Criptografia Avançada em Java
Boas Práticas e Exemplos em Java 8, 11 e 17

Neste artigo, abordaremos a aplicação de criptografia em Java de maneira avançada, utilizando as versões 8, 11 e 17 do JDK.

A criptografia, tanto simétrica quanto assimétrica, é essencial para proteger dados sensíveis, e é crucial adotar boas práticas de segurança e utilizar os recursos mais recentes oferecidos por cada versão.

Os exemplos de código foram enriquecidos com comentários detalhados para um entendimento completo das melhores práticas e otimizações.

Criptografia Simétrica com AES

A criptografia simétrica utiliza a mesma chave tanto para cifrar quanto para decifrar dados. O AES (Advanced Encryption Standard) é o padrão mais usado por ser rápido e seguro.

Exemplo em Java 8:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class AESEncryption {
    public static void main(String[] args) throws Exception {
        // Geração de chave simétrica AES com tamanho de 128 bits
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(128);  // Tamanho mínimo recomendado
        SecretKey secretKey = keyGen.generateKey();

        // Instância de Cipher configurada para AES
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);

        // Texto a ser criptografado
        byte[] encrypted = cipher.doFinal("Texto a ser criptografado".getBytes());

        // Codificação em Base64 para garantir integridade ao armazenar
        System.out.println(Base64.getEncoder().encodeToString(encrypted));
    }
}
```

Boas Práticas:

- **Gerenciamento de Chaves**: Certifique-se de armazenar as chaves de maneira segura, como em HSMs (Hardware Security Modules).
- **Uso de Salt e IV**: Sempre utilize um vetor de inicialização (IV) aleatório para garantir segurança adicional, mesmo se o texto plano for similar.

Exemplo em Java 17 (AES/GCM):

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class AESEncryption {
    public static void main(String[] args) throws Exception {
        // Geração de chave AES com 256 bits para maior segurança
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(256);  // Tamanho maior para aumentar a robustez
        SecretKey secretKey = keyGen.generateKey();

        // AES/GCM adiciona autenticação ao processo de criptografia
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);

        // Texto a ser criptografado
        byte[] encrypted = cipher.doFinal("Texto a ser criptografado".getBytes());

        // Codificação em Base64
        System.out.println(Base64.getEncoder().encodeToString(encrypted));
    }
}
```

Boas Práticas:

- **AES/GCM**: GCM (Galois/Counter Mode) não só criptografa, mas também autentica os dados, o que previne ataques de alteração.
- **Tamanho da Chave**: Usar 256 bits é recomendado para sistemas que exigem alta segurança, como sistemas financeiros.

Criptografia Assimétrica com RSA

A criptografia assimétrica utiliza um par de chaves — pública e privada. O RSA é amplamente usado para troca segura de chaves e assinatura digital.

Exemplo em Java 8:

```
import javax.crypto.Cipher;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Base64;

public class RSAEncryption {
    public static void main(String[] args) throws Exception {
        // Geração de par de chaves RSA com 2048 bits
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
        keyPairGen.initialize(2048);  // Tamanho mínimo recomendado para segurança
        KeyPair pair = keyPairGen.generateKeyPair();

        // Instância de Cipher configurada para RSA
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, pair.getPublic());

        // Texto a ser criptografado
        byte[] encrypted = cipher.doFinal("Texto a ser criptografado".getBytes());
```

```
        // Codificação em Base64
        System.out.println(Base64.getEncoder().encodeToString(encrypted));
    }
}
```

Boas Práticas:

- **Tamanho de Chave**: 2048 bits é o tamanho mínimo recomendado. Para maior segurança, especialmente em transações sensíveis, use 3072 bits ou mais.
- **Combinando RSA e AES**: Para grandes volumes de dados, criptografe os dados com AES e use RSA para criptografar a chave simétrica. Isso melhora o desempenho.

Exemplo em Java 17 (com OAEP):

```
import javax.crypto.Cipher;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Base64;

public class RSAEncryption {
    public static void main(String[] args) throws Exception {
        // Geração de par de chaves RSA com 3072 bits para maior segurança
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
        keyPairGen.initialize(3072);
        KeyPair pair = keyPairGen.generateKeyPair();

        // RSA com OAEP e SHA-256 para maior segurança contra ataques de padding
        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(Cipher.ENCRYPT_MODE, pair.getPublic());

        // Texto a ser criptografado
        byte[] encrypted = cipher.doFinal("Texto a ser criptografado".getBytes());

        // Codificação em Base64
        System.out.println(Base64.getEncoder().encodeToString(encrypted));
    }
}
```

Boas Práticas:

- **Padding Seguro**: Use RSA/OAEP (Optimal Asymmetric Encryption Padding) para proteger contra ataques de padding (ataques de Bleichenbacher).
- **Tamanho da Chave**: 3072 bits é recomendado para sistemas com vida útil prolongada.

Hashing com SHA-256

O hashing é uma técnica usada para garantir a integridade dos dados, criando uma "impressão digital" única. SHA-256 é amplamente utilizado por sua segurança e desempenho.

Exemplo em Java 8:

```
import java.security.MessageDigest;
import java.util.Base64;
```

```
public class HashingExample {
    public static void main(String[] args) throws Exception {
        // Instância de SHA-256
        MessageDigest digest = MessageDigest.getInstance("SHA-256");

        // Geração do hash
        byte[] hash = digest.digest("Texto a ser hashado".getBytes());

        // Codificação em Base64
        System.out.println(Base64.getEncoder().encodeToString(hash));
    }
}
```

Boas Práticas:

- **Uso de Salting**: Combine o uso de salting com hashing para evitar ataques de dicionário e rainbow tables.
- **Evitar MD5 e SHA-1**: São considerados inseguros para aplicações modernas devido a vulnerabilidades conhecidas.

Conclusão

Este artigo demonstrou a aplicação de criptografia em Java nas versões 8, 11 e 17, com exemplos práticos e boas práticas de segurança.

A evolução das versões do Java trouxe melhorias significativas em termos de suporte a algoritmos e desempenho, o que reforça a importância de manter seus sistemas atualizados.

Seguir boas práticas, como o uso adequado de padding, tamanhos de chave e algoritmos modernos, é fundamental para garantir a segurança de dados sensíveis em qualquer aplicação.

EducaCiência FastCode para a comunidade