7.揭密支付安全：为什么你的交易无法被篡改_V20240202

本文主要讲清楚支付系统中为什么要做签名验签，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. 什么是签名验签
2. 支付系统为什么一定要做签名验签
3. 哪些是安全的算法，哪些是不安全的算法
4. 常见签名验签算法核心代码
5. 联调中常见的问题

1. 什么是签名验签

在电子支付的万亿市场中，安全无疑是核心中的核心。有一种称之为“签名验签”的技术在支付安全领域发挥着至关重要的作用。那什么是签名验签呢?

签名验签是数字加密领域的两个基本概念。

签名：发送者将数据通过特定算法和密钥转换成一串唯一的密文串，也称之为数字签名，和报文信息一起发给接收方。

验签：接收者根据接收的数据、数字签名进行验证，确认数据的完整性，以证明数据未被篡改，且确实来自声称的发送方。如果验签成功，就可以确信数据是完好且合法的。

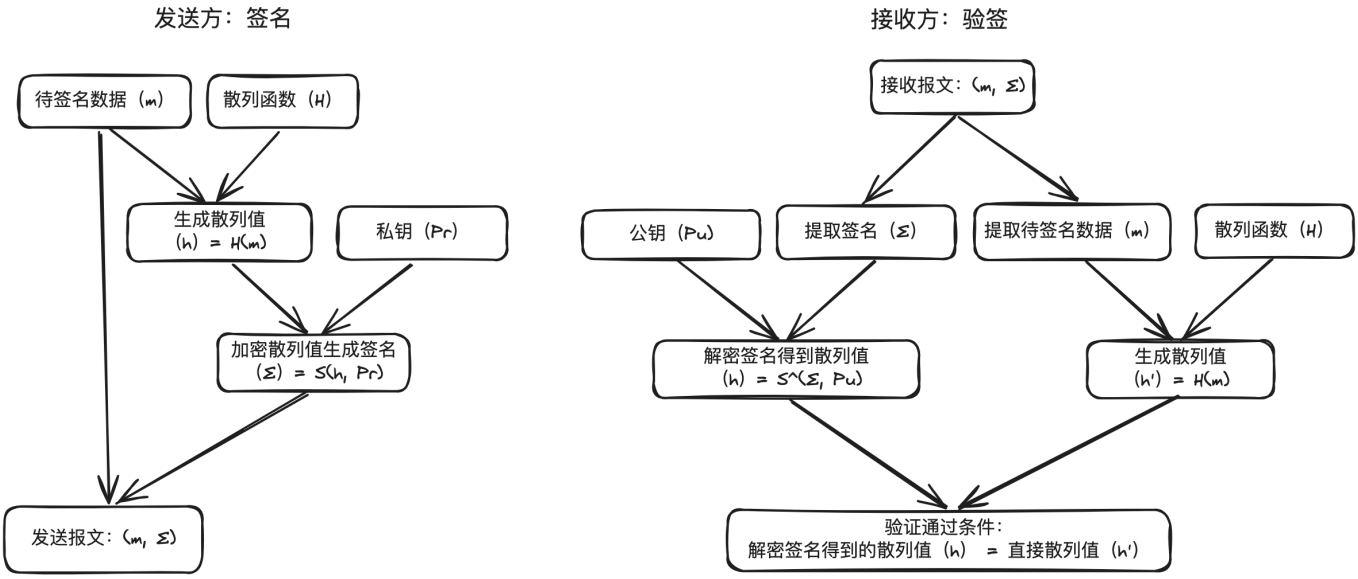
假设被签名的数据（m），签名串（Σ），散列函数（H），私钥（Pr），公钥（Pu），加密算法（S），解密算法（S^），判断相等（eq）。

简化后的数学公式如下：

签名：Σ=S[H(m), Pr]。

验签：f(v)=[H(m) eq S^(Σ, Pu)]。

流程如下：



签名流程：

1. **散列消息**：对消息(m)应用散列函数（H）生成散列值（h）。
2. **加密散列值**：使用发送方的私钥（Pr）对散列值（h）进行加密，生成签名（Σ）。 Σ = S(h, Pr)

把数字签名（Σ）和原始消息（m）一起发给接收方。

验签流程：

1. **散列收到的消息**：使用同样的散列函数（H）对消息（m）生成散列值（h'）。 h' = H(m)

2. **解密签名**：使用发送方的公钥（Pu）对签名（Σ）进行解密，得到散列值（h）。 h = S^(Σ, Pu)
3. **比较散列值**：比较解密得到的散列值（h）与直接对消息（m）散列得到的（h'）是否一致。验证成功条件： h = h' 。

如果两个散列值相等，那么验签成功，消息（m)被认为是完整的，且确实来自声称的发送方。如果不一致，就是验签失败，消息可能被篡改，或者签名是伪造的。

现实中的算法会复杂非常多，比如RSA，ECDSA等，还涉及到填充方案，随机数生成，数据编码等。

2. 支付系统为什么一定要做签名验签

银行怎么判断扣款请求是从确定的支付平台发出来的，且数据没有被篡改？商户不承认发送过某笔交易怎么办？这都是签名验签技术的功劳。

签名验签主要解决3个问题：

1. **身份验证：确认支付信息是由真正的发送方发出，防止冒名顶替。**

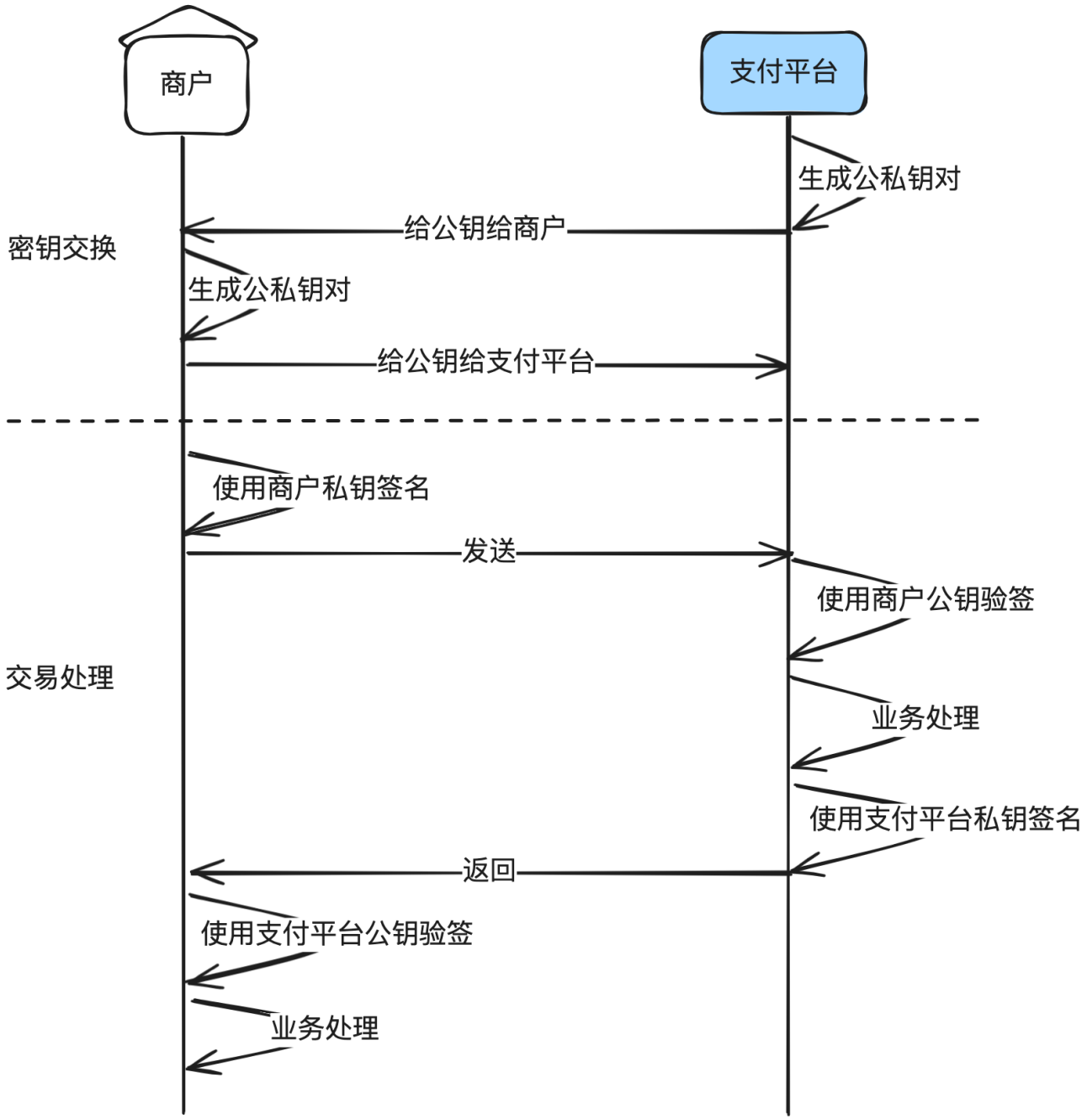
如果无法做身份验证，支付宝就无法知道针对你的账户扣款99块的请求是真实由你楼下小卖部发出去的，还是我冒充去扣的款。

2. **完整性校验：确认支付信息在传输过程中未被篡改，每一笔交易都是完整、准确的。**

如果无法校验完整性，那么我在公共场景安装一个免费WIFI，然后截获你的微信转账请求，把接收者修改成我的账号，再转发给微信，微信就有可能会把钱转到我的账号里。

3. **防抵赖性：避免任何一方否则曾经进行过的交易，提供法律证据支持。**

比如微信支付调用银行扣款100块，银行返回成功，商户也给用户发货了，几天后银行说这笔扣款成功的消息不是他们返回的，他们没有扣款。而签名验签就能让银行无法抵赖。



流程：

1. 双方先交换密钥，可以通过线下邮件交换，也可以通过线上自助平台交换。
2. 请求方发出交易报文前使用自己的私钥进行签名，接收方接收报文后先进行验签，验签通过后再进行业务处理。
3. 接收方处理完业务，返回前使用自己的私钥进行签名，请求方接收返回报文后先进行验签，验签通过后再进行业务处理。

3. 安全签名验签算法推荐

安全一直是一个相对的概念，很多曾经是安全的算法，随着计算机技术的发展，已经不安全了，以后到了量子计算的时代，现在大部分的算法都将不再安全。

一般而言，安全同时取决于算法和密钥长度。比如SHA-256就比MD5更安全，RSA-2048就比RSA-1024更安全。

已经被认为**不安全的算法有MD5、SHA-1等算法**，容易受到碰撞攻击，不应该在支付系统中使用。

仍然被认为是安全的算法有：SHA-256，SHA-3， RSA-1024，RSA-2048，ECDSA等。

当前最常见**推荐的算法是RSA-2048**。RSA-1024以前使用得多，但因为密钥长度较短，也已经不再推荐使用。

SHA-256只是一种单纯的散列算法，其实是**不适合**做签名验签算法的，因为**需要双方共用一个API密钥**，一旦泄露，无法确认是哪方被泄露，也就是**只解决了完整性校验，无法解决身份验证和防抵赖性**。但因为使用简单，国内外仍然有不少的支付公司公司在大量使用。

4. 常见签名验签算法核心代码

下面以RSA（SHA256withRSA）为例，示例代码如下：

```java
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

public class RSASignatureUtil {

    // 使用私钥对数据进行签名
    public static byte[] sign(byte[] data, byte[] privateKey) throws Exception {
        PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(privateKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PrivateKey priKey = keyFactory.generatePrivate(pkcs8KeySpec);
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initSign(priKey);
        signature.update(data);
        return signature.sign();
    }

    // 使用公钥验证签名
    public static boolean verify(byte[] data, byte[] publicKey, byte[] signatureBytes) throws Exception {
        X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PublicKey pubKey = keyFactory.generatePublic(keySpec);
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initVerify(pubKey);
        signature.update(data);
        return signature.verify(signatureBytes);
    }
}

```

签名输出是字节码，还需要编码，一般是base64。

如果使用SHA-256（很多公司仍在使用，但不推荐），如下：

```java
import java.security.MessageDigest;

public class SHA256Util {

    // 使用SHA-256对数据进行散列
    public static byte[] hash(byte[] data) throws Exception {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        return digest.digest(data);
    }
}

```

这里data已经是加了API密钥（也称为API KEY）。所谓的API密钥，就是交易双方共享的一个密钥，这样双方生成的哈希值才会一致。

5. 联调中常见的问题

不管是与商户的联调，还是与支付渠道（或银行）之间的联调，签名验签都是非常耗费精力的环节。验签不通过通常有以下几个情况：

1. **密钥不匹配**：双方以为自己都配置了正确的密钥，但实际没有。
2. **数据编码不一致**：比如一方使用GBK，一方使用UTF-8。
3. **原始数据选择不一致**：比如接口文档要求拼接10个字段，但是代码实现却只拼接了9个字段。或者一方没有把空值放入计算，另一方把空值也放入计算。
4. **原始数据排序方式不一致**：比如接口要求按key的升序排列，调用方却忘记排序就进行签名。
5. **字符转义不一致**：特殊字段的转义必须保持一致。

解决上述问题的最好办法，就是**让服务提供方提供一段示例代码，以及示例报文+示例签名，然后在本地使用main方法先跑成功，再移植到项目代码中。**

6. 结束语

本章主要讲了签名验签名的概念，对于支付系统的重要性，以及常见签名验签名算法及JAVA代码实现。

但是还有一个同样非常重要的问题没有讲：**如何安全储存密钥？**如果密钥放在代码里或数据库里，开发人员是可以直接获得的，如果不小心泄露出去怎么办？

应对的解决方案就是创建一个**密钥中心**专门负责密钥的管理，无论加密解密还是签名验签，全部调用密钥中心来处理，**业务系统不接触密钥明文**。

那又来了一个新的问题：这个密钥中心如何设计和实现，才能既**保证很高的安全性**，又能有**非常高的性能**表现呢？

后面有机会再开一个密钥中心的设计和实现专题来聊。

这是《百图解码支付系统设计与实现》专栏系列文章中的第（7）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。