8.金融密语：揭秘支付系统的加解密艺术_V20240113

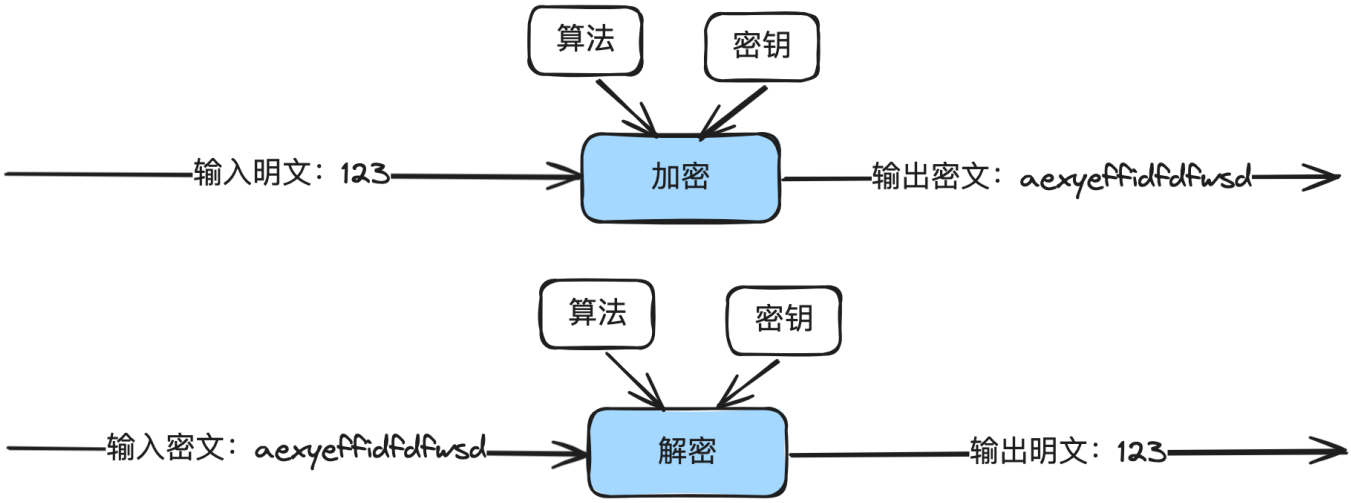
本文主要讲清楚加解密技术在支付系统中的重要地位，核心应用场景，哪些是安全的算法，哪些是不安全的算法，以及对应的核心代码实现。

通过这篇文章，你可以了解到：

1. 什么是加解密
2. 支付系统中哪些核心场景需要用到加解密技术
3. 哪些是安全的加解密算法，哪些是不安全的加解密算法
4. 常见加解密算法核心代码
5. 日常研发过程中常见的问题

1. 什么是加密解密

在数字经济的舞台上，在线支付系统扮演着至关重要的角色。究竟是什么技术让金钱能够在公开的互联网上安全无忧地穿梭？加密和解密是这舞台背后隐秘的艺术，同时确保资金的安全流转和个人信息安全。

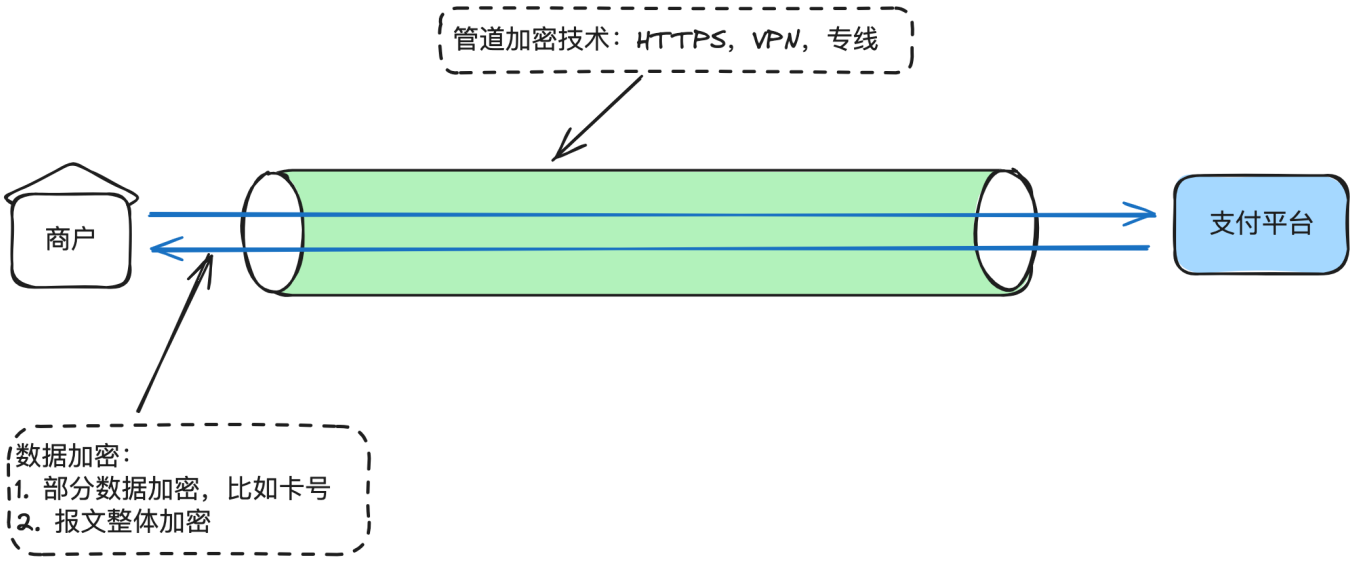


在数字通信中，**加密**是将明文通过一定的算法和密钥转换成无法识别的密文的过程。这样即使数据被截获，未经授权的第三方也无法理解其内容。

解密则是加密的逆向过程，通过一定的算法和密钥将密文转换成明文的过程。

2. 核心应用场景

支付系统做为一个安全系数非常高的系统，加解密技术在里面起到了极其重要的作用。通常以下几个核心应用场景都会用到加解密技术：1）传输加密；2）存储加密。



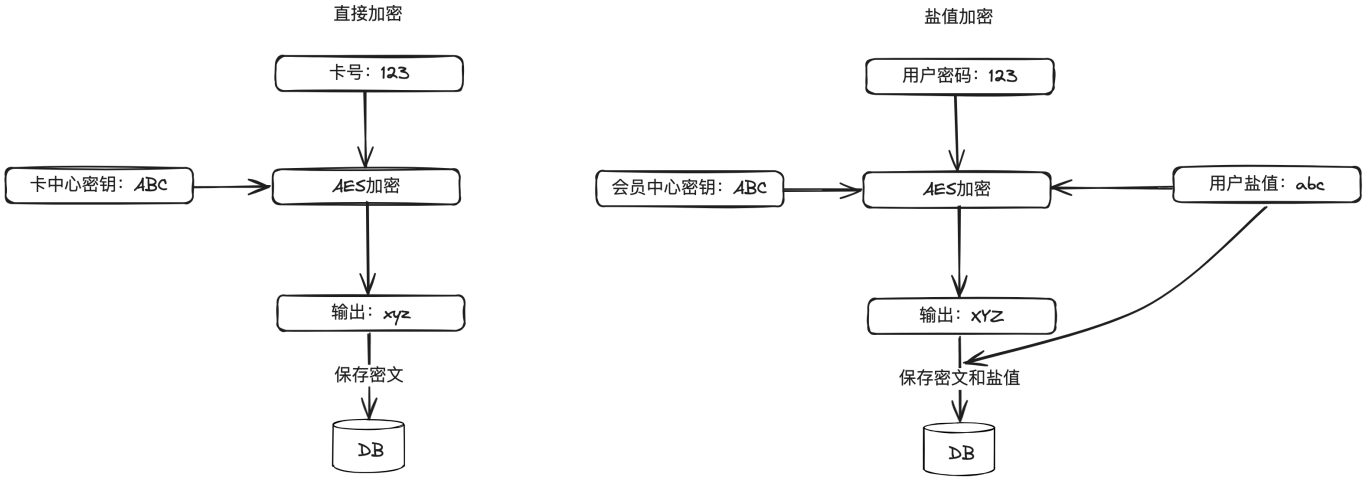
1. 传输加密：保护交易数据在互联网上传输过程中的安全，防止数据被窃听或篡改。

具体的实现通常有两种：

1）**通道加密**：比如使用HTTPS，或者VPN、专线等，实现数据传输端到端的加密。HTTPS和VPN可以参考网络上公开的文档。

2）**部分字段单独加密**：比如把卡号等关键信息进行加密后再发出去。

3）**整体报文单独加密**：先生成业务报文，然后对整个报文加密再发出去。



2. **存储加密**：对敏感数据比如信用卡信息、用户身份证信息、密码等需要进行加密后存储到数据库中，以防止数据泄露。

   具体的实现通常也会分两种：

   1）**直接加密**：原始信息直接加密。通常用于信用卡、身体证等常规数据的加密。

   2）**加盐值（SALT）后再加密**：原始信息先加上盐值，然后再进行加密。通常用于密码管理。

3. 密码的特殊处理

密码的存储比较特殊，值得单独说一说。

前面有说过，登录或支付密码需要加上盐值后，再进行加密存储。那为什么密码管理需要使用盐值？为了**提高密码安全性**。

1. **防止彩虹表攻击**。彩虹表是一种预先计算出来的哈希值数据集，攻击者可以使用它来查找和破译未加盐的密码。通过为每个用户加盐，即使是相同的密码，由于盐值不同，加密后的密文也是不一样的。
2. **保护相同密码的用户**。如果多个用户使用了相同的密码，没有盐值情况下，一个被破解后，就

能找到使用相同密码的其它用户。每个用户不同的盐值，确保生成的密文不同。

3. **增加破解难度**。尤其是密码较弱时，显著增加攻击者难度。

在实现时，需要留意加盐策略：

1. **随机和唯一**：每个用户都是随机和唯一的。
2. **存储盐值**：每个用户的密码和盐值都需要配对存储。因为在加密密钥更新时，需要使用盐值一起先解密再重新加密。
3. **盐值足够长**：增加复杂性，推荐至少128位。

4. 加解密算法选择推荐

推荐的算法如下：

AES：当前最广泛使用的对称加密算法，速度快，适用于高速加密大量数据。密钥长度推荐256或以上。

RSA：广泛使用的非对称加密算法，安全性比AES更高，但是加密速度慢，适用于小量数据或做为数字签名使用。密钥长度推荐2048或以上。

在https里面，数据加密使用AES，AES密钥通过RSA加密后传输，这样既解决了安全性，又解决了加密速度的问题。

当前公认不够安全的算法，不推荐使用，主要有：

DES：密钥长度较短，不够安全。

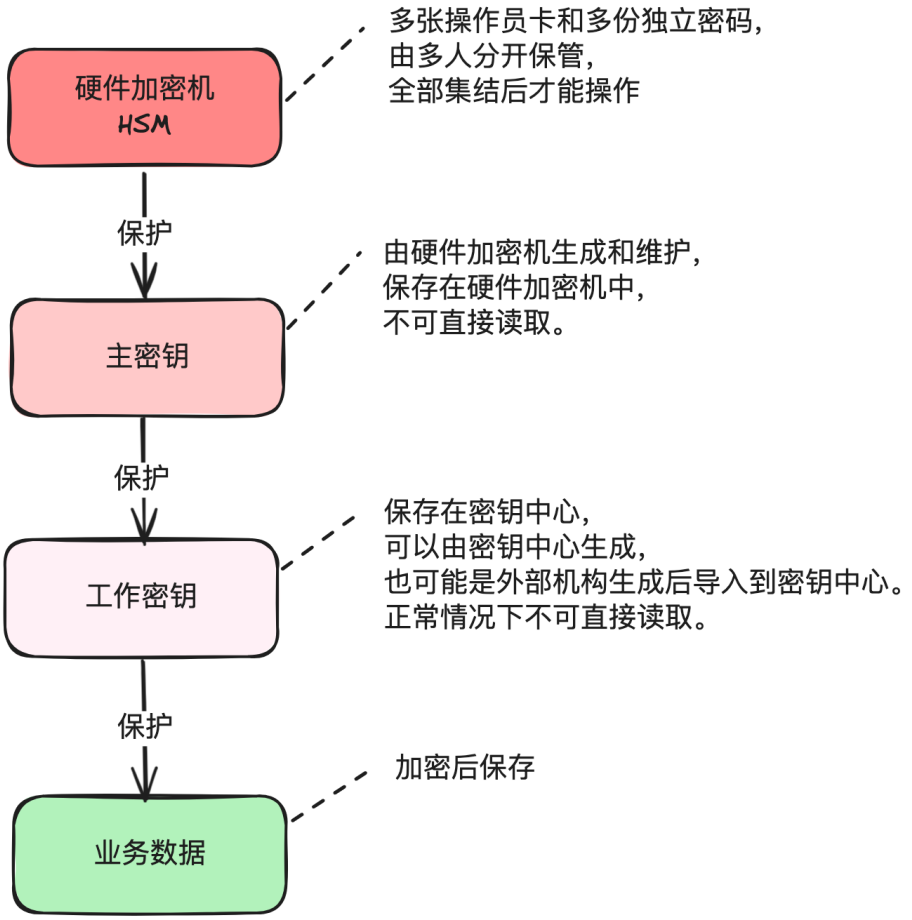
特别强调一点：**千万千万不要自己去发明一种【私有的】，【自己认为很安全】的算法，并应用到生产环境**。因为业界推荐的这些算法的安全性是经过大量数字家和计算机科学家论证过的，也经过工业界持续地验证，每天都有无数的攻击或破解在进行，一旦有被破解的风险就会很快知道。

5. 加密密钥的存储及更新

明文数据被加密存储，安全了，那加密明文数据的密钥怎么办?

加密密钥有多重要呢？有一个公式是这样的：密钥的价值 = 密文的价值。比如你加密存储的密文价值10亿，那对应的密钥价值也有10亿。

密钥的管理涉及4个方面：密钥存储、更新、备份和恢复、废止和销毁。



密钥存储：

安全存储环境：密钥保存在特殊的安全环境中，包括服务器、网络环境、硬件加密机等。

最小权限原则：管理密钥的人越少越好。

密钥分为**主密钥**和**工作密钥**，其中工作密钥用来加解密普通的业务数据，而主密钥用来加解密工作密钥。

一般来说主密钥应该存储在专门的硬件安全模块（HSM）中，俗称：硬件加密机，安全性极高。但是相对来说性能有限，且价格昂贵，管理复杂。

工作密钥一般由主密钥加密后保存在DB中，在需要的时候调用主密钥解密后，缓存在内存中，然后再去加解密普通的业务数据。

密钥更新机制：

1. 需要定期更新，减少被破解的风险。
2. 自动定时更新，减少人为失误。‘
3. 版本控制和回滚：要有版本号，要能快速回滚。

密钥备份和恢复，废止和销毁等机制，以及如何设计主密钥和工作密钥等细节，后面在介绍密钥中心设计与实现的章节再详细说明。

6. 常见加解密算法核心代码

以经常使用的AES加解密为例：

Java

```java
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.security.spec.KeySpec;
import java.util.Base64;
import java.security.SecureRandom;
import javax.crypto.SecretKey;

public class AESWithPasswordExample {
    private static final String PASSWORD = "123456";
    private static final int ITERATION_COUNT = 65536;
    private static final int KEY_LENGTH = 256; // AES密钥长度可以是128、192或256比特
    private static final String ALGORITHM = "AES/CBC/PKCS5Padding";

    // 使用PBKDF2从密码派生AES密钥
    private static SecretKey getKeyFromPassword(String password) throws Exception {
        SecureRandom random = new SecureRandom();
        byte[] salt = new byte[16];
        random.nextBytes(salt); // 创建安全随机盐

        KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, ITERATION_COUNT, KEY_LENGTH);
        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
        byte[] secretKey = factory.generateSecret(spec).getEncoded();
        return new SecretKeySpec(secretKey, "AES");
    }

    // 加密
    public static String encrypt(String data, SecretKey key, IvParameterSpec iv) throws Exception {
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, key, iv);
        byte[] encrypted = cipher.doFinal(data.getBytes("UTF-8"));
        return Base64.getEncoder().encodeToString(encrypted);
    }

    // 解密
    public static String decrypt(String encryptedData, SecretKey key, IvParameterSpec iv) throws Exception {
        Cipher cipher = Cipher.getInstance(ALGORITHM);
```

```
            cipher.init(Cipher.DECRYPT_MODE, key, iv);
            byte[] original = cipher.doFinal(Base64.getDecoder().decode(encryptedData));
            return new String(original, "UTF-8");
        }

        public static void main(String[] args) throws Exception {
            String originalData = "Confidential data that needs to be encrypted and decrypted";

            // 生成密钥和初始化向量（IV）
            SecretKey key = getKeyFromPassword(PASSWORD);
            byte[] ivBytes = new byte[16]; // AES使用16字节的IV
            SecureRandom random = new SecureRandom();
            random.nextBytes(ivBytes);
            IvParameterSpec iv = new IvParameterSpec(ivBytes);

            // 加密数据
            String encryptedData = encrypt(originalData, key, iv);
            System.out.println("Encrypted data: " + encryptedData);

            // 解密数据
            String decryptedData = decrypt(encryptedData, key, iv);
            System.out.println("Decrypted data: " + decryptedData);
        }
    }
```

请注意，这是一个简化的版本，实际应用中需要采取更多的安全措施，比如加密密钥的存储，盐值需要一起保存等。

7. 日常研发过程中的常见问题

曾经碰到的常见问题有：

密钥管理不规范：把密钥加密后保存在数据库，但是加密密钥用的密钥是123456。

算法选择不合适：大批量数据选择使用速度极慢的非对称的RSA算法。

兼容性算法不对：尤其是模式、填充方式是直接影响加解密结果的。比如AES下面仍然细分为：ECB，CBC，CFB，OFB，CTR，GCM等模式，以及PKCS7/PKCS5填充，零填充等填充方式。具体的可以找密码学相关资料参考。

异想天开地使用自己创造的私有算法：以为很安全，其实太傻太天真。

管理机制不完善：没有制定严格的规范，或有规范执行不严重，导致密钥能被轻易访问。

8. 结束语

在数字支付世界里，加解密是支付系统安全的基石之一，和众多安全措施一起保护用户和平台的资产。安全的加解密算法，严谨的管理密钥，是支付系统安全的两大支柱。

加解密涉及的密码学是一个很大的领域，支付系统的安全则是一个更大的领域，因篇幅关系，这里只介绍了一些入门知识，不过对于支付系统日常研发已经足够。

这是《百图解码支付系统设计与实现》专栏系列文章中的第（8）篇。和墨哥（隐墨星辰）一起深入解码支付系统的方方面面。

欢迎转载。

Github（PDF文档全集，不定时更新）：https://github.com/yinmo-sc/Decoding-Payment-System-Book

公众号：隐墨星辰。

有个小群不定时解答一些问题或知识点，有兴趣的同学可先加微信（yinmo_sc）后进入，添加微信请备注：加支付系统设计与实现讨论群。


隐墨星辰
扫一扫上面的二维码图案，加我为朋友。