非对称密钥沉思系列（4）：密钥交换 原创

修改于 2023-01-04 14:33:54 2.9K 13

密钥交换的概念

密钥交换，也有称作密钥协商，这套机制，最主要的作用是用来得到通信双方的临时会话密钥。

这里的临时会话密钥，可以理解为对称加密的密钥，只不过他的有效性仅限于一次会话链接，并不是长期有效的。

前面其实我们分析过，分对称密钥比如RSA也是可以做加解密的，那么在实际的通信过程比如TLS中，为什么还需要生成临时的对称密钥呢？

这里最核心的原因有两个：

- 非对称密钥加解密，对于数据量有比较严格的要求，比如RSA算法，在使用OAEP填充模式时，每次最多只能加密190字节。

```
mLen = k - 2 * hLen - 2 if we want to calculate the maximum message size:
    k - length in octets of the RSA modulus n
    hLen - output length in octets of hash function Hash
    mLen - length in octets of a message M
k = 2048 / 8 = 256
hLen = 256 / 8 = 32 (SHA256的输出是256bits)
256 - 2 * 32 - 2 = 256 - 64 - 2 = 190字节

RSA基于密钥长度来做密文运算，所以有长度限制
```

- 非对称密钥加解密的性能相对于对称密钥，差了很多，在这实际的业务流加解密中，无法进行业务落地。
  因此在实际的工程化上，一般使用非对称密钥进行数据密钥的协商与交换，而使用数据密钥与对称加密算法进行数据流的加解密保护。

基于RSA的密钥交换

简单的密钥交换过程

基于RSA进行密钥交换，基于非对称密钥的两个基本特性：

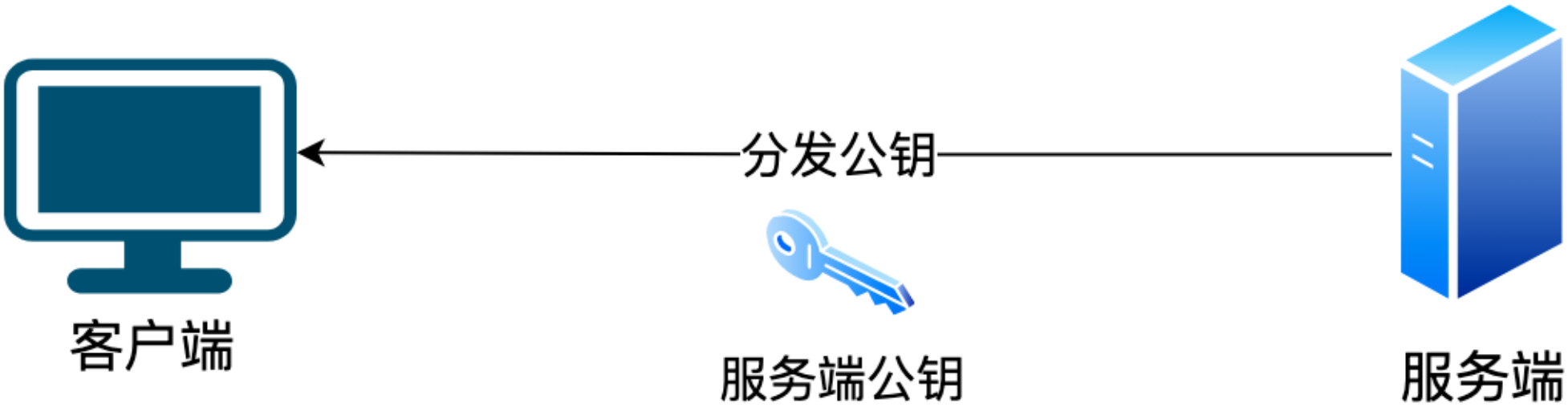
- 使用公钥加密、私钥解密，且此过程无法逆向
- 公钥是对外公开的，私钥是私密不公开的

客户端与服务端

在简单的密钥交换场景中，有两个基本角色，客户端与服务端。

客户端与服务端进行正常的业务流通信前，总是需要先线上一把数据密钥，用于给业务流进行加解密。

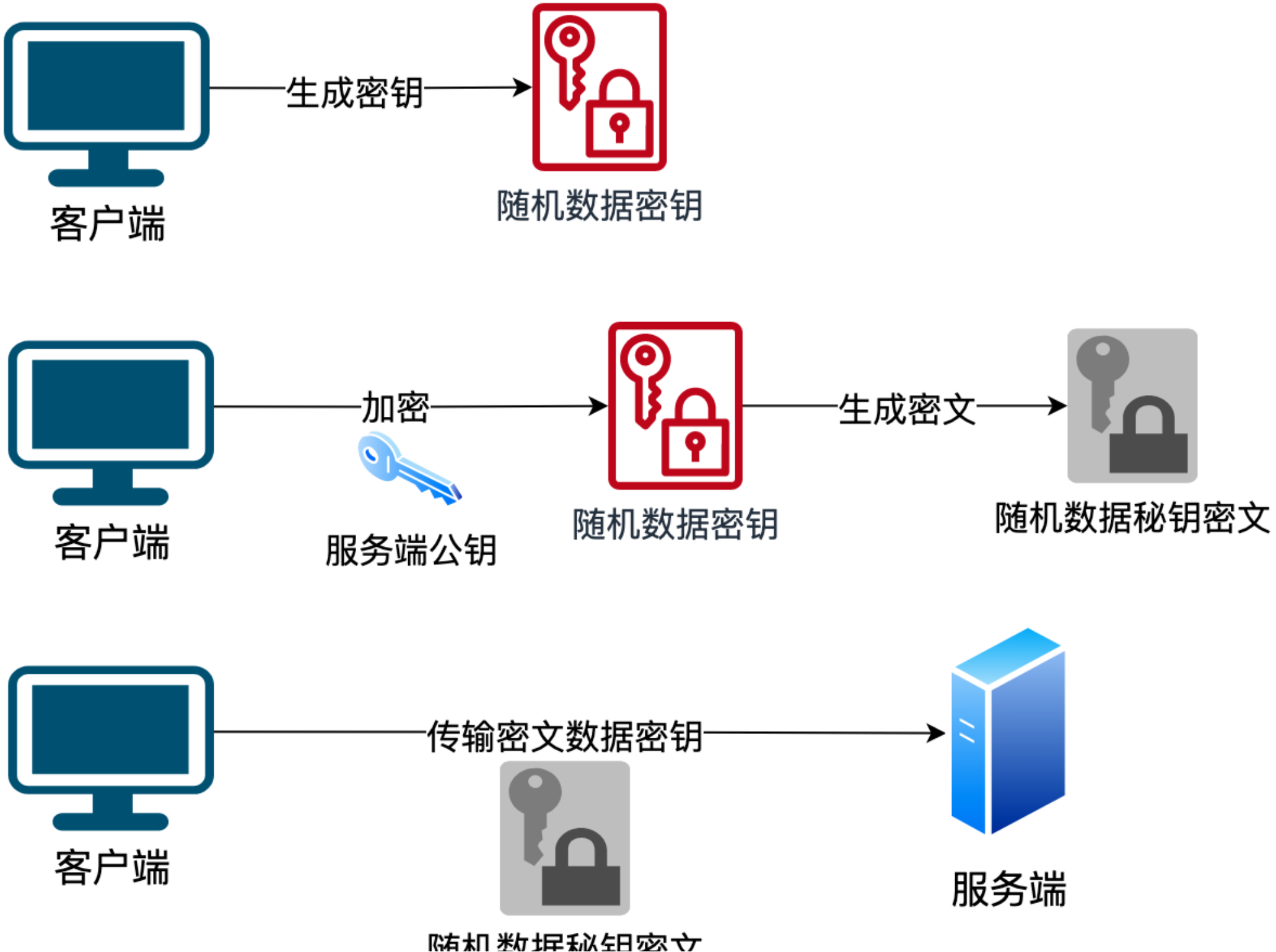
这里我们约定服务端总是权威的，其公钥是被所有客户端所感知的，客户端是任意的，没有身份上的要求，客户端总是主动发起于服务端的通信，并且服务端总是回复可信的数据给客户端。



客户端是密钥生成的决定方

在基于RSA的密钥交换体系中，总是由客户端来生成密钥。

这是因为，客户端总是能够获取到服务端的公钥，由客户端生成随机密钥，然后由服务端使用私钥解密，这样可以保证随机密钥的保密性。


客户端
生成密钥
随机数据密钥
客户端
加密
服务端公钥
随机数据密钥
生成密文
随机数据秘钥密文
客户端
传输密文数据密钥
随机数据秘钥密文
服务端

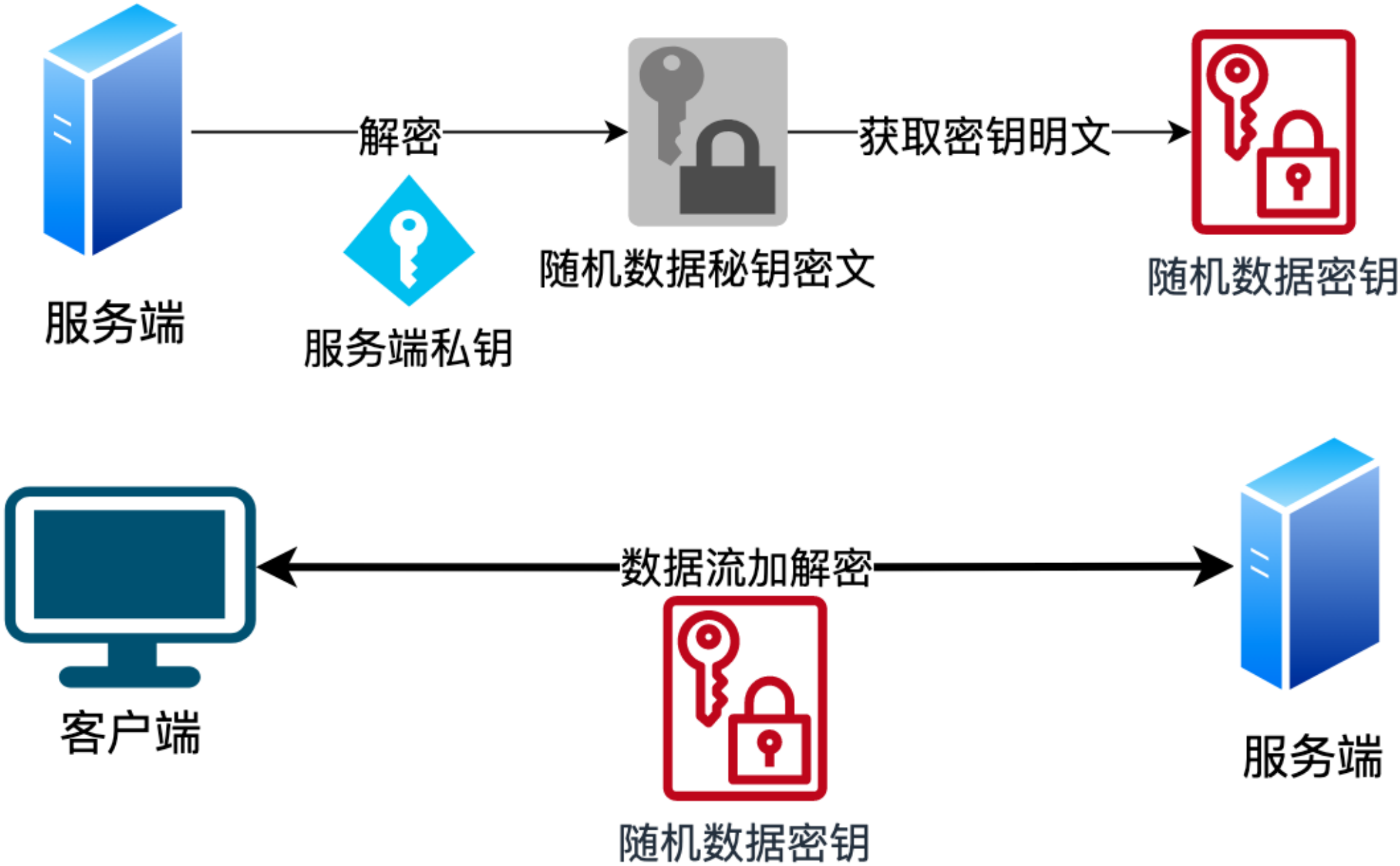
随机数据秘钥密文

服务端使用私钥获取随机密钥明文

私钥总是由服务端私密保存，绝对不会公开。

这是基于RSA进行数据安全加密的前提。

也是基于RSA进行密钥交换的基础。



服务端在使用私钥对随机密钥密文解密后便默认承认了与客户端使用的是同一把随机密钥。

随后的业务流数据便使用这把随机密钥进行数据通信。

针对RSA密钥交换的中间人攻击

客户端无法区分公钥来源

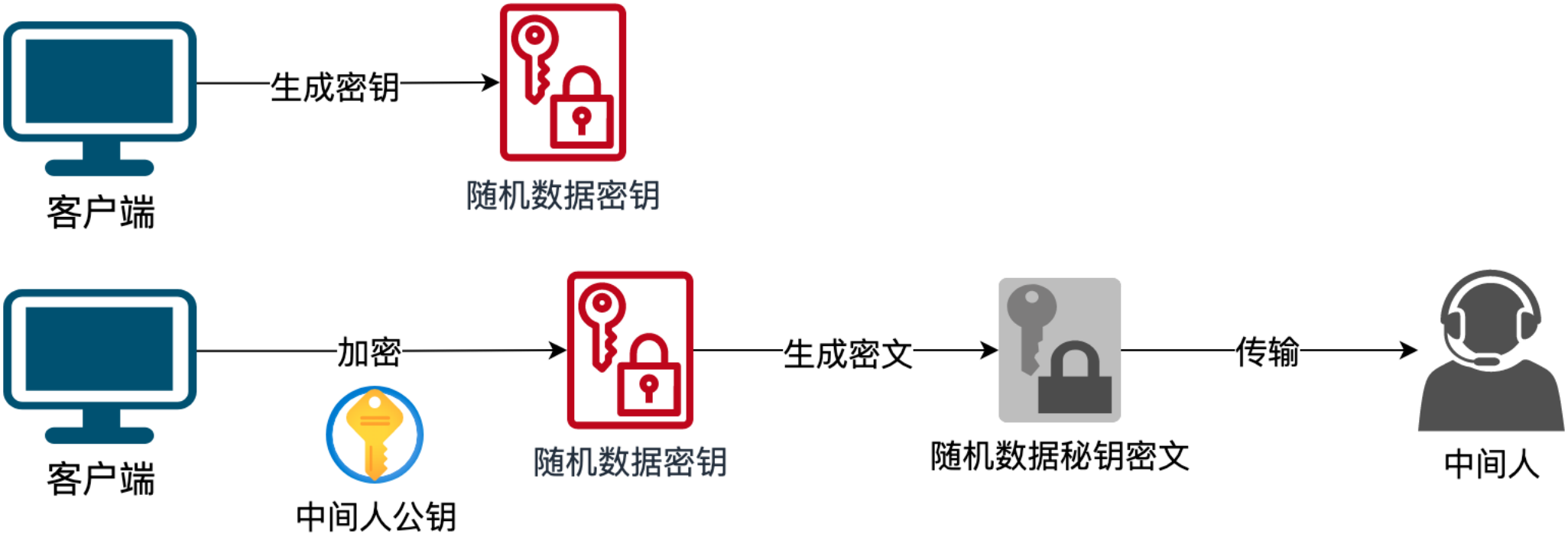
客户端是无法区分公钥来源的，这是RSA可能被中间人攻击的前提。

客户端只能够使用公钥对随机密钥进行加密，但是，这份公钥究竟是属于真实服务端，还是属于中间人的，客户端自己无法区分。

中间人对客户端的拦截

一旦客户端接受了中间人的公钥，那么就意味着客户端默认了中间人是真实的服务端。

那么在前面我们描述的数据密钥的交换过程，就可以被中间人完全监听甚至是篡改。



客户端在生成随机密钥并加密后，其消息会被中间人全部监听，由于客户端使用的是中间人的公钥，因此即使随机密钥被加密，中间人还是可以完全获取其明文。

中间人对服务端的迷惑

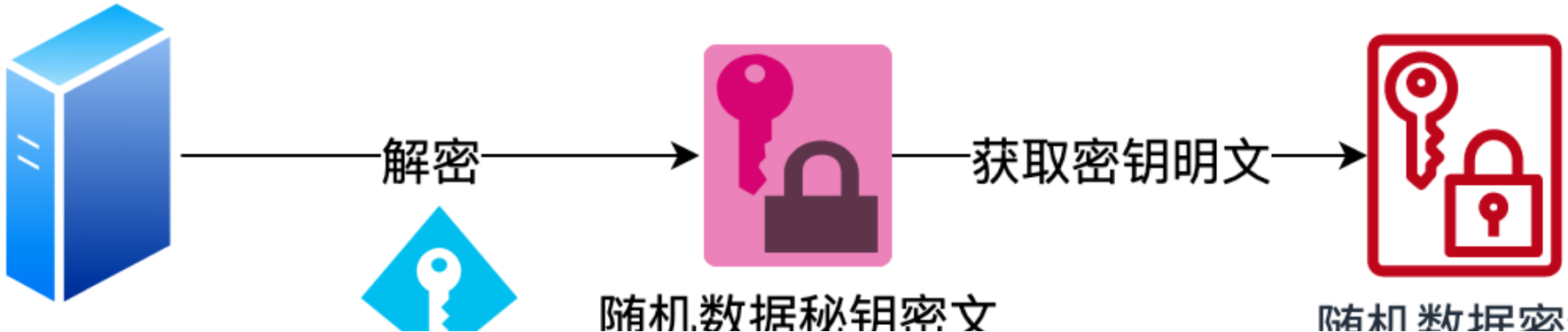
中间人同样可以对服务端进行迷惑。

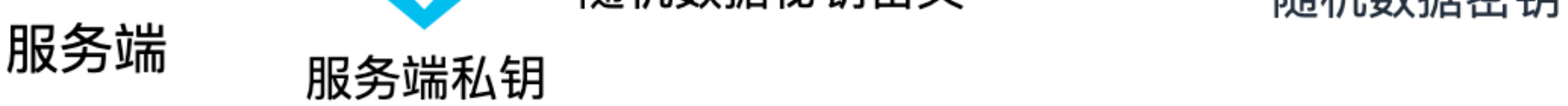
比如，在使用自己的私钥对随机密钥解密后，再试用服务端公钥对随机密钥加密，然后发送给服务端。

此时的服务端，其实也无法区分，发送数据过来的，究竟是客户端还是中间人。


解密
获取密钥明文
中间人
随机数据秘钥密文
随机数据密钥
中间人私钥


加密
生成密文
中间人
随机数据密钥
随机数据秘钥密文
服务端公钥


解密
获取密钥明文
随机数据秘钥密文

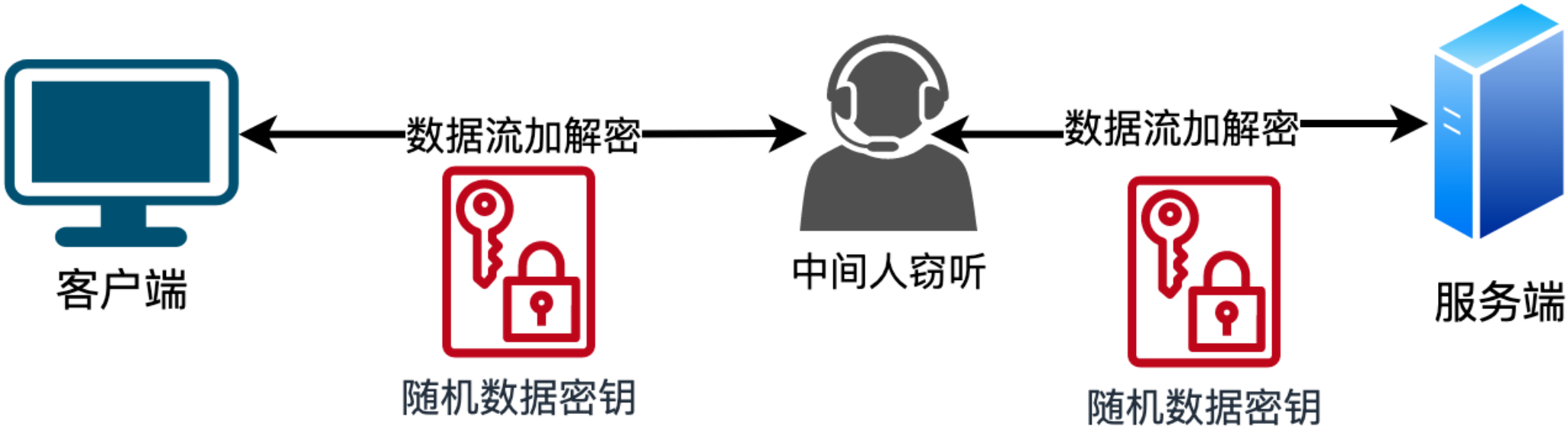


中间人对消息进行监听

在RSA密钥交换过程中，中间人需要保证：

1. 截取客户端的真实数据密钥
2. 迷惑服务端，使其相信中间人投递的数据密钥就是客户端产生的真实数据密钥

完成上述两个步骤后，中间人就可以进行双向的消息监听甚至篡改。

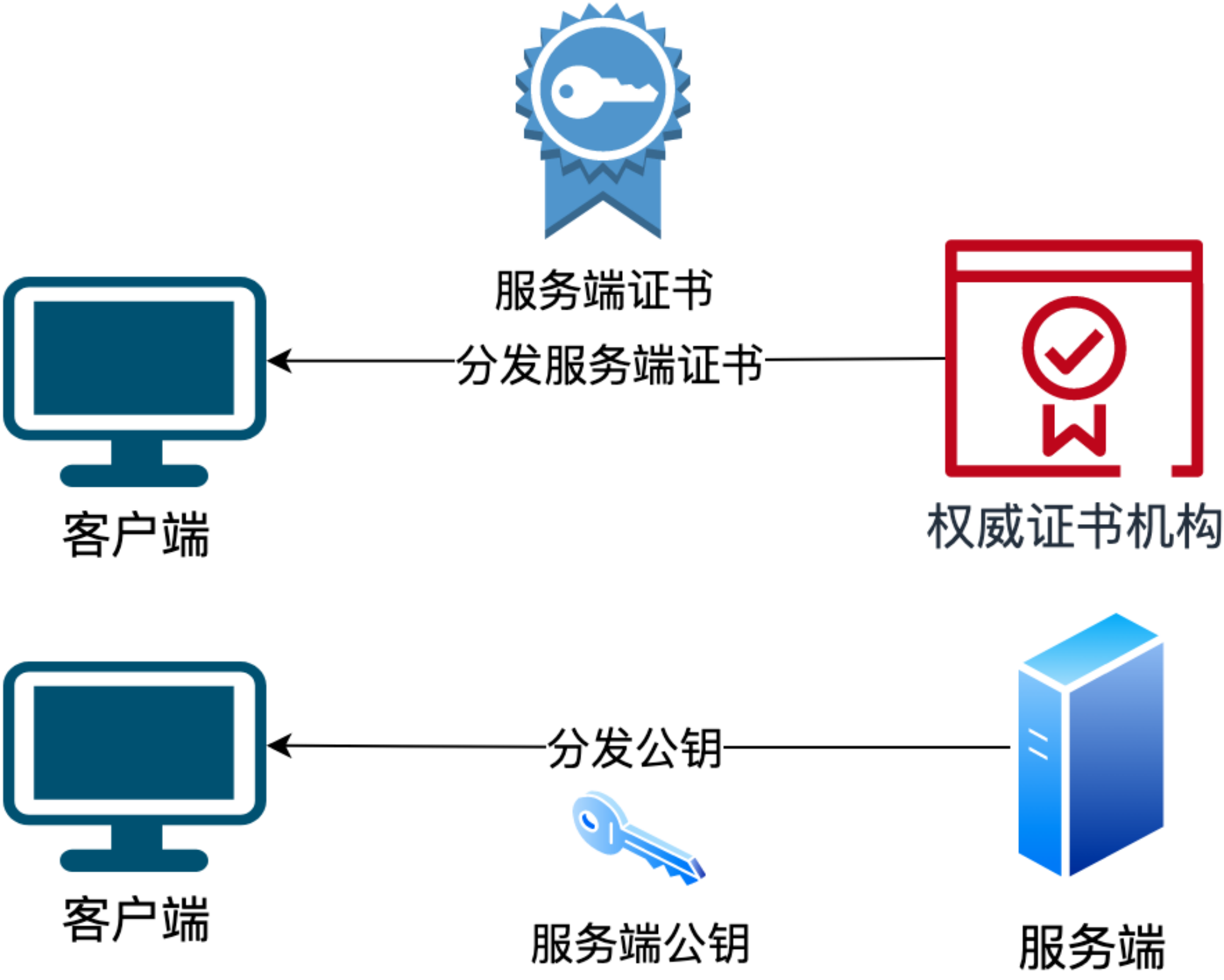


避免RSA的中间人攻击

防止中间人攻击的方法实际上就是身份证认证方式，目前主流方式就是数字签名的方式。

在前面的文章《非对称密钥沉思系列（3）：公钥、签名与证书》中我们聊了证书与身份认证的一些底层逻辑。

在RSA的这种密钥交换过程中，同样可以很好地应用证书来进行身份的鉴别与认证。


服务端证书
分发服务端证书
客户端
权威证书机构
分发公钥
客户端
服务端公钥
服务端

客户端

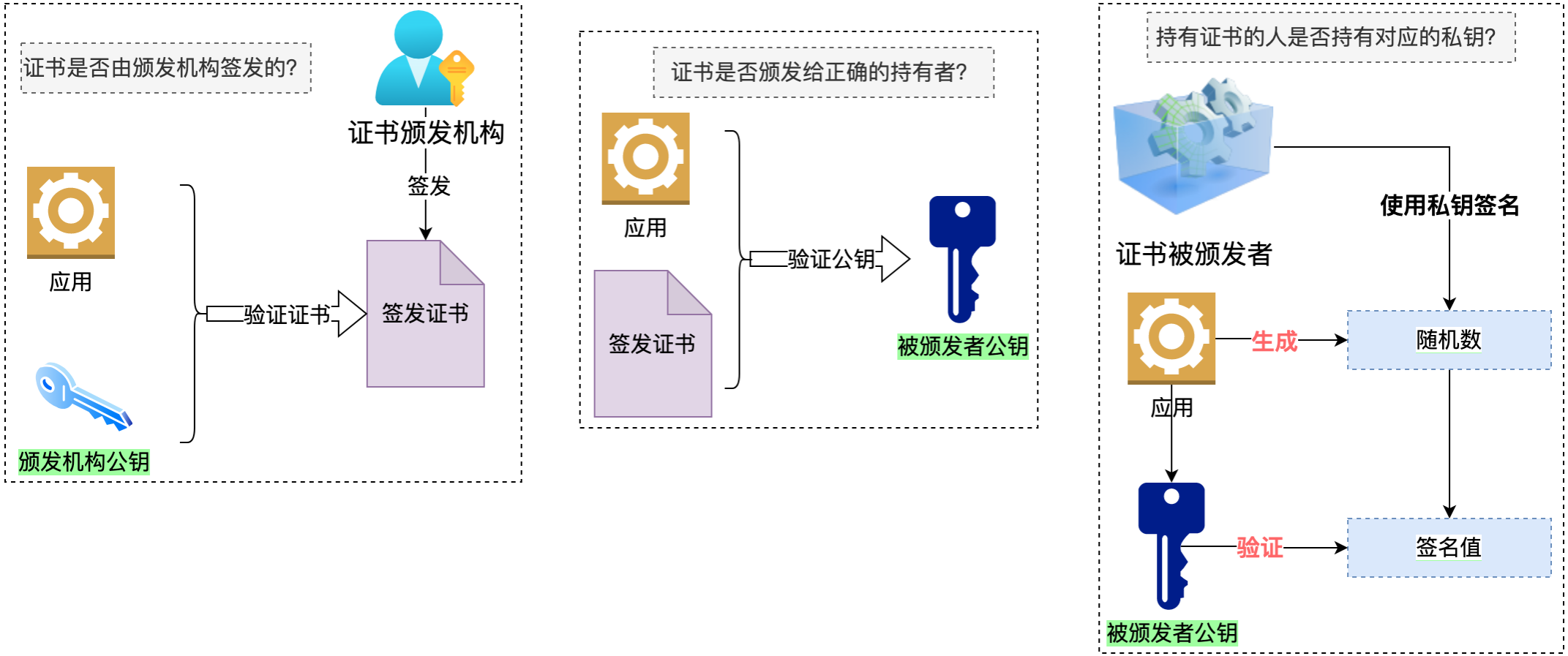
服务端证书

服务端

身份验证

服务端公钥

在对服务端的认证过程中，最重要的仍然是要解决三个问题：



这里细节就不赘述了，在前文中有详细描述，感兴趣的可以翻一下《非对称密钥沉思系列（3）：公钥、签名与证书》。

关于RSA交换的更多思考

在前面的描述中，我们其实只是着重的描述了客户端使用公钥加密随机密钥，然后服务端使用私钥对随机密钥解密的过程。其实结合《非对称密钥沉思系列（2）：聊聊RSA与数字签名》中的内容，我们在做随机密钥的交换时，还可以结合对随机密钥的HMAC等手段，保证随机密钥的不被篡改。这里感兴趣的同学可以自己思考下。

密钥交换协议DH

前面我们聊了很多RSA，但其实，RSA更侧重于非对称密钥算法，主要功能其实还是在于加密与解密。

而密钥交换协议DH，是专门用于协商密钥生成的。

RSA可以用来传输信息，DH更适合用来协商密钥。

DH算法解决了密钥在双方不直接传递密钥的情况下完成密钥交换，这个神奇的交换原理完全由数学理论支持。

由于DH系列算法涉及比较复杂的数学推理运算，这里不做过多展开讲解，感兴趣的同学可以翻阅相关RFC文档等。

最原始的DH算法并不能对抗MIMT（Man-in-the-middle attack），所以一般需要配合签名技术，不配合签名技术的DH称为，DH-ANON；

配合RSA签名的称为，DH-RSA；

配合DSA签名的称为，DH-DSA；

配合ECDSA签名的称为，DH-ECDSA；

总的来说，DH协议也怕中间人攻击，一般来说，也需要配置数字证书来进行身份的认证。

关于Forward security前向保密

Forward security前向保密，最初用来定义绘画密钥交换协议的一种安全性。即使长期密钥已经泄露，也不会影响之前的会话密钥的泄露，也就不会暴露之前的会话内容。

DH和ECDH算法为了实现前向安全，变种加入了另一个随机变量ephemeral key得到新的算法DHE、ECDHE。

RSA、DH和DSA都是基于整数有限域离散对数来实现。

ECC和ECDH都是基于椭圆曲线的离散对数难题来实现的。

现在实际使用中，优先选择 ECDHE>DHE> DH,RSA ...等。

python库中关于DH协议使用的示例

```
from typing import Tuple

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.asymmetric.dh import DHParameters, DHPrivateKey, DHPublicKey
from cryptography.hazmat.primitives.kdf.hkdf import HKDF


def data_key_exchange(owner_pri_key: DHPrivateKey, peer_pub_key: DHPublicKey) -> bytes:
    shared_key = owner_pri_key.exchange(peer_pub_key)
    derived_key = HKDF(algorithm = hashes.SHA256(),

```

```
                        length = 32,
                        salt = None,
                        info = b'handshake data',
                        backend = default_backend()).derive(shared_key)
    print("derived_key:{}, length:{}".format(list(derived_key), len(derived_key)))
    return derived_key


def generate_common_parameters() -> DHParameters:
    parameters = dh.generate_parameters(generator = 2, key_size = 2048,
                                        backend = default_backend())
    return parameters


def generate_dh_key(parameters: DHParameters) -> Tuple[DHPrivateKey, DHPublicKey]:
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()
    return private_key, public_key


if __name__ == '__main__':
    param = generate_common_parameters()
    a_pri_key, a_pub_key = generate_dh_key(param)
    b_pri_key, b_pub_key = generate_dh_key(param)
    a_data_key = data_key_exchange(a_pri_key, b_pub_key)
    b_data_key = data_key_exchange(b_pri_key, a_pub_key)
    print(a_data_key == b_data_key)
```

最终生成的密钥数据：

```
derived_key:[3, 139, 194, 229, 249, 124, 131, 14, 141, 172, 168, 29, 26, 152, 63, 253, 227, 234, 189, 101, 130, 192, 139, 99, 50, 8, 153, 75, 31
derived_key:[3, 139, 194, 229, 249, 124, 131, 14, 141, 172, 168, 29, 26, 152, 63, 253, 227, 234, 189, 101, 130, 192, 139, 99, 50, 8, 153, 75, 31
True
```


数据安全　密钥管理系统　#RSA　#密钥协商　#中间人攻击　#DH　#密钥交换