a integridade. A irretratabilidade e a autenticidade não são garantidas.

Basicamente temos como principais problemas na criptografia simétrica manter o sigilo da chave e a dificuldade no compartilhamento da chave devido os problemas de segurança nos canais de comunicação. Segue na **Figura 1** um esquema de como funciona basicamente a criptografia simétrica.

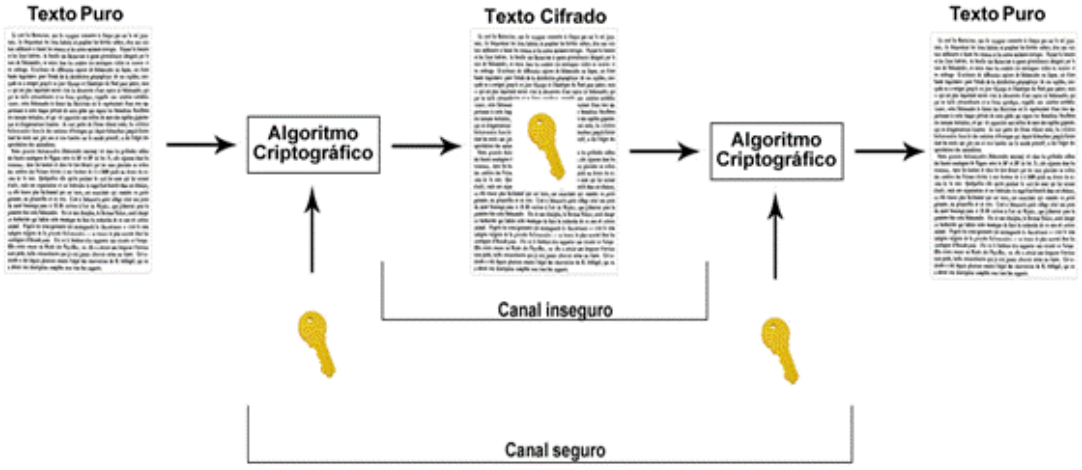

Figura 1. Funcionamento básico da Criptografia Simétrica

A chave geralmente é um número pequeno de até 256 bits e utilizamos o *RNG* (*Random Number Generator*) ou *PRNG* (*Pseudo Random Number Generator*) para a sua geração. Utilizando o RNG temos a geração de números aleatórios em que não há como repetir o processo ou podemos utilizar o PRNG onde temos pseudos números aleatórios gerados em que conseguimos gerar o mesmo número se tivermos a mesma situação que gerou o número anterior. Neste caso o processo de gerenciamento de chaves é complexo. Em média para se quebrar uma chave de 40 bits ao custo de 100 mil dólares têm-se um tempo estimado no total de dois segundos para quebrá-la, porém se tivermos uma chave de 128 bits e gastarmos 10 trilhões de dólares em recursos para quebrar essa chave teremos um total estimado de dez elevado na décima primeira potência (10^{11}) de anos para quebrá-la. Esses são estudos comprovados baseando-se em tecnologia atual e de ponta.

Entre os sistemas criptográficos simétricos temos: *IDEA*, *TwoFish*, *BlowFish*, *Serpent*, *DES*, *AES*, *RC5*, *RC6*. Esses também são chamados de criptografia "*De Bloco*" ou "*Baseada em Bloco*". Nesse tipo de criptografia são processados blocos de informação de uma só vez, concatenando-os no final do processo. Outros dois sistemas criptográficos são o *RC4* e *OTP* que são chamados criptografia "*De Fluxo*" ou "*Baseada em Fluxo*" em que nesse tipo de criptografia é processado cada bit da mensagem individualmente (processamento bit a bit).

Nas próximas seções veremos um pouco mais sobre as criptografias de bloco destacando o *AES* e o *DES* e na sequencia veremos a criptografia de fluxo destacando o *RC4* e *OTP*.

Cifragem de Bloco

Na cifragem de bloco o texto é dividido em blocos antes da cifragem, assim a cifragem é aplicada em cima de cada bloco. Com isso cada bloco pode ser tratado paralelamente. Os blocos possuem o mesmo tamanho, portanto, quando um bloco não fecha com o tamanho total há um preenchimento. O tamanho do bloco não pode ser muito pequeno nem muito grande para evitar problemas. As chaves podem ser reutilizadas. Cabeçalhos são inseridos no arquivo cifrado informando qual foi o algoritmo usado, qual o tamanho de bloco e qual o tamanho real do arquivo para que no momento da decriptação o padding (preenchimento para blocos menores) seja descartado.

As cifras em bloco são consideradas melhores para criptografar dados estáticos, pois já sabemos antecipadamente o tamanho e assim podemos dividir em blocos de M bits. Um possível problema com cifras em bloco é o fato da existência de blocos repetitivos que acabam por criar um padrão. Para evitar o reconhecimento de padrões repetitivos usam-se os "*feedback modes*" como o *Electronic Code Book* (*EBC*), *Cipher Block Chaining* (*CBC*), *Cipher Feedback Block* (*CFB*) e *Output Feedback Block* (*OFB*). No *ECB* a mensagem é dividida em blocos de tamanho adequado, cada bloco é cifrado de forma separada e os blocos cifrados são concatenados na mesma ordem. O problema nesta técnica é que blocos da mensagem original idênticos vão produzir blocos com cifras idênticas, o que não é desejável. O *CBC* faz uma operação XOR do bloco de texto puro com o texto cifrado anteriormente e então ocorre o processo de codificação. Um vetor de inicialização é utilizado para iniciar o processo, visto que não existe nenhum texto cifrado para o bloco inicial. O *CFB* opera com cada bloco de texto cifrado anteriormente codificado e o resultado é combinado com o bloco de texto puro através de uma operação de XOR para produzir o bloco cifrado atual. Neste caso também se utiliza um vetor de inicialização para iniciar o processo. O *OFB* é similar ao *CFB*, exceto pelo fato de que a quantidade de XOR com cada bloco de texto puro é gerada independentemente do bloco de texto puro ou do bloco de texto cifrado.

Outro problema na cifragem de bloco reside na distribuição da chave, visto que os canais de comunicação não são seguros.

Abaixo veremos como funcionam os dois principais algoritmos da cifragem de bloco: o DES e o AES.

DES (Data Encryption Standard)

O DES é um algoritmo de criptografia de bloco que utiliza uma chave simétrica de 64 bits em disco (armazenada), uma chave de 56 bits para execução quando ela for submetida (8 bits restantes são usados para paridade e depois descartados) e 16 subchaves de 48 bits. Essas 16 subchaves são derivadas da chave anterior de 56 bits através de um total de 16 iterações.

O mesmo método usado na criptografia é usado na descriptografia, sendo que a diferença está na sequencia que as subchaves são utilizadas.

Entre os problemas do DES está a chave que é relativamente pequena (56 bits), e por isso foi criado o método "Whitening" e o método de "Multiplicidade".

O Whitening é constituído por um texto puro de 56 bits, uma chave de 56 bits, o algoritmo DES para processamento e tem como saída um texto cifrado de 64 bits. Portanto, o processo começa com a entrada de um texto puro e uma chave de 64 bits que será processada pelo DES em conjunto com outra chave de 56 bits, que totaliza uma chave de 120 bits, no momento posterior ao processamento pelo DES temos ainda outra chave de 64 bits que não agrega força ou segurança, pois ela é derivada da primeira.

Algumas das evoluções do DES é o 2DES e o 3DES que aumentam o número de chaves no processo.

Segue na **Listagem 1** um exemplo utilizando a biblioteca padrão do Java para criptografar e descriptografar um texto puro utilizando o DES.

Listagem 1. Criptografando e Descriptografando dados com DES

```
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;

public class EncriptaDecriptaDES
{


       public static void main(String[] argv) {

             try{

                 KeyGenerator keygenerator = KeyGenerator.getInstance("DES");
                 SecretKey chaveDES = keygenerator.generateKey();

                 Cipher cifraDES;

                 // Cria a cifra
                 cifraDES = Cipher.getInstance("DES/ECB/PKCS5Padding");

                 // Inicializa a cifra para o processo de encriptação
                 cifraDES.init(Cipher.ENCRYPT_MODE, chaveDES);

                 // Texto puro
                 byte[] textoPuro = "Exemplo de texto puro".getBytes();

                 System.out.println("Texto [Formato de Byte] : " + textoPuro);
                 System.out.println("Texto Puro : " + new String(textoPuro));
```

```
                // Texto encriptado
                byte[] textoEncriptado = cifraDES.doFinal(textoPuro);

                System.out.println("Texto Encriptado : " + textoEncriptado);

                // Inicializa a cifra também para o processo de decriptação
                cifraDES.init(Cipher.DECRYPT_MODE, chaveDES);

                // Decriptografa o texto
                byte[] textoDecriptografado = cifraDES.doFinal(textoEncriptado);

                System.out.println("Texto Decriptografado : " + new String(textoDecriptografado));

            }catch(NoSuchAlgorithmException e){
                   e.printStackTrace();
            }catch(NoSuchPaddingException e){
                   e.printStackTrace();
            }catch(InvalidKeyException e){
                   e.printStackTrace();
            }catch(IllegalBlockSizeException e){
                   e.printStackTrace();
            }catch(BadPaddingException e){
                   e.printStackTrace();
            }

        }

}
```

Inicialmente criamos uma instância de *Cipher* e especificamos o nome do algoritmo, o modo e o esquema de padding que é opcional, tudo separado por uma barra "/" conforme o código abaixo:

```
Cipher cifraDES;
  // Cria a cifra
  cifraDES = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

Após isso convertemos o texto para byte conforme destacado abaixo:

```
  // Texto puro
  byte[] textoPuro = "Exemplo de texto puro".getBytes();
```

Para criptografar o texto puro utilizamos o método *Cipher.doFinal(),* conforme mostra o código abaixo:

```
  // Inicializa a cifra para o processo de encriptação
  cifraDES.init(Cipher.ENCRYPT_MODE, chaveDES);
  // Texto encriptado
  byte[] textoEncriptado = cifraDES.doFinal(textoPuro);
```

Podemos verificar que o método *doFinal* também espera um *array de bytes*, por isso realizamos a conversão anteriormente.

Por fim, utilizamos o método *Cipher.doFinal()* em modo de decriptação para descriptografar o texto puro conforme o código destacado abaixo:

```
  // Inicializa a cifra também para o processo de decriptação
  cifraDES.init(Cipher.DECRYPT_MODE, chaveDES);
  // Decriptografa o texto
  byte[] textoDecriptografado = cifraDES.doFinal(textoEncriptado);
```

Para se aprofundarmos mais nas funcionalidades que estão disponíveis para os desenvolvedores podemos visitar a especificação oficial no site do Java em http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html.

O algoritmo *DES* é hoje considerado inseguro na sua forma original sendo desenvolvido há mais de 20 anos, e nestes 20 anos não apareceu nenhuma descrição de um caminho para quebrá-lo, exceto pela força bruta. O DES possui uma versão mais poderosa conhecida por *3DES* ou *TDES* (*Triple DES*). Outro algoritmo que superou o *DES* e tem ganhado espaço em várias aplicações onde o *DES* não se mostra adequadamente seguro é o *AES* (*Advanced Encryption Standard*) que será explicado mais abaixo.

AES (Advanced Encryption Standard)

O *Advanced Encryption Standard* (*AES*) é uma cifra de bloco sucessora do *DES* que surgiu através de um concurso promovido pelo governo dos Estados Unidos para substituir o *DES*. Entre as condições necessárias para a candidatura de um algoritmo foram especificadas as seguintes características obrigatórias que o algoritmo deveria possuir: divulgação aberta e pública, livre de direitos autorais, e ser de chave privada (simétricos) que suporte blocos de 128 bits e chaves de 128, 192 e 256 bits. Três anos e meio após o início do concurso, o comitê responsável chegou à escolha do vencedor: *Rijndael*. O nome é uma fusão de *Vincent Rijmen* e *Joan Daemen*, os dois belgas criadores do algoritmo.

A criptografia *AES* usa o algoritmo de criptografia *Rijndael*, que envolve métodos de substituição e permutação para criar dados criptografados de uma mensagem.

O *AES* foi oficialmente anunciado em 26 de novembro de 2001 e tornou-se um padrão em 26 de maio de 2002. O *AES* atualmente é dos algoritmos mais populares usados para criptografia de chave simétrica.

O AES tem como principais características segurança, desempenho, facilidade de implementação, flexibilidade e exige pouca memória, o que o torna adequado para operar em ambientes restritos como *Smart cards*, *PDAs* e telefones celulares.

Basicamente o AES opera sobre um arranjo bidimensional de bytes com 4x4 posições. Para criptografar, cada turno do AES consiste em quatro estágios: *AddRoundKey* onde cada byte do estado é combinado com a subchave própria do turno e cada subchave é derivada da chave principal usando-se um algoritmo de escalonamento de chaves, SubBytes que consiste de uma substituição não linear onde cada byte é substituído por outro de acordo com uma tabela de referência, ShiftRows que consiste de uma transposição onde cada fileira do estado é deslocada em um determinado número de posições, MixColumns que consiste de uma operação de mescla que opera nas colunas do estado e combina os quatro bytes de cada coluna usando uma transformação linear. Porém o último turno é diferente onde o estágio de MixColumns é substituído por um novo estágio de AddRoundKey.

Tanto em hardware quanto em software o *AES* bastante é rápido, sendo utilizado nos processadores da Intel. Como o *AES* é um algoritmo de chave simétrica, podemos utilizar a chave tanto para criptografar quanto para descriptografar a mensagem. A chave em questão será representada em 256 bits, o que significa que se alguém tentar quebrar a mensagem teria que descobrir o valor da chave de 256 bits. A tecnologia para decifrar uma chave de 256 bits em uma quantidade razoável de tempo ainda não foi inventada.

Segue na **Listagem 2** um exemplo utilizando a biblioteca padrão do Java para criptografar e descriptografar um texto puro utilizando o *AES*.

Listagem 2. Criptografando e Descriptografando dados com *AES*.

```
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
```

```
import javax.crypto.Cipher;

public class EncriptaDecriptaAES {

	static String IV = "AAAAAAAAAAAAAAAA";
	static String textopuro = "teste texto 12345678\0\0\0";
	static String chaveencriptacao = "0123456789abcdef";

	public static void main(String [] args) {

		try {

			System.out.println("Texto Puro: " + textopuro);

			byte[] textoencriptado = encrypt(textopuro, chaveencriptacao);

			System.out.print("Texto Encriptado: ");

			for (int i=0; i<textoencriptado.length; i++)
				System.out.print(new Integer(textoencriptado[i])+" ");

			System.out.println("");

			String textodecriptado = decrypt(textoencriptado, chaveencriptacao);

			System.out.println("Texto Decriptado: " + textodecriptado);

		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public static byte[] encrypt(String textopuro, String chaveencriptacao) throws Exception {
		Cipher encripta = Cipher.getInstance("AES/CBC/PKCS5Padding", "SunJCE");
		SecretKeySpec key = new SecretKeySpec(chaveencriptacao.getBytes("UTF-8"), "AES");
		encripta.init(Cipher.ENCRYPT_MODE, key,new IvParameterSpec(IV.getBytes("UTF-8")));
		return encripta.doFinal(textopuro.getBytes("UTF-8"));
	}

	public static String decrypt(byte[] textoencriptado, String chaveencriptacao) throws Exception{
		Cipher decripta = Cipher.getInstance("AES/CBC/PKCS5Padding", "SunJCE");
		SecretKeySpec key = new SecretKeySpec(chaveencriptacao.getBytes("UTF-8"), "AES");
		decripta.init(Cipher.DECRYPT_MODE, key,new IvParameterSpec(IV.getBytes("UTF-8")));
		return new String(decripta.doFinal(textoencriptado),"UTF-8");
	}

}
```

Executando o algoritmo acima teremos como resultado a saída abaixo:

```
Texto Puro: teste texto 12345678
Texto Encriptado: 80 20 103 61 -80 -122 -15 -45 116 -53 73 -33 55 -58 -99 25 -106 124 -117 -18 37 -127 86 112 
Texto Decriptado: teste texto 12345678
```

Podemos verificar a semelhança com o algoritmo anterior (*DES*), porém com diferenças nas configurações dos métodos.

Cifragem de Fluxo

Diferentemente das cifras em bloco que operam em blocos de dados, as cifras de fluxo operam em unidades menores, geralmente bits, o que as tornam bem mais rápidas. As cifras de fluxo geram uma sequência de bits que serão usados como chave, essa sequencia gerada também é conhecida como *keystream*, a partir de uma chave inicial. A encriptação basicamente ocorre pela combinação do texto puro com a chave através de operações *XOR*.

Neste cifra não é necessário termos um bloco para cifrar, ciframos o que temos e, se for desejado, no momento que quisermos. Os algoritmos utilizados no passado em que uma mensagem era cifrada letra a letra e não precisávamos agrupa-las para completar um bloco são considerados como cifras de fluxo.

Abaixo veremos como funcionam os dois principais algoritmos de cifragem de fluxo: *RC4* e o *OTP*.

RC4

O *RC4* é um algoritmo que promove transformações lineares não sendo necessários cálculos complexos, visto que o sistema funciona basicamente por meio de permutações e somas de valores inteiros, o que o torna muito simples e rápido.

A chave pode ter de zero (inexistente) até 256 bytes. Ele é utilizado nos padrões *SSL/TLS*, *WEP* e *WPA*, ou seja, é muito utilizado em aplicações web e redes sem fio sendo o primeiro algoritmo disponível nas redes sem fio. É a cifragem simétrica de fluxo mais utilizada e possui como princípio de funcionamento o segredo criptográfico perfeito, ou seja, a chave do tamanho da mensagem. É considerado seguro num contexto prático (duração das transações). No entanto, de forma geral os adeptos da criptografia não consideram o *RC4* um dos melhores sistemas criptográficos, e em algumas aplicações são considerados como sendo muito inseguros.

O RC4 foi inicialmente desenvolvido por *Ronald Rivest* para a empresa a *RSA Data Security, Inc.*, líder mundial em algoritmos de criptografia e foi durante muito tempo um segredo comercial muito bem guardado e popular utilizando por grandes softwares como *Lotus Notes*, *Apple*, *Oracle Secure SQL*, *Internet Explorer*, *Netscape*, *Adobe Acrobat*, entre outros.

De uma forma geral, o algoritmo *RC4* consiste em utilizar um *array* que a cada utilização tem os seus valores permutados, e misturados com a chave, o que provoca uma dependência muito forte com esta chave. Esta chave, utilizada na inicialização do *array*, pode ter até 256 bytes (2048 bits). Entretanto o algoritmo é mais eficiente quando temos uma chave menor devido a superior perturbação aleatória induzida no *array*.

Segue na **Listagem 3** um exemplo de um algoritmo em Java para criptografar e descriptografar uma mensagem em texto puro utilizando o *RC4*.

Listagem 3. Criptografando e Descriptografando dados com RC4.

```
import java.security.InvalidKeyException;

public class EncriptaDecriptaRC4 {

    private char[] key;
    private int[] sbox;
    private static final int SBOX_LENGTH = 256;
    private static final int TAM_MIN_CHAVE = 5;

    public static void main(String[] args) {
        try {

             EncriptaDecriptaRC4 rc4 = new EncriptaDecriptaRC4("testkey");
            char[] textoCriptografado = rc4.criptografa("Teste de Mensagem de Texto Puro".toCharArray());
            System.out.println("Texto Criptografado:\n" + new String(textoCriptografado));
            System.out.println("Texto Decriptografado:\n"
```

```
                + new String(rc4.decriptografa(textoCriptografado)));

    } catch (InvalidKeyException e) {
        System.err.println(e.getMessage());
    }
}

public EncriptaDecriptaRC4(String key) throws InvalidKeyException {
    setKey(key);
}

public EncriptaDecriptaRC4() {

}

public char[] decriptografa(final char[] msg) {
    return criptografa(msg);
}

public char[] criptografa(final char[] msg) {
    sbox = initSBox(key);
    char[] code = new char[msg.length];
    int i = 0;
    int j = 0;
    for (int n = 0; n < msg.length; n++) {
        i = (i + 1) % SBOX_LENGTH;
        j = (j + sbox[i]) % SBOX_LENGTH;
        swap(i, j, sbox);
        int rand = sbox[(sbox[i] + sbox[j]) % SBOX_LENGTH];
        code[n] = (char) (rand ^ (int) msg[n]);
    }
    return code;
}

private int[] initSBox(char[] key) {
    int[] sbox = new int[SBOX_LENGTH];
    int j = 0;

    for (int i = 0; i < SBOX_LENGTH; i++) {
        sbox[i] = i;
    }

    for (int i = 0; i < SBOX_LENGTH; i++) {
        j = (j + sbox[i] + key[i % key.length]) % SBOX_LENGTH;
        swap(i, j, sbox);
    }
    return sbox;
}

private void swap(int i, int j, int[] sbox) {
    int temp = sbox[i];
    sbox[i] = sbox[j];
    sbox[j] = temp;
}

public void setKey(String key) throws InvalidKeyException {
    if (!(key.length() >= TAM_MIN_CHAVE && key.length() < SBOX_LENGTH)) {
        throw new InvalidKeyException("Tamanho da chave deve ser entre "
                + TAM_MIN_CHAVE + " e " + (SBOX_LENGTH - 1));
    }

    this.key = key.toCharArray();
}
```

```
}
```

OTP (One-Time-Pad)

A *OTP* (em português cifra de uso único ou chave de uso único) foi primeiramente descrita pelo banqueiro e criptografista *Frank Miller* no ano de 1882. No ano de 1917 a *OTP* foi reinventada e, poucos anos depois, registrada. Ela é derivada da cifra de *Vernam* que consistia numa cifra que combinava uma mensagem com uma chave lida através de uma fita perfurada o que a tornava vulnerável porque a fita com as chaves era reutilizada ao chegar ao fim. Anos mais tarde o renomado autor *Claude Shannon*, nos anos 1940, comprovou cientificamente a segurança do sistema *OTP*.

Na *OTP* temos uma chave aleatória do mesmo tamanho da mensagem, diferente da *RC4* que a chave de tamanho original é menor. Se a chave for verdadeiramente aleatória, nunca reutilizada, e mantida em segredo, a *One-Time-Pad* pode ser inquebrável. Por isso este algoritmo é incondicionalmente seguro.

Nesse algoritmo temos que cada letra da cifra será combinada de uma forma padrão a uma letra da mensagem. Dessa forma, atribui-se a cada letra um valor numérico, por exemplo, a letra A é 0, B é 1, C é 2 e assim por diante, até Z que é 25. A técnica é combinar a chave e a mensagem usando a adição modular. Assim, somamos os numéricos de cada letra da mensagem ao seu correspondente na cifra. Devemos lembrar-nos de considerar o módulo 26, de modo que, se uma soma ultrapassa 26, o número deve ser dividido por 26, tomando-se, no lugar dele, apenas o resto da divisão inteira. Para uma melhor compreensão de como funciona o algoritmo podemos tomar como exemplo a mensagem HELLO que teria como resultado H=7, E=4, L=11, O=14, lembrando que A é 0. Também devemos ter uma chave, suponhamos que seja XMCKL, assim teríamos X=23, M=12, C=2, K=10 e L=11. Somando-se a chave com a mensagem teríamos H+X que é 7+23 resultando em 30, E+M que é 4+12 resultando em 16, L+C que é 11+2 resultando em 13, L+K que é 11+10 resultando em 21 e O+L que é 14+11 resultando em 25. Após somarmos a mensagem com a chave temos como resultado 4, 16, 13, 21 e 25 conforme verificamos anteriormente. Se tomarmos cada uma dessas somas teremos uma letra correspondente e todas essas letras concatenadas será a nossa mensagem criptografada. Dessa forma, temos como resultado “4” que corresponde a letra “E”, “16” que corresponde a letra “Q”, “21” que corresponde a letra “V” e “25” que corresponde a “Z”. Portanto nossa mensagem criptografada é “EQNVZ”. No processo de decifração subtraímos a mensagem cifrada pela chave. Assim para o nosso exemplo teríamos E(4)-X(23) que é -19 e por isso somamos com +26 para que o resultado seja positivo, Q(16)-12(M) que é 4, N(13)-C(2) que é 11, V(21)-K(10) que é 11 e Z(25)-L(11) que é 14. Dessa forma, temos como resultado o “7” que corresponde ao “H”, “4” que corresponde ao “E”, “11” que corresponde ao “L”, “11” que novamente corresponde ao “L” e “14” que corresponde ao “O”. Dessa forma, temos a mensagem original recriada "HELLO".

Um detalhe importante é que após a transmissão ambos (transmissor e receptor) devem destruir completamente a chave impedindo sua reutilização e evitando um eventual ataque à cifra.

A operação de *XOR* é usada frequentemente para combinar o texto puro e os elementos chaves, sendo uma operação muito rápida para os computadores.

Um dos problemas é que devemos criar quantidades gigantes de chaves aleatórias. Além disso, temos o problema da distribuição e da proteção das chaves de forma segura, e dependendo do ambiente também temos um problema devido o grande consumo de banda.

Segue na **Listagem 4** um exemplo de um algoritmo em Java para criptografar e descriptografar texto puro utilizando o OTP.

Listagem 4. Criptografando e Descriptografando dados com OTP.

```
import java.util.Random;

public class EncriptaDecriptaOTP {

	public String criptografa(String mensagem, String chave) {
		if (mensagem.length() != chave.length()) error("O tamanho da mensagem e da chave devem ser igu
		int[] im = charArrayToInt(mensagem.toCharArray());
		int[] ik = charArrayToInt(chave.toCharArray());
		int[] data = new int[mensagem.length()];

		for (int i=0;i<mensagem.length();i++) {
			data[i] = im[i] + ik[i];
		}

		return new String(intArrayToChar(data));
	}

	public String decriptografa(String mensagem, String chave) {
		if (mensagem.length() != chave.length()) error("O tamanho da mensagem e da chave devem ser igu
		int[] im = charArrayToInt(mensagem.toCharArray());
		int[] ik = charArrayToInt(chave.toCharArray());
		int[] data = new int[mensagem.length()];

		for (int i=0;i<mensagem.length();i++) {
			data[i] = im[i] - ik[i];
		}

		return new String(intArrayToChar(data));
	}

	public String genKey(int tamanho) {
		Random randomico = new Random();
		char[] key = new char[tamanho];
		for (int i=0;i<tamanho;i++) {
			key[i] = (char) randomico.nextInt(132);
			if ((int) key[i] < 97) key[i] = (char) (key[i] + 72);
			if ((int) key[i] > 122) key[i] = (char) (key[i] - 72);
		}

		return new String(key);
	}

	public static void main(String[] args){
	    EncriptaDecriptaOTP otp = new EncriptaDecriptaOTP();
		String menssagem = "hello";
		String chave = otp.genKey(menssagem.length());
		String msgCriptografada = otp.criptografa(menssagem, chave);
		String msgDecriptografada = otp.decriptografa(msgCriptografada, chave);

		System.out.println("Menssagem: "+menssagem);
		System.out.println("Chave: "+chave);
		System.out.println("Mensagem Criptografada: "+msgCriptografada);
		System.out.println("Mensagem Decriptografada: "+msgDecriptografada);

	}

	private int chartoInt(char c) {
		return (int) c;
	}

	private char intToChar(int i) {
		return (char) i;
	}
```

```
        private int[] charArrayToInt(char[] cc) {
                int[] ii = new int[cc.length];
                for(int i=0;i<cc.length;i++){
                        ii[i] = chartoInt(cc[i]);
                }
                return ii;
        }

        private char[] intArrayToChar(int[] ii) {
                char[] cc = new char[ii.length];
                for(int i=0;i<ii.length;i++){
                        cc[i] = intToChar(ii[i]);
                }
                return cc;
        }

        private void error(String msg) {
                System.out.println(msg);
                System.exit(-1);
        }
}
```

Se executarmos o código acima teremos como resultado:

```
Menssagem: hello
Chave: xIOOJ
Mensagem Criptografada: à®»»¹
Mensagem Decriptografada: hello
```

Como a chave é randômica note que a cada execução teremos resultados diferentes.

Bibliografia

[1] MessageDigest, disponível em http://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html

[2] STALLINGS, William. Criptografia e segurança de redes: Princípios e práticas, 4 ed. São Paulo: Prentice Hall, 2008.

por Higor Medeiros
Expert em Java e programação Web