第7讲

认证

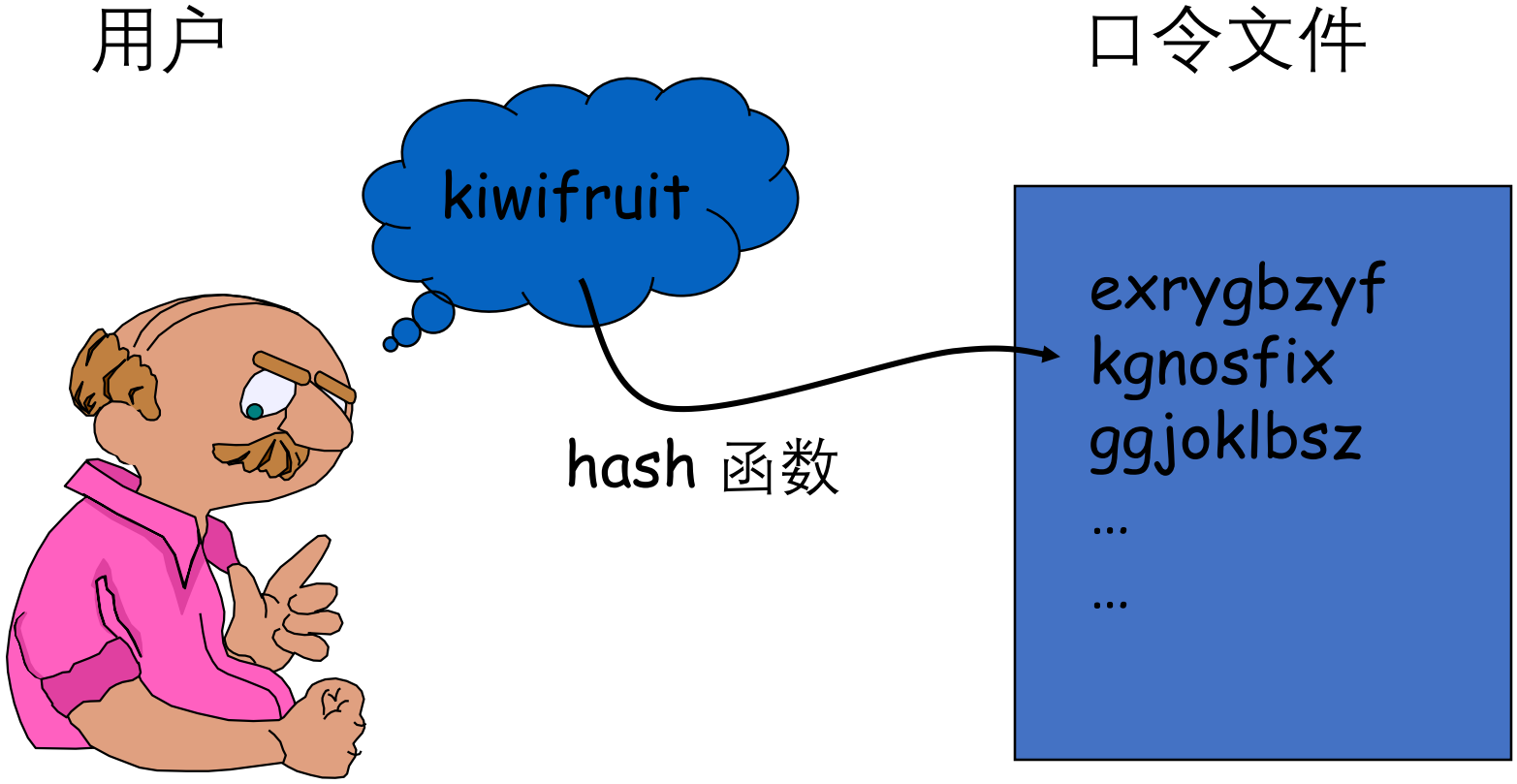
大纲

- 用户认证
  - 口令认证，Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介（KPC和CA）

口令认证

- 基本概念
  - 用户有一个秘密的口令
  - 系统检查口令以验证用户身份
- 问题
  - 口令如何存储?
  - 系统如何检查口令?
  - 猜测口令有多容易?
    - 口令文件难以保密
    - 更好的方法：即使拥有口令文件，也很难猜出口令

基本的口令方案



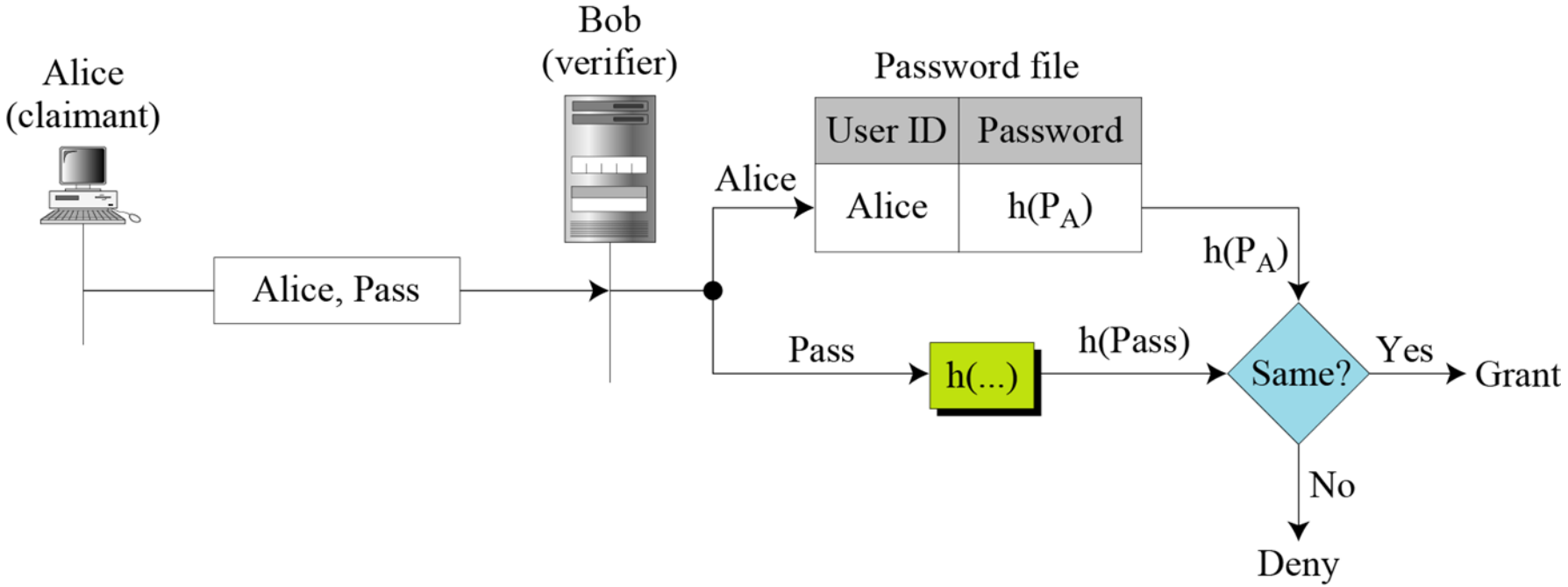
基本的口令方案

- Hash 函数 h : strings → strings
  - 给定 h(password), 难以找到 password
  - 没有比试错法更好的已知算法
- 用户口令存储为 h(password)
- 当用户输入口令时
  - 系统计算 h(password)
  - 与口令文件中的条目进行比较
- 口令本身没有被存储在磁盘上

口令认证

P_A: Alice's stored password

Pass: Password sent by claimant



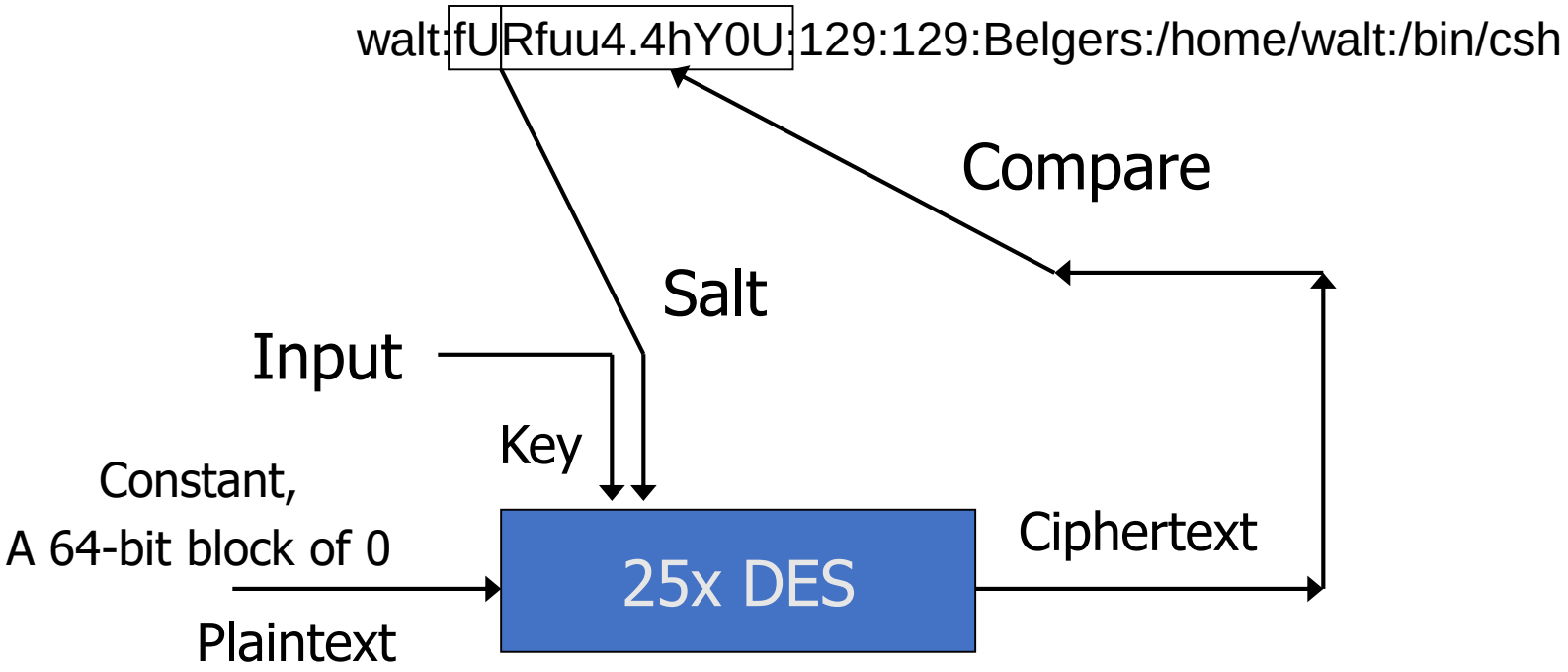
Unix 口令系统

- Hash 函数为 25xDES
  - 25 轮 DES 变体加密
- 任何用户都可以尝试“字典攻击”

R.H. Morris and K. Thompson, Password security: a case history, Communications of the ACM, November 1979

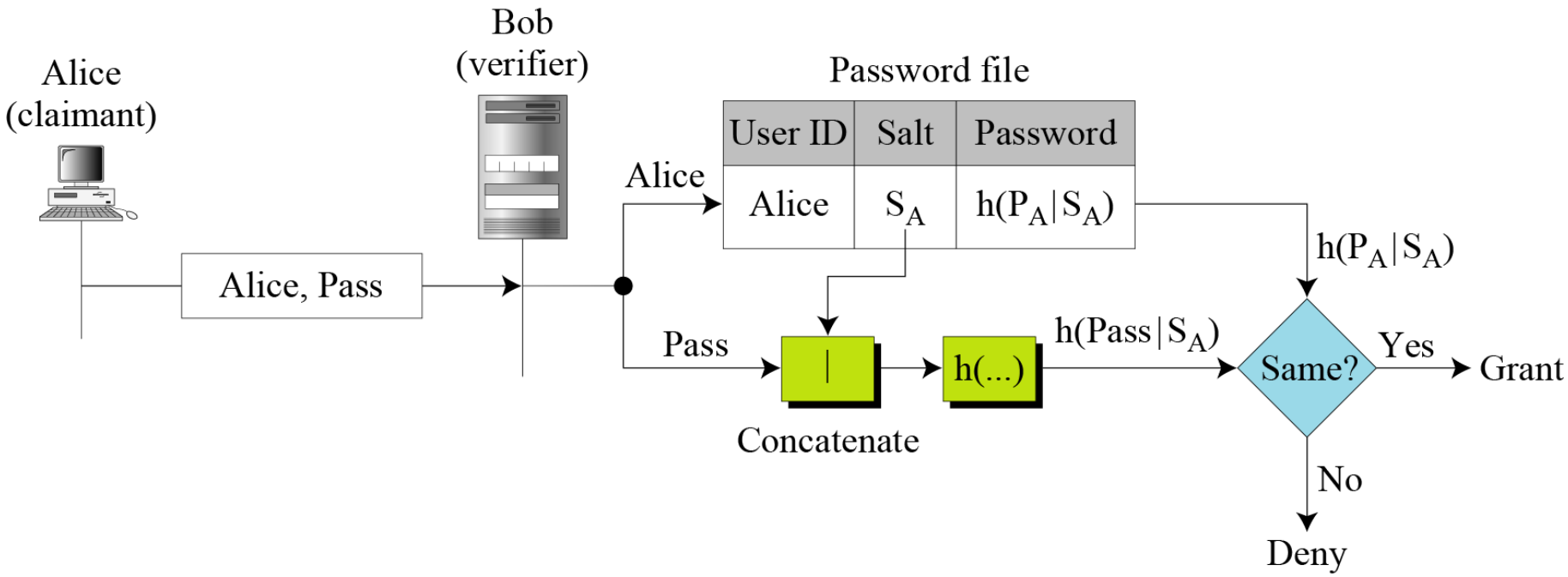
Unix 口令系统

- 口令行



口令认证

P_A: Alice's password
S_A: Alice's salt
Pass: Password sent by claimant



为什么需要Salt?

salt的优点

- 没有salt
  - 所有机器都具有相同的散列函数
    - 计算所有常用字符串的哈希值一次
    - 比较哈希文件和所有已知的口令文件
- 有salt
  - 一个口令散列了2^{12}种不同的方式
    - 预计算散列文件?
      - 需要更大的文件来覆盖所有常见的字符串
    - 对已知口令文件的字典攻击
      - 需要对于文件中的每个salt，尝试使用所有常见字符串

字典攻击(Dictionary Attack)

- 典型的口令字典
  - 1,000,000条通用口令
    - 人们的名字，普通的宠物名字和普通的词汇
  - 假设你每秒生成并分析10次猜测
    - 这对于一个网站来说可能是合理的; 离线速度要快得多
  - 最多100,000秒的字典攻击= 28小时，平均14小时
- 如果口令是随机的
  - 假设有六个字符的口令
    - 大写和小写字母，数字，32个标点符号
    - 689,869,781,056个口令组合
    - 穷举搜索平均需要1,093年

口令认证

2011年12月21日，CSDN后台数据库被盗，由于明文存储，642万多个用户的帐号、口令等信息被泄露

服务器口令表

用　户	口 令
…	…
张　三	1a23
李　四	33z4
王　五	66a1
…	…

口令认证

2017年初，一个网络黑市商家兜售了10亿个被盗的中国网络巨头的用户账户；

近日，CosmicDark上的一个黑市商家刚上架了一个用户数据库，该数据库包含了100,759,591个被盗的优酷用户账户；

大纲

- 用户认证
  - 口令认证，Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介（KPC和CA）

挑战-应答认证

Goal: Bob 希望 Alice向他“证明”自己的身份

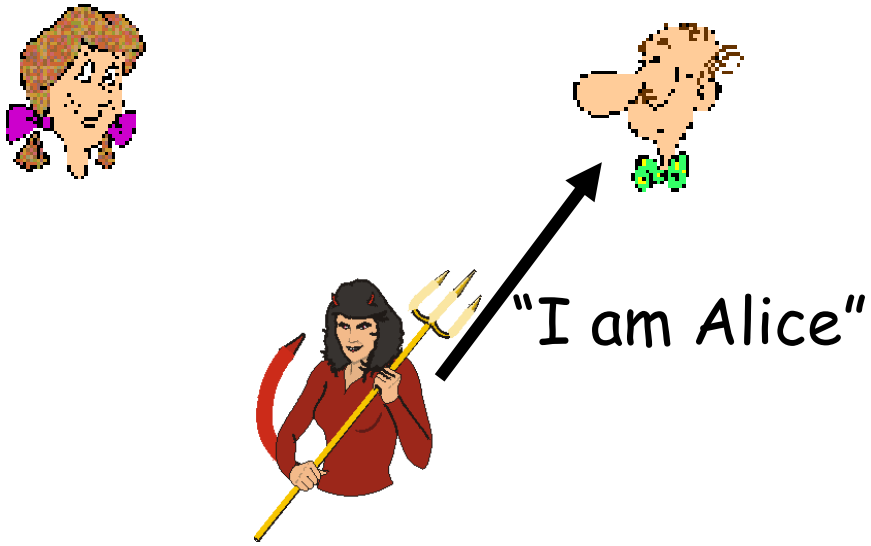
Protocol ap1.0: Alice 说 “I am Alice”

失败场景??

认证

Goal: Bob 希望 Alice向他“证明”自己的身份

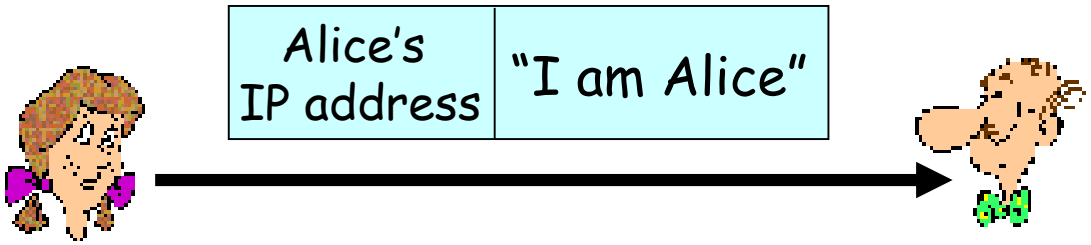
Protocol ap1.0: Alice 说“I am Alice”



在网络环境中，Bob看不到Alice，所以Trudy 可以简单申明她是Alice

认证

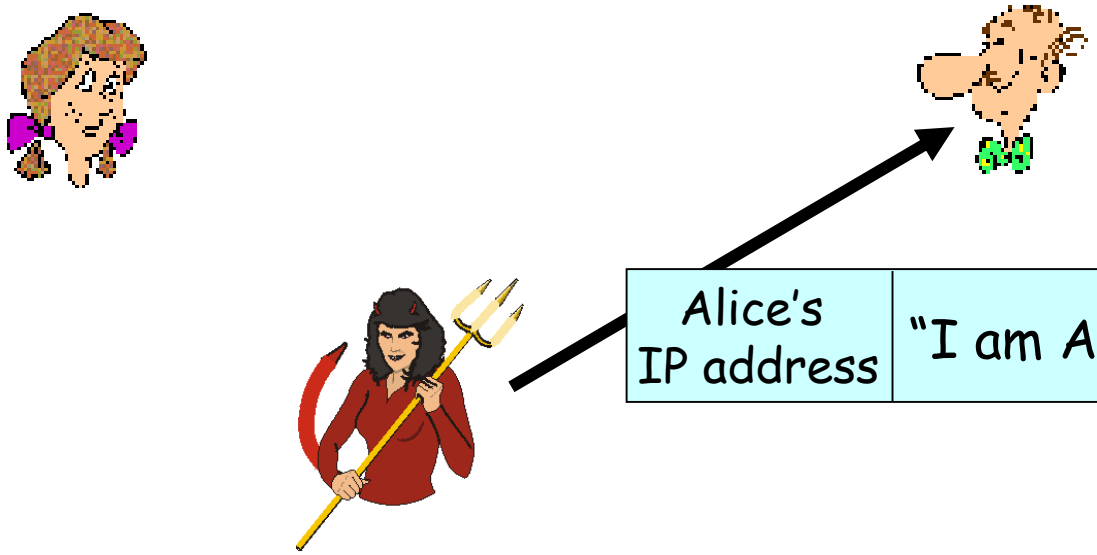
Protocol ap2.0: Alice 在包含她的IP地址的IP包中说"I am Alice"



失败场景??

认证

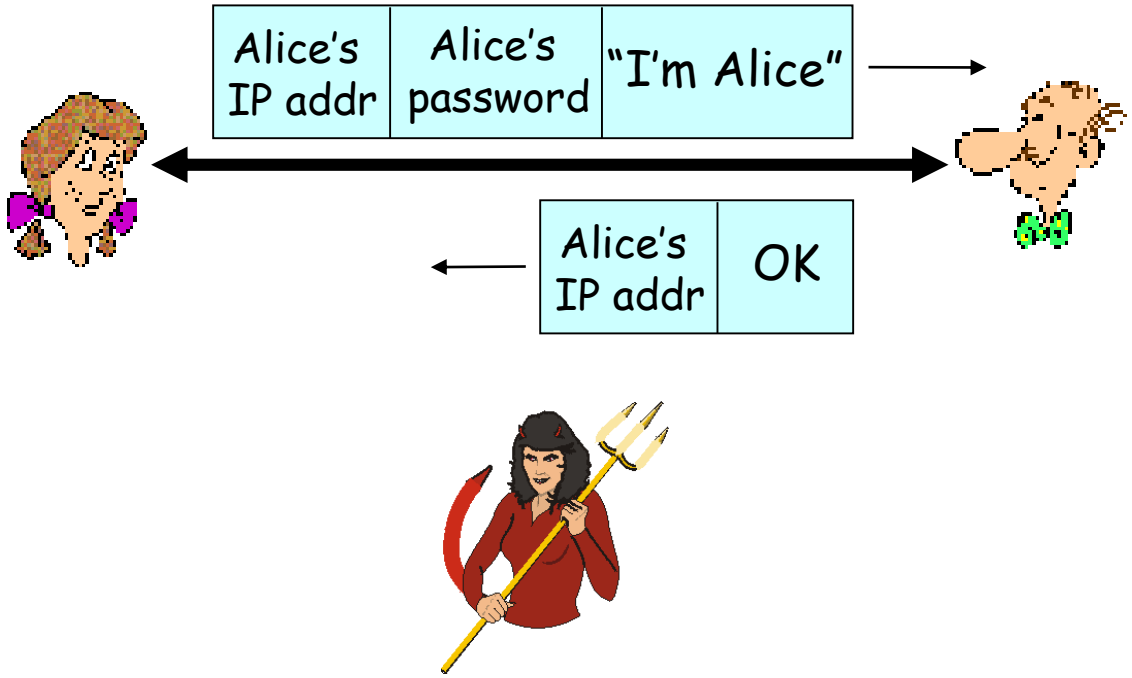
Protocol ap2.0: Alice 在包含她的IP地址的IP包中说"I am Alice"



Trudy 可以创建欺骗Alice的地址数据包

认证

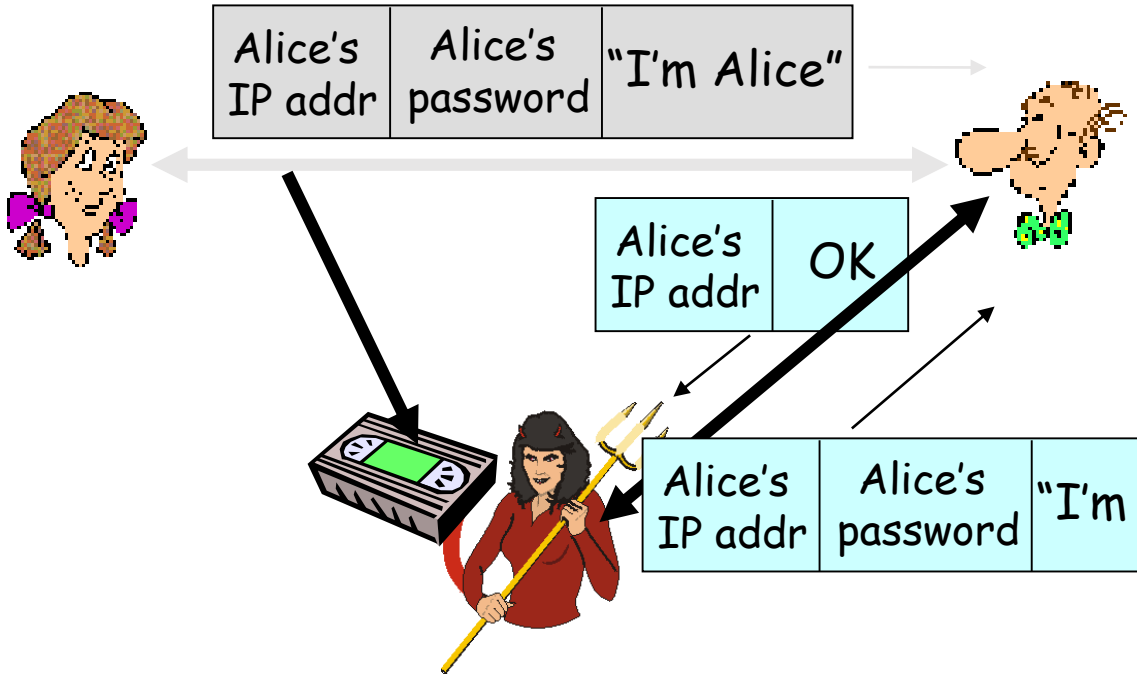
Protocol ap3.0: Alice 在包含她的IP地址的IP包中说 "I am Alice"，并且发送她的口令来证明



失败场景??

认证

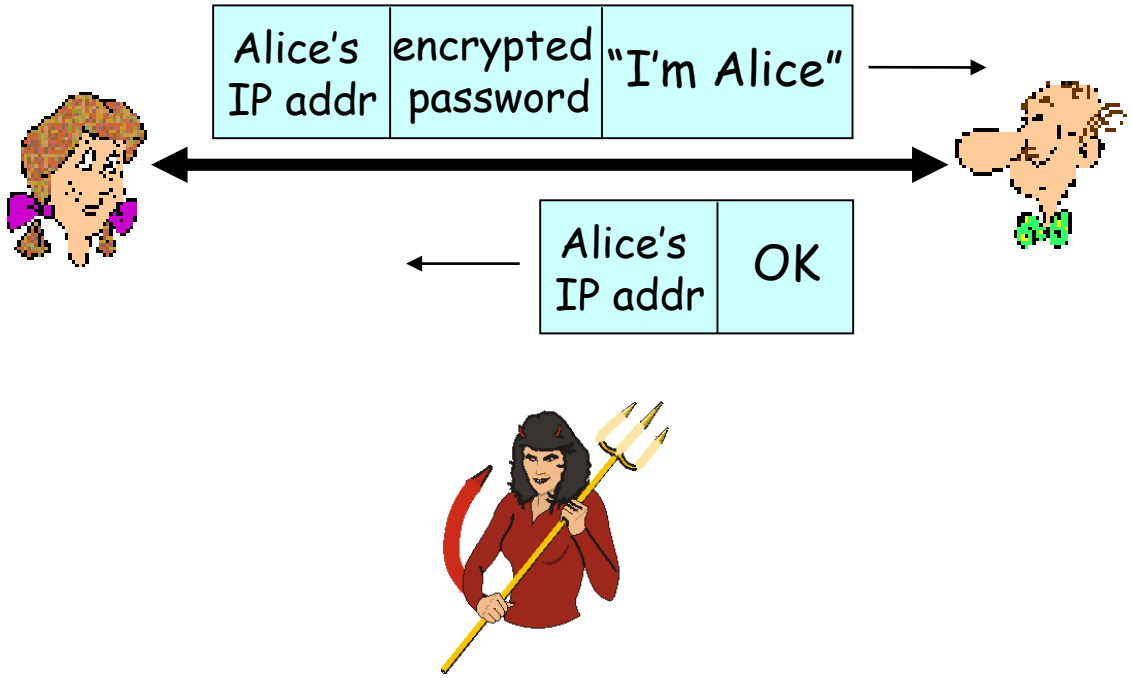
Protocol ap3.0: Alice 在包含她的IP地址的IP包中说 "I am Alice"，并且发送她的口令来证明



重放攻击(Playback Attack): Trudy 记录 Alice's数据包，并稍后将数据发送给Bob

认证

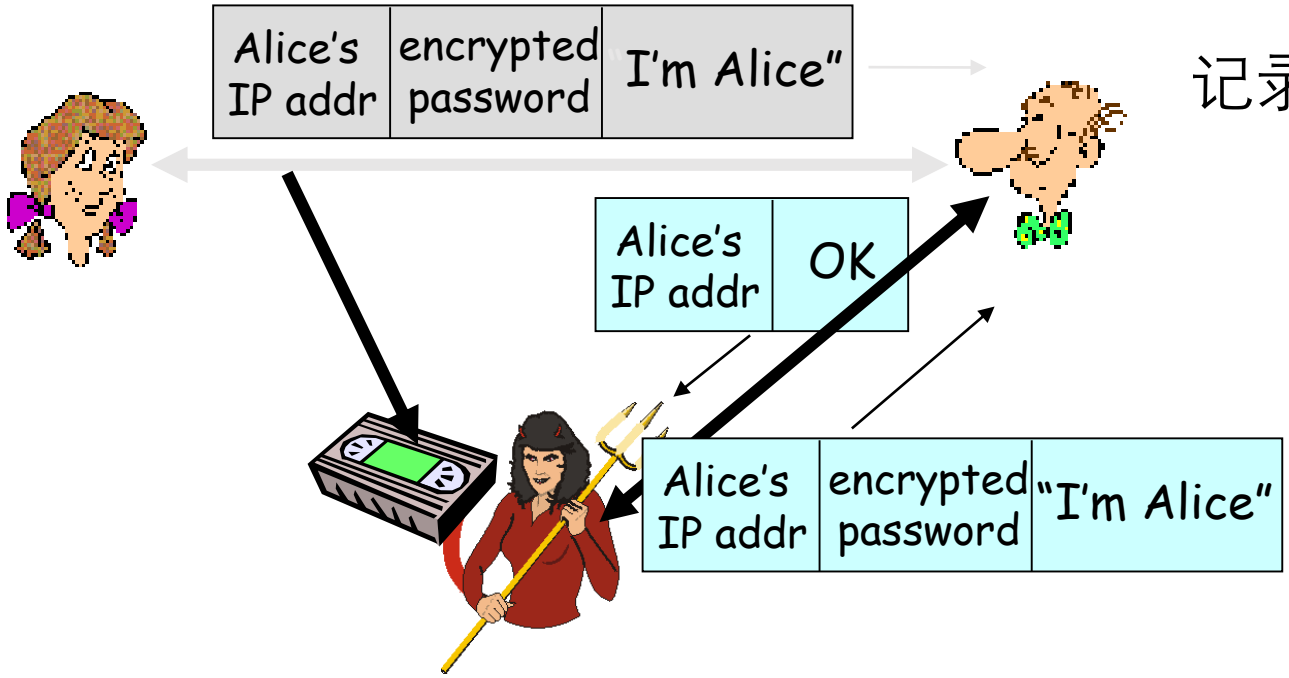
Protocol ap3.1: Alice 在包含她的IP地址的IP包中说 "I am Alice"，并且发送她的加密的口令来证明



失败场景??

认证

Protocol ap3.1: Alice 在包含她的IP地址的IP包中说 "I am Alice"，并且发送她的加密的口令来证明



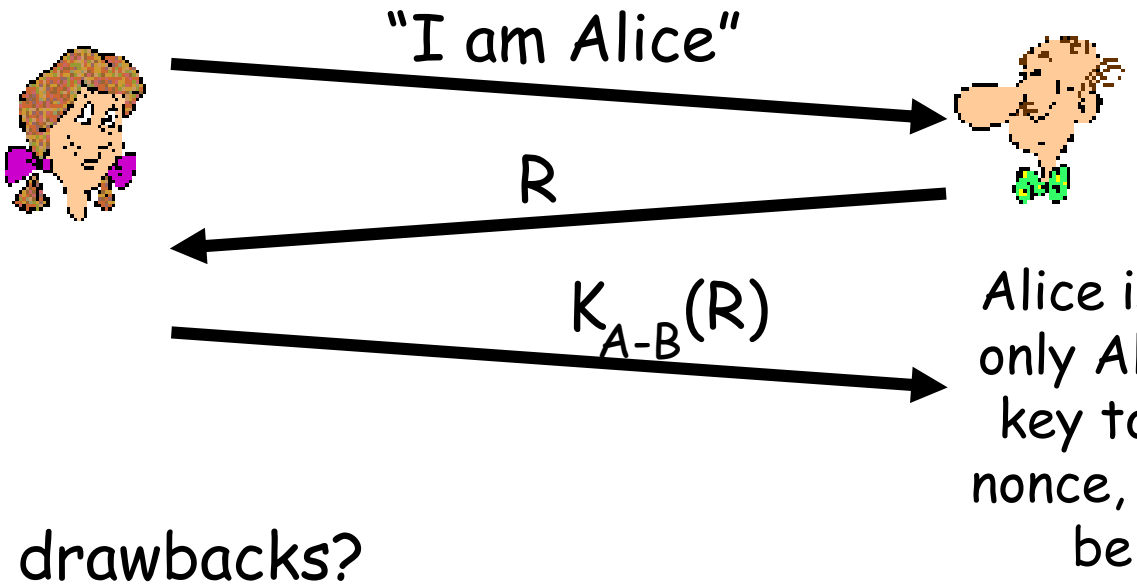
记录和重放仍然有效!

认证

目标: 避免重放攻击

Nonce: number (R) used only *once -in-a-lifetime*

ap4.0: 为证明Alice是"live"的, Bob 发送nonce (R)给Alice.
Alice必须返回用共享密钥加密的 R



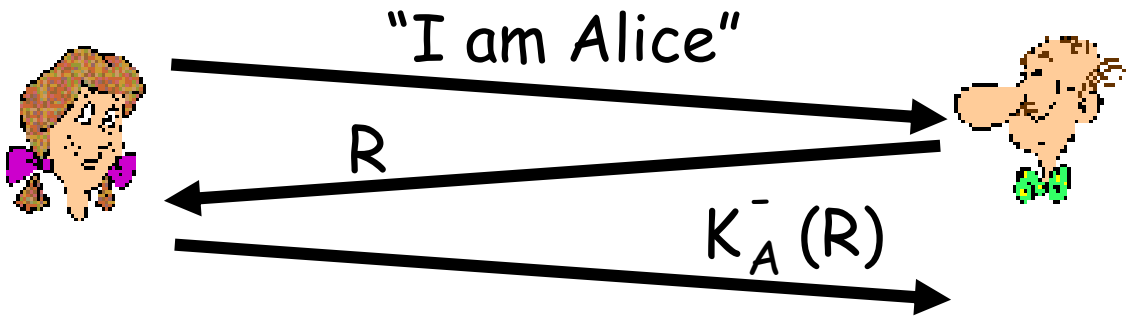
Failures, drawbacks?

认证

ap4.0 不保护服务器数据库读取攻击(server database reading attack)

- 使用公钥密码技术进行认证?

ap5.0: 使用公钥密码体制



Bob 计算

$K_A^+(K_A^-(R)) = R$

共享密钥解决重放问题，但是不能解决攻击者读取数据库的问题

Bob知道只有Alice有私钥，可以对R进行加密，这样：

$K_A^+(K_A^-(R)) = R$

静态口令

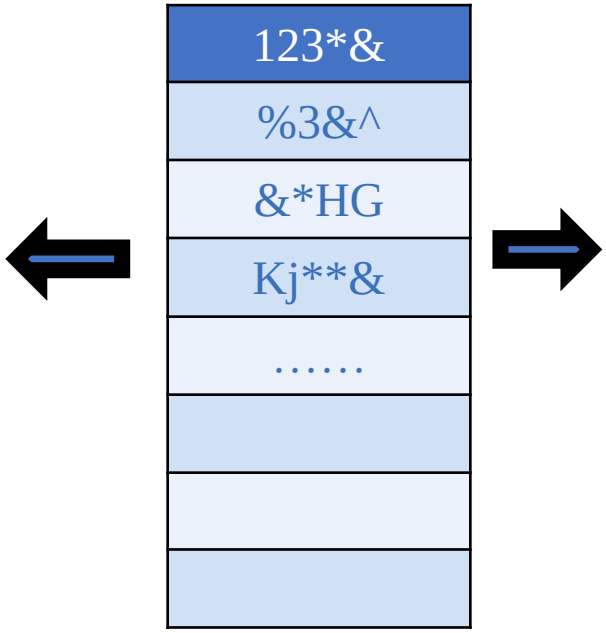
- 静态口令的缺点：
  - 通常使用的计算机口令是静态的，即在一定时间内是不变的，而且可重复使用。
  - 极易被网上嗅探劫持、重放攻击

动态口令

- 一次性口令技术
  - 20世纪80年代初，美国科学家Leslie Lamport首次提出了一次性口令（OTP）的思想，即用户每次登录系统时使用的口令是变化的。
  - 实现方法有多种
    - 一次性密码本
    - 发送口令短信
    - 挑战-应答方式
    - ……

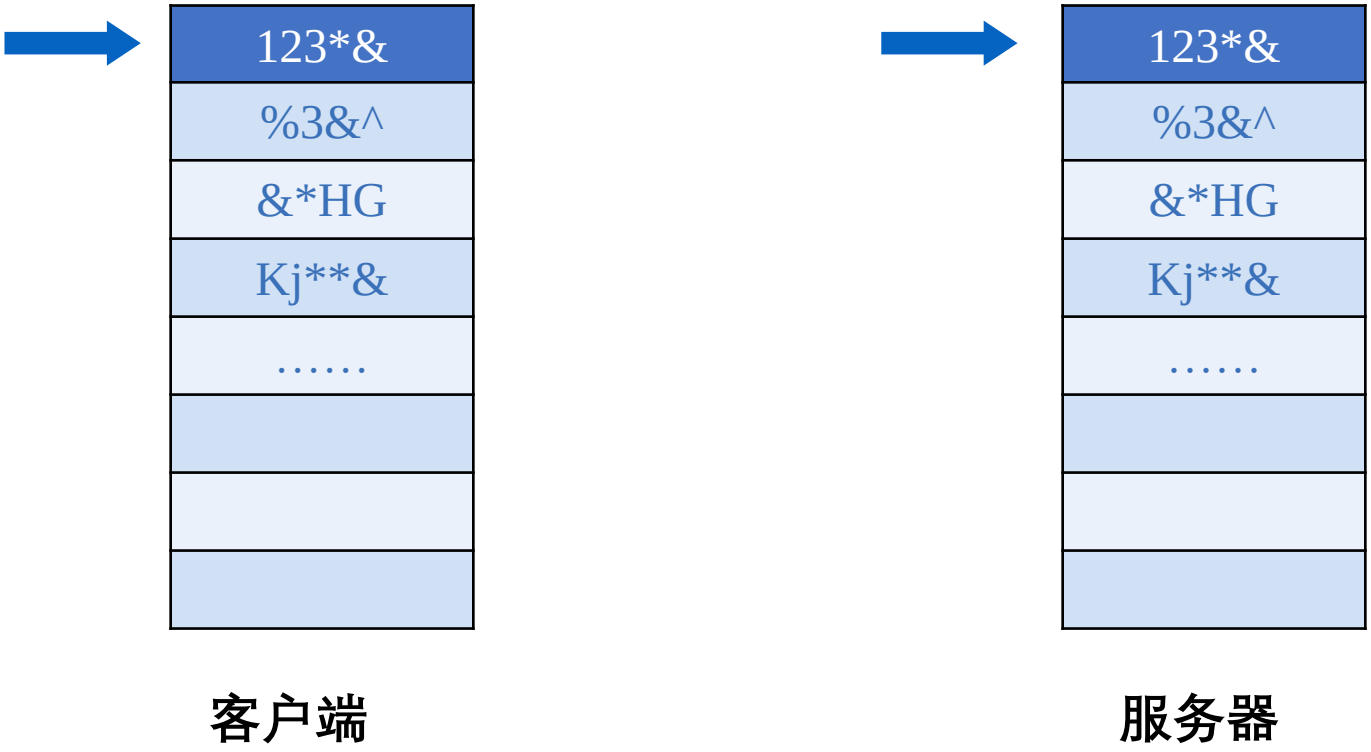
动态口令

- 一次性密码本



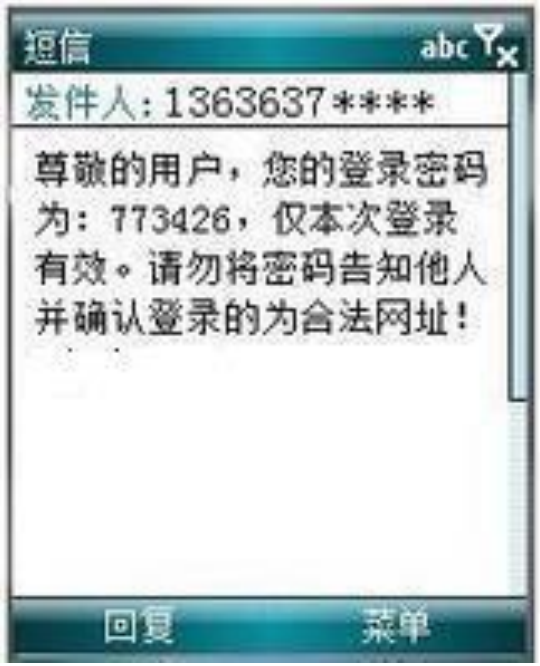
动态口令

- 一次性密码本



动态口令

- 通信双方事先约定口令更改方式

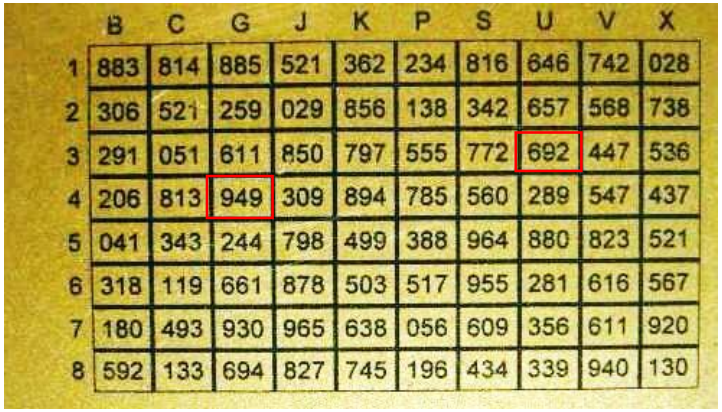


短信密码

动态口令牌

动态口令

- “挑战-应答”



	B	C	G	J	K	P	S	U	V	X
1	883	814	885	521	362	234	816	646	742	028
2	306	521	259	029	856	138	342	657	568	738
3	291	051	611	850	797	555	772	692	447	536
4	206	813	949	309	894	785	560	289	547	437
5	041	343	244	798	499	388	964	880	823	521
6	318	119	661	878	503	517	955	281	616	567
7	180	493	930	965	638	056	609	356	611	920
8	592	133	694	827	745	196	434	339	940	130

工行动态口令卡：在规定时间内输入U3G4

CAPTCHA

动态口令

- 动态口令（一次性口令）技术
  - 用户每次登录系统时使用的口令是变化的，不重复的

动态口令如何实现？

动态口令

动态口令实现思路：

口令产生

1、由算法(散列函数)产生，记为F

2、F算法的输入中应包含变动因子X

3、变动因子不能有重复

F(X,K)

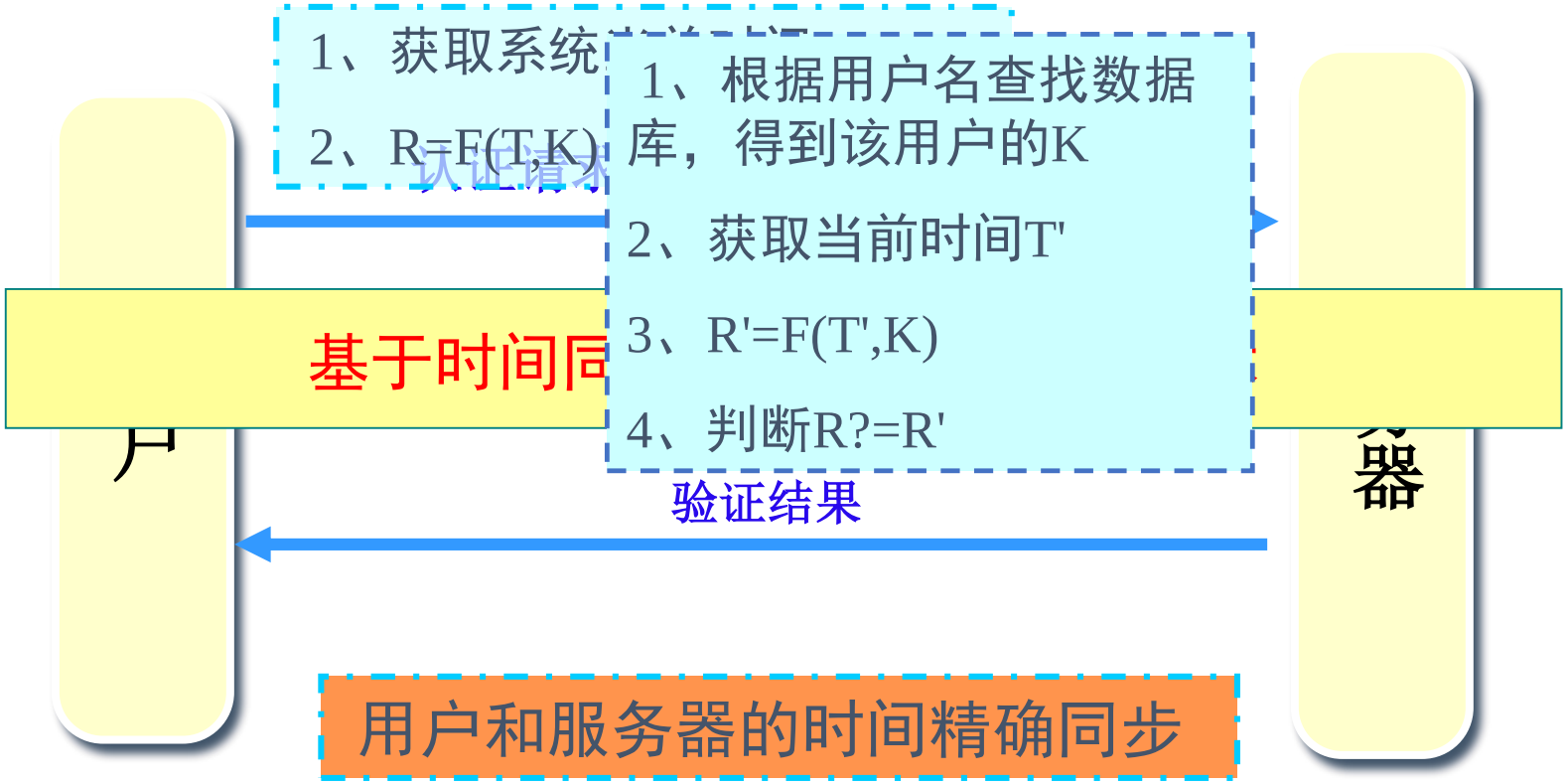
口令验证

1、通信双方的变动因子必须保持一致

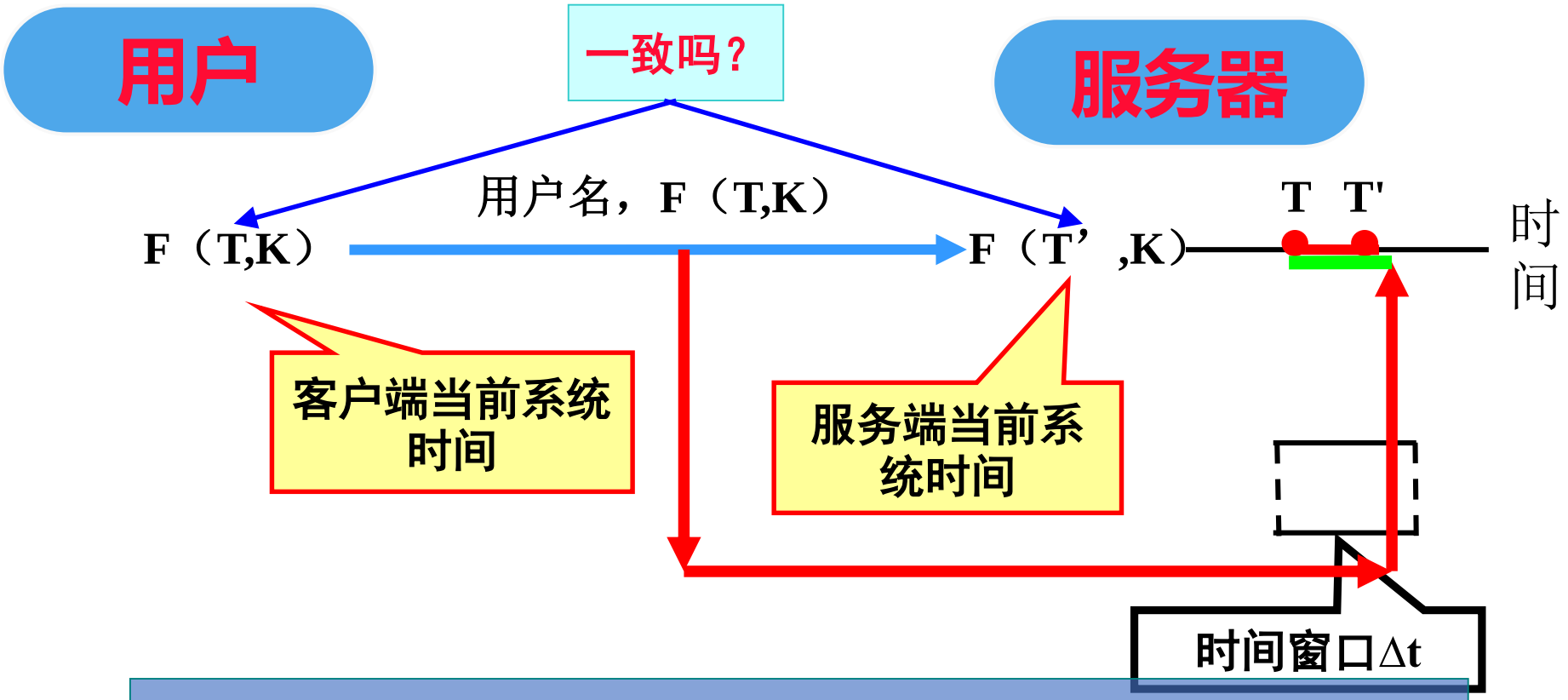
2、算法输入中包括通信双方共享的密钥K

动态口令

- 动态口令实现技术一
  - 通信双方事先共享密钥K,以登录时间作为变动因子



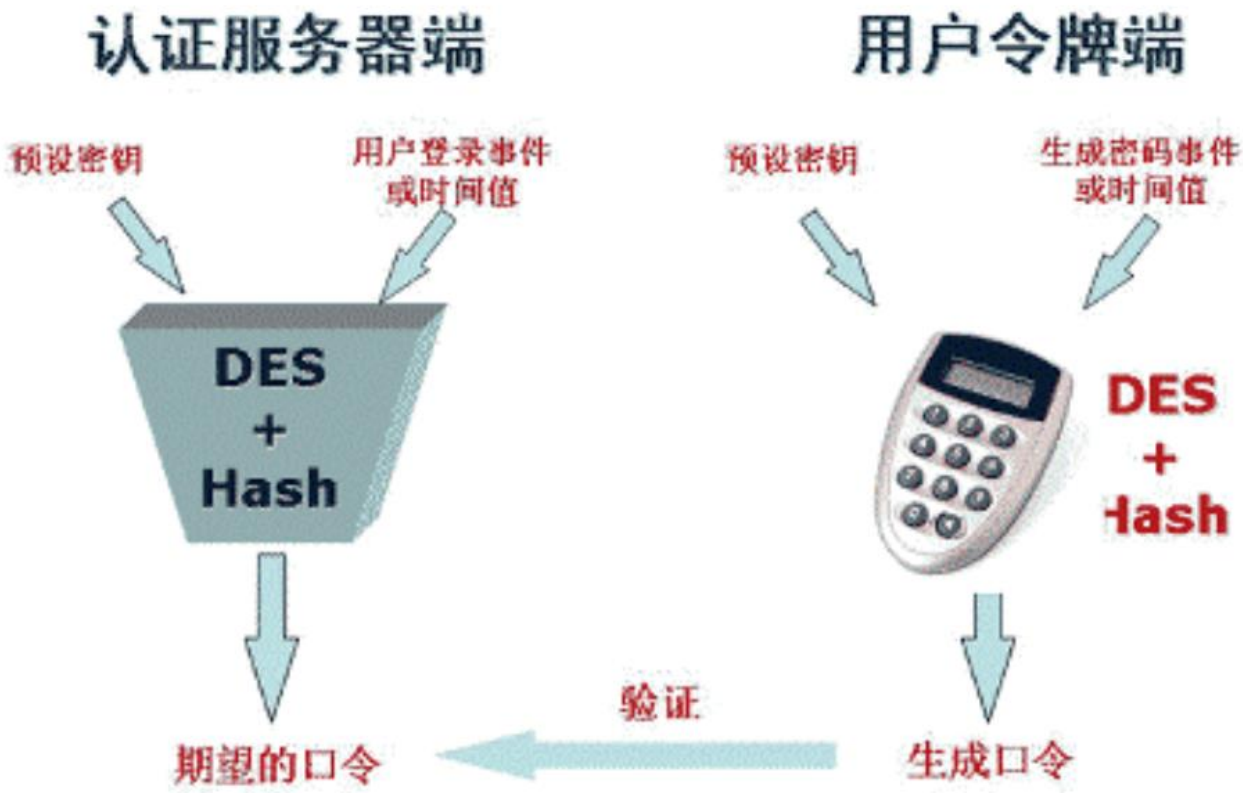
动态口令



1、由于网络时延，某些合法用户的身份无法验证

2、由于时间窗口的存在，无法完全避免重放攻击

动态口令

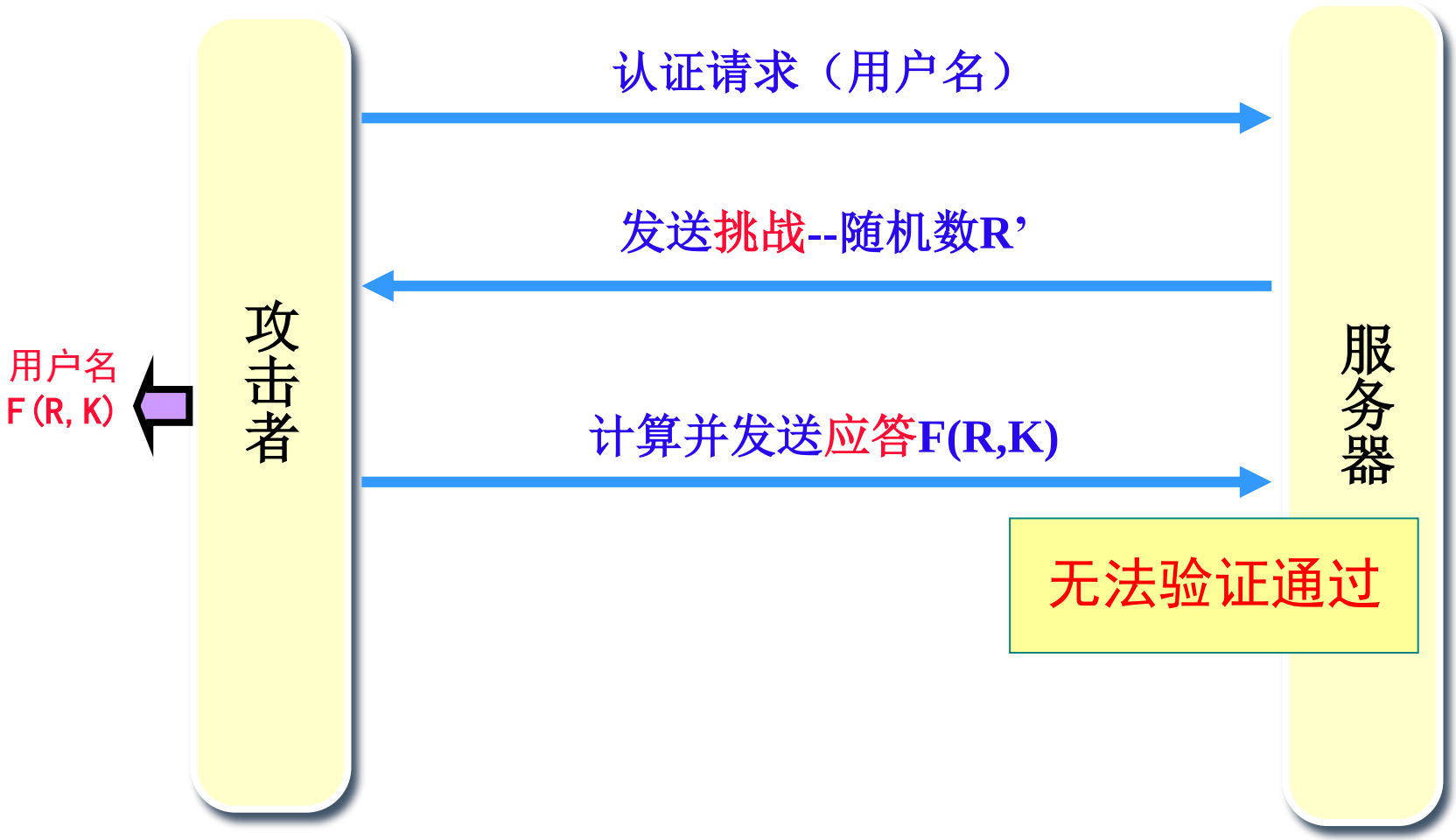

认证服务器端
用户令牌端
预设密钥
用户登录事件
或时间值
预设密钥
生成密码事件
或时间值
DES
+
Hash
DES
+
Hash
验证
期望的口令
生成口令

动态口令

- 动态口令实现方案二

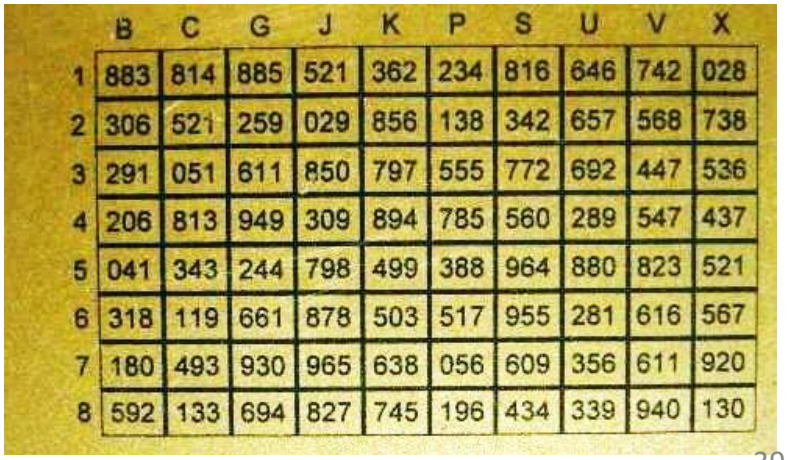
动态口令

- 动态口令实现方案二



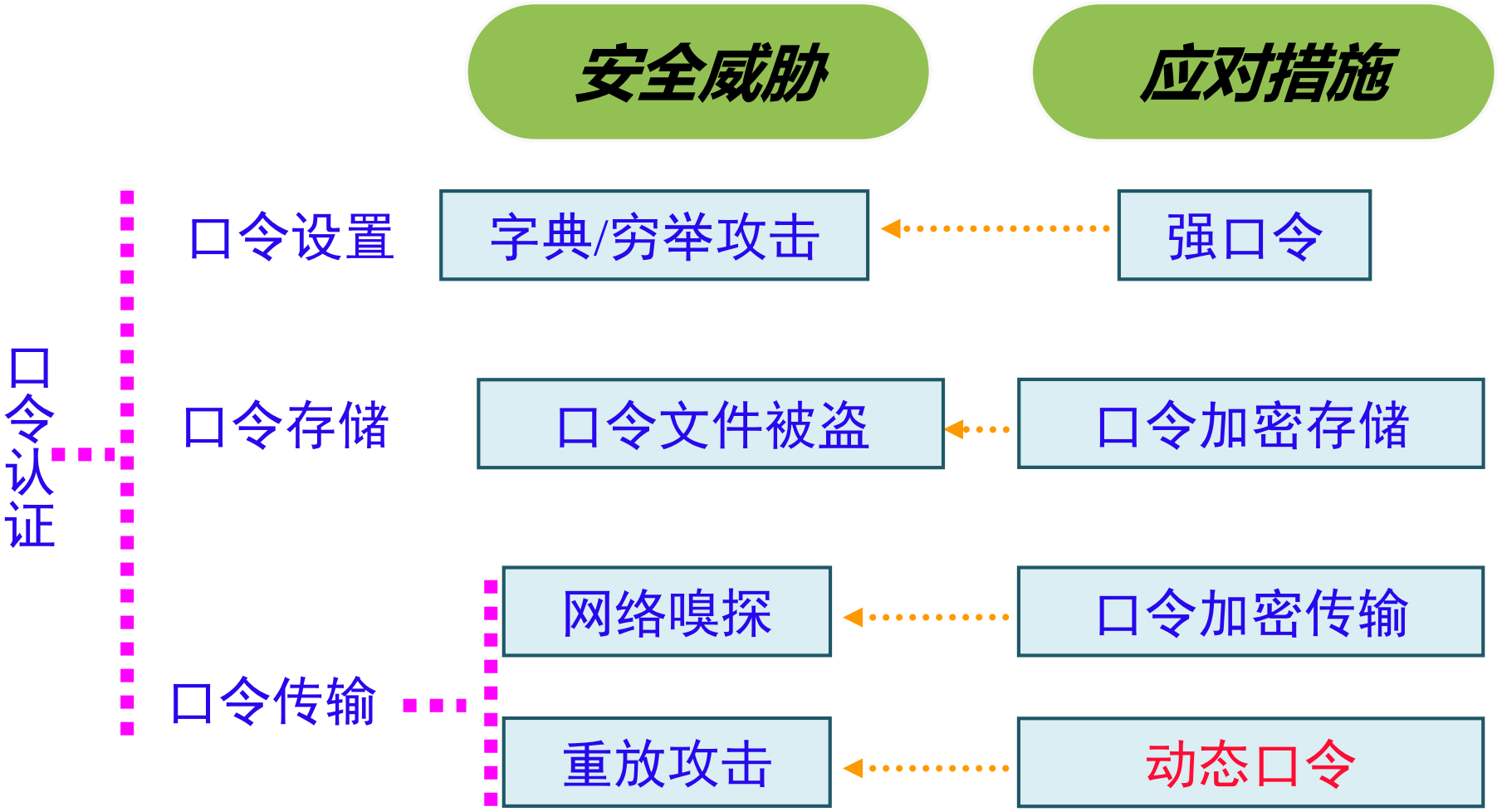
动态口令

- 动态口令（一次性口令）实现方法
  - 时间同步技术
    - 适用于网络性能稳定，验证效率高
  - 挑战-应答技术
    - 适用于分布式网络环境，安全性、可靠性高



	B	C	G	J	K	P	S	U	V	X
1	883	814	885	521	362	234	816	646	742	028
2	306	521	259	029	856	138	342	657	568	738
3	291	051	611	850	797	555	772	692	447	536
4	206	813	949	309	894	785	560	289	547	437
5	041	343	244	798	499	388	964	880	823	521
6	318	119	661	878	503	517	955	281	616	567
7	180	493	930	965	638	056	609	356	611	920
8	592	133	694	827	745	196	434	339	940	130

口令认证总结


安全威胁
应对措施
口令认证
口令设置
字典/穷举攻击
强口令
口令存储
口令文件被盗
口令加密存储
口令传输
网络嗅探
口令加密传输
重放攻击
动态口令

生物识别技术(Biometrics)

- 使用一个人的身体特征
  - 指纹，语音，脸部，键盘计时，…
- 优点
  - 无法泄露、丢失、遗忘
- 缺点
  - 成本、安装、维护
  - 比较算法的可靠性
    - 假阳性(False positive):允许未经授权的人访问
    - 假阴性(False negative):禁止授权的人访问
  - 隐私?
  - 如果被伪造，如何revoke?

    可以被伪造。收回权限

生物识别技术

- 常见用途
  - 特殊情况下的物理安全
  - 结合
    - 多种生物识别技术
    - 生物识别和PIN
    - 生物识别和令牌(token)

生物识别技术(Biometrics)

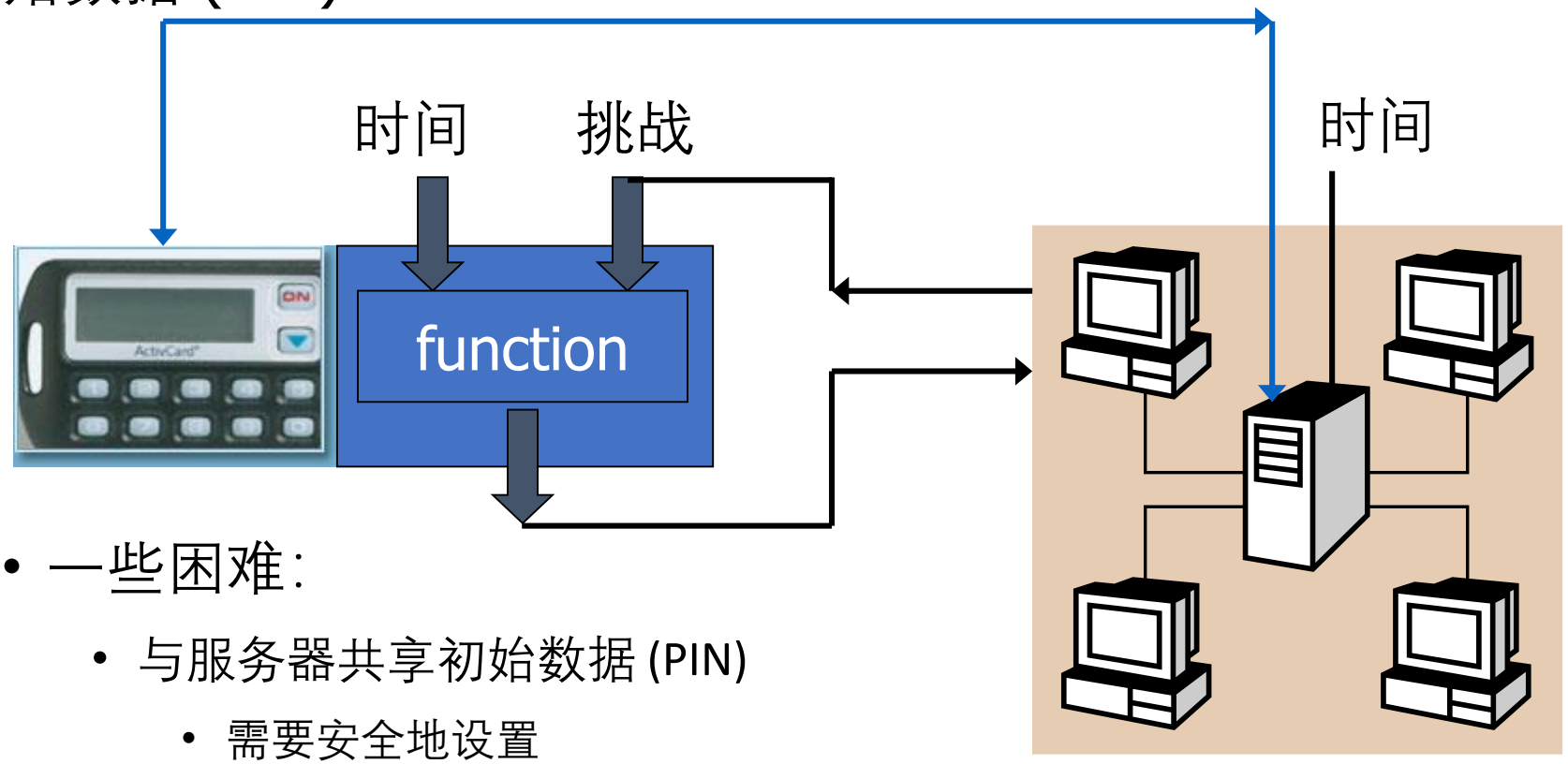
- RSA 2017创新沙盒冠军UNIFYID (https://unify.id)
  - 利用现在越来越丰富的各种传感器获取用户的数据。比如，手机里的陀螺仪，加速器，GPS，周围光感应，WIFI，蓝牙等等，对pc可以包括击键间隔，鼠标的移动和滚轮的滚动，触摸板的移动，周边WIFI，蓝牙等等
  - 基于这些数据上传云端并使用学习算法，对用户的行为进行学习。
  - 5个9的识别正确率

基于令牌的认证智能卡(Token-based Authentication Smart Card)

- 具有嵌入式CPU和内存
  - 与小型智能卡阅读器进行对话
- 各种形式
  - PIN 保护的存储卡
    - 输入PIN以获取密码
  - 基于PIN 和智能手机的令牌
    - Google 认证
  - 挑战/应答卡
    - 计算机创建一个随机挑战
    - 输入PIN 对卡内的挑战值进行加密/解密

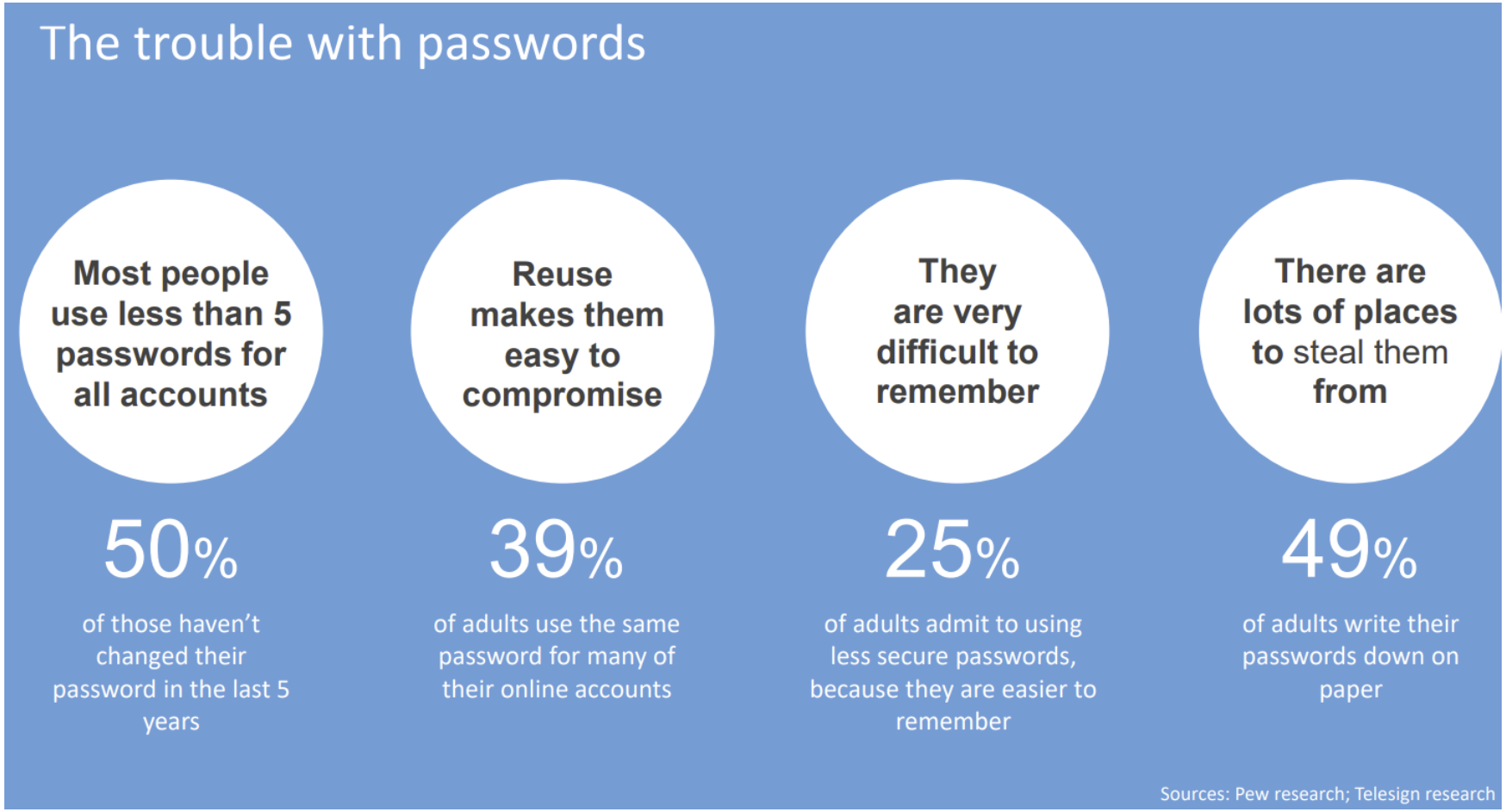
智能卡示例

初始数据 (PIN)



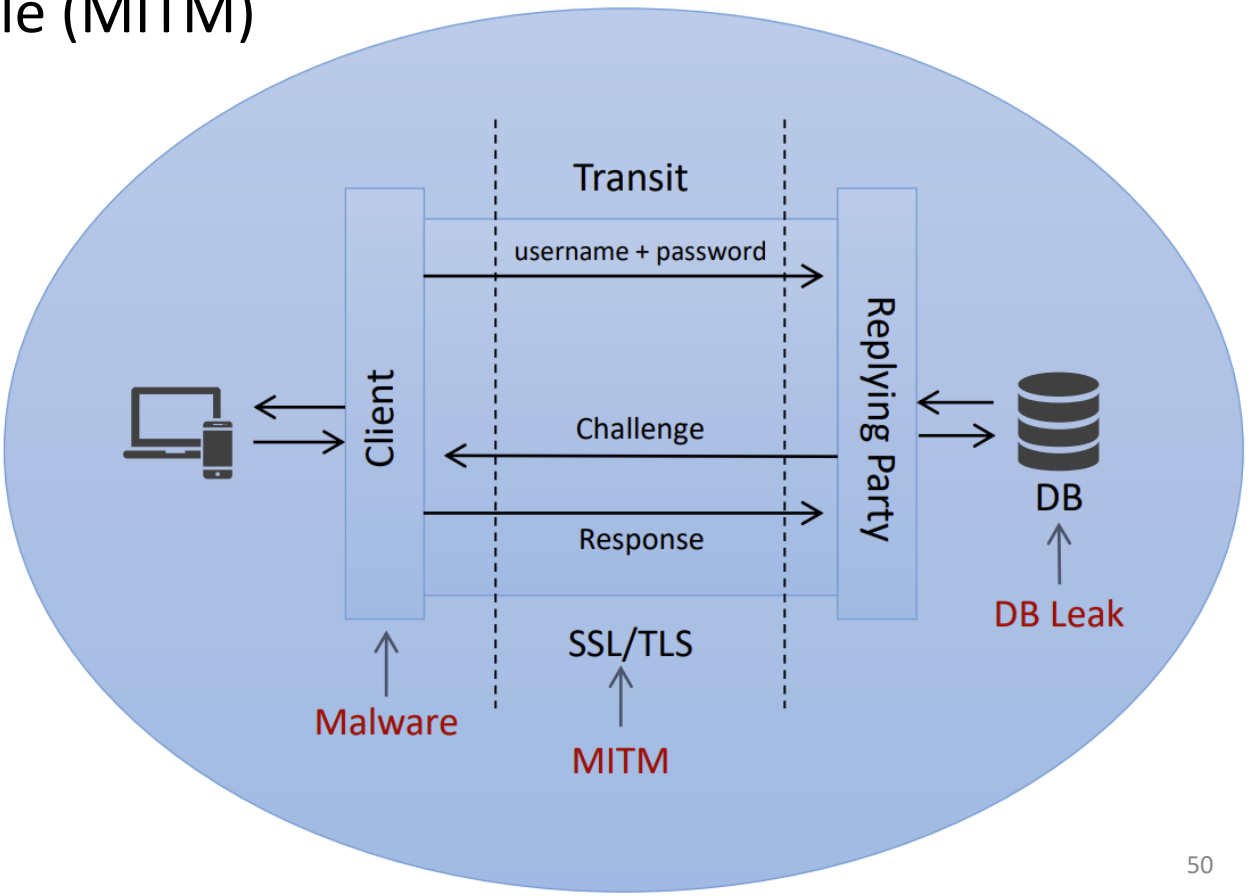
- 一些困难:
  - 与服务器共享初始数据 (PIN)
    - 需要安全地设置
    - 多站点共享数据库
  - 时钟倾斜(Clock skew)

基于password的认证的问题



基于password的认证的问题

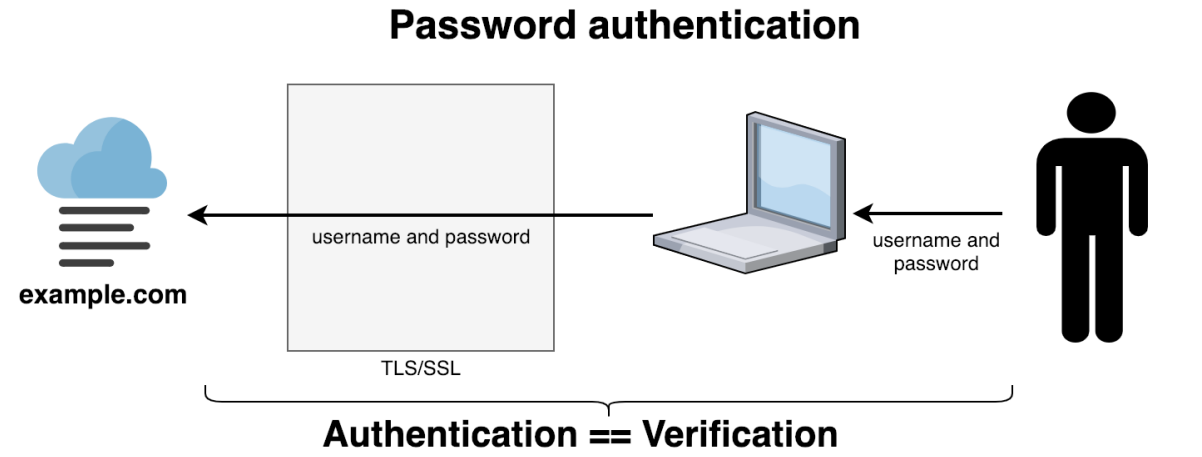
- Malware
- Man In The Middle (MITM)
- Database Leak

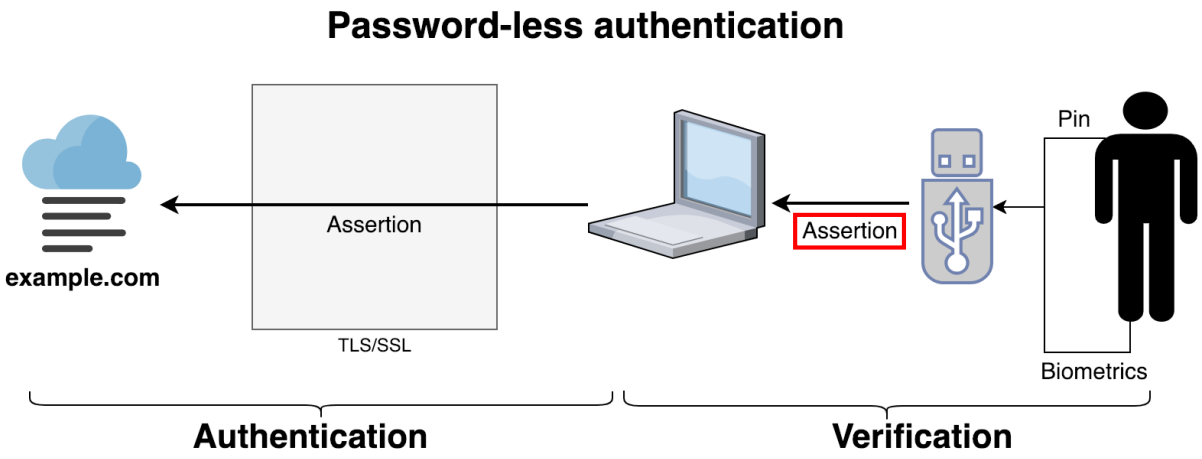


改进的方法

- 多因素认证，Multi-Factor Authentication (MFA)
  - 要求用户使用两个类别进行身份验证的安全增强方法
    - Something you know (such as password or PIN)
    - Something you have (such as one-time pin (OTP) to mobile phone)
    - Something you are (biometrics such as Fingerprint or FaceID)
  - credentials必须来自两个不同的类别，以提供更高的安全性

改进的方法

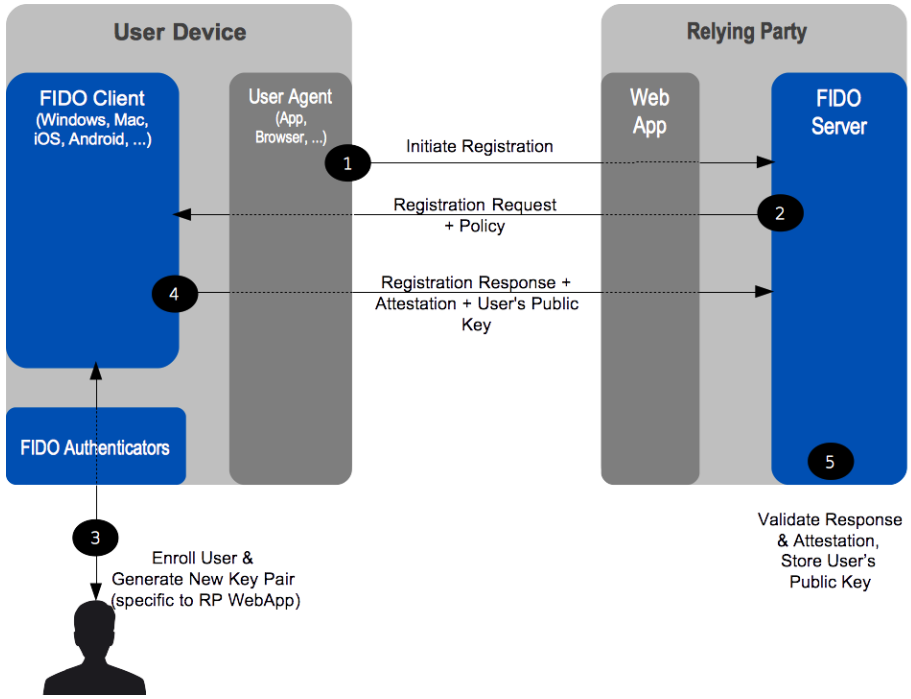

Password authentication
example.com
username and password
TLS/SSL
username and password
Authentication == Verification


Password-less authentication
example.com
Assertion
TLS/SSL
Assertion
Pin
Biometrics
Authentication
Verification

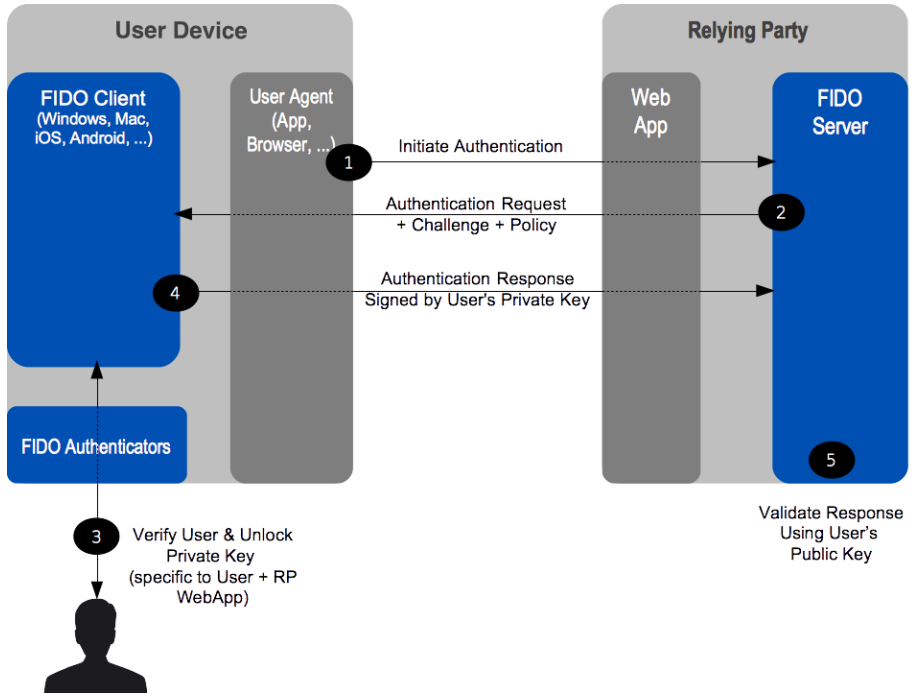
FIDO

- **FIDO** —**F**ast **ID**entity **O**nline
- FIDO相较于传统认证方式的两个重要不同点
  - 基于公私钥对的非对称加密体系
  - 只在本地的可信执行环境（TEE）中存储用户的生物特征信息
- FIDO 1.0: Universal Authentication Framework (UAF) Protocol, Universal 2nd Factor (U2F) Protocol
- FIDO 2.0: WebAuthn, CTAP

Universal Authentication Framework (UAF)



Registration Message Flow



Authentication Message Flow

https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-uaf-overview-v1.1-id-20170202.html

The Universal Second Factor (U2F)

目标:

- **Browser malware cannot steal user credentials**
- U2F should not enable tracking users across sites
- U2F uses counters to defend against token cloning

U2F token
(holds user credentials)

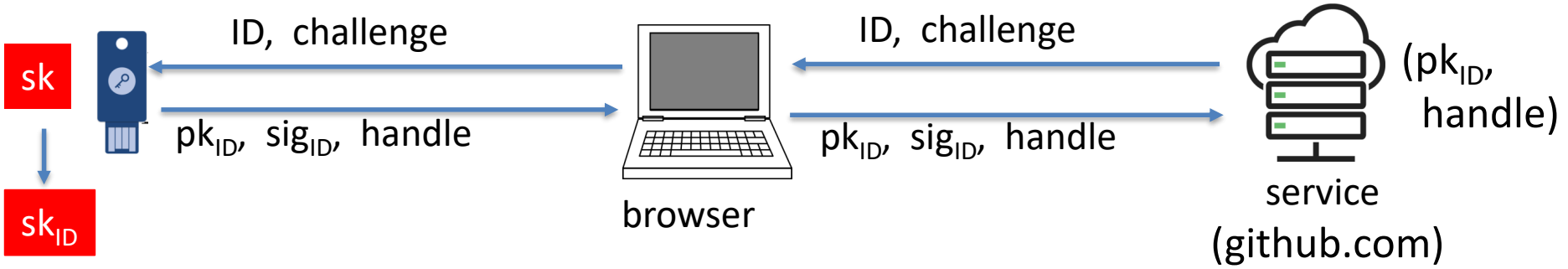
browser

service (github.com)

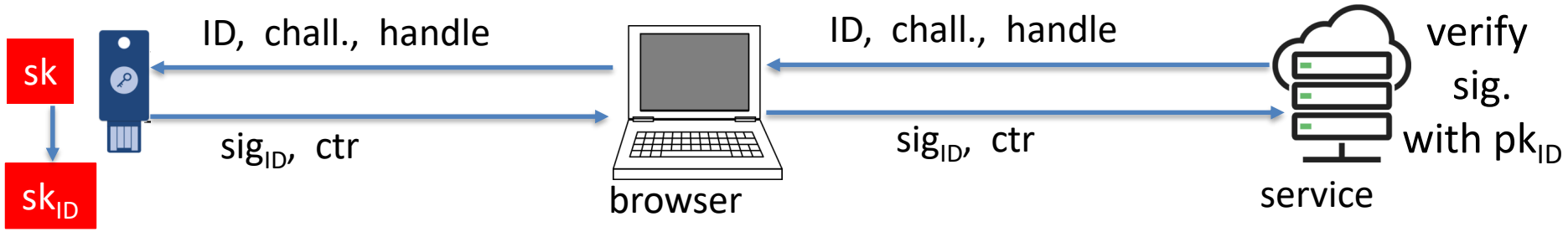
https://hybrismart.com/2019/05/23/authentication-with-hardware-security-keys-via-webauthn-in-sap-commerce-cloud/

The U2F protocol: two parts (simplified)

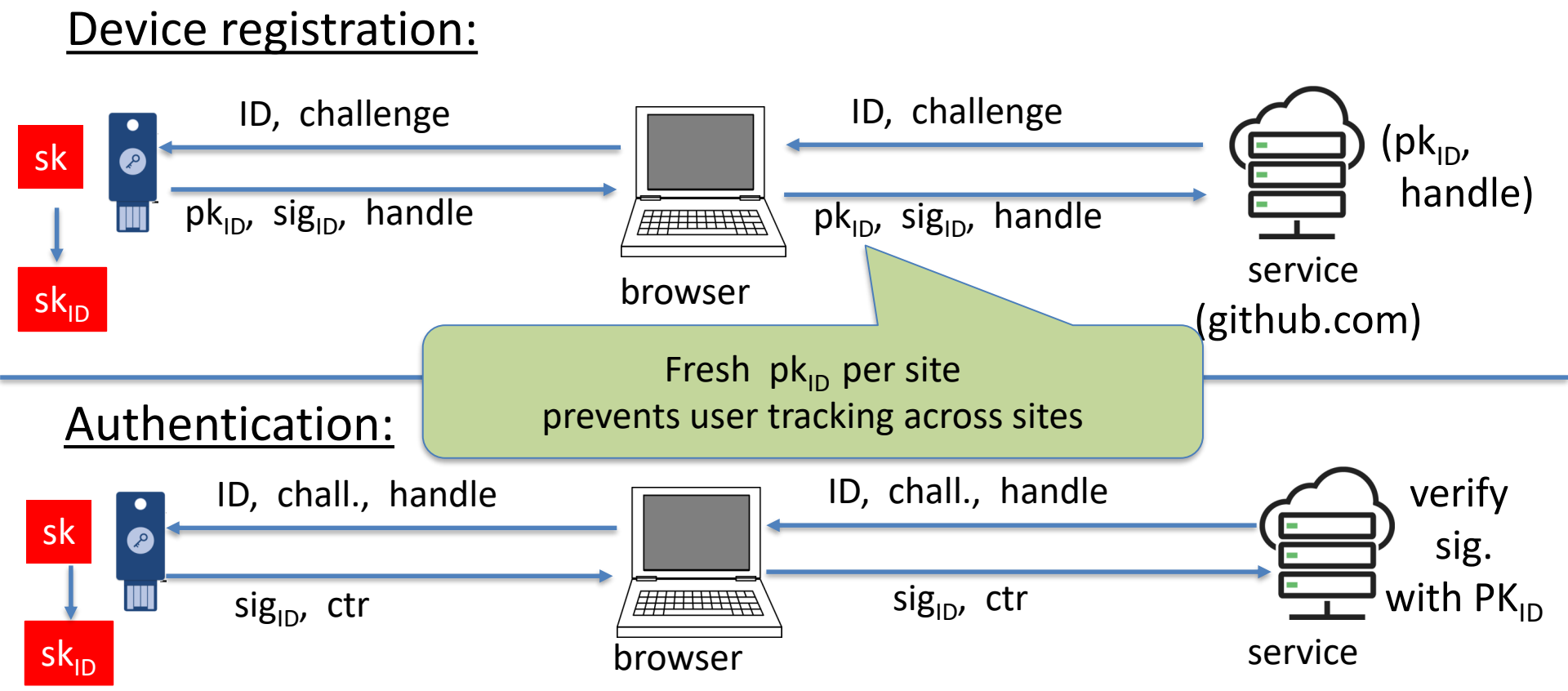
Device registration:



Authentication:

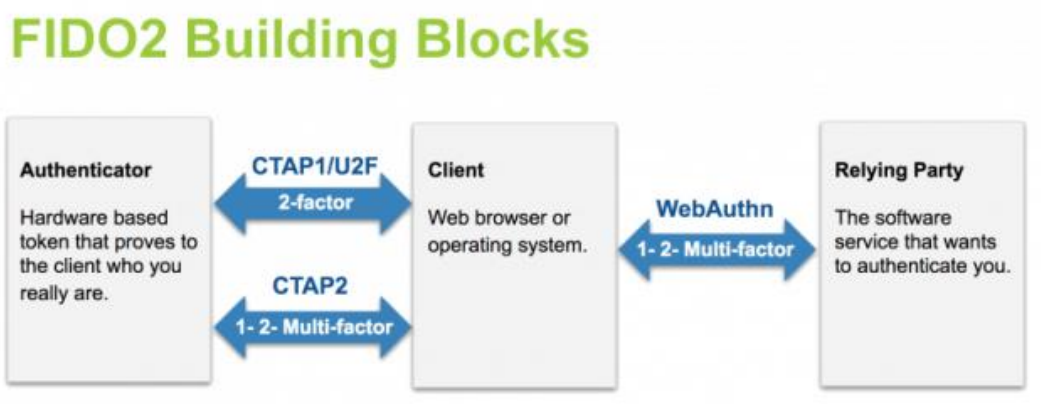


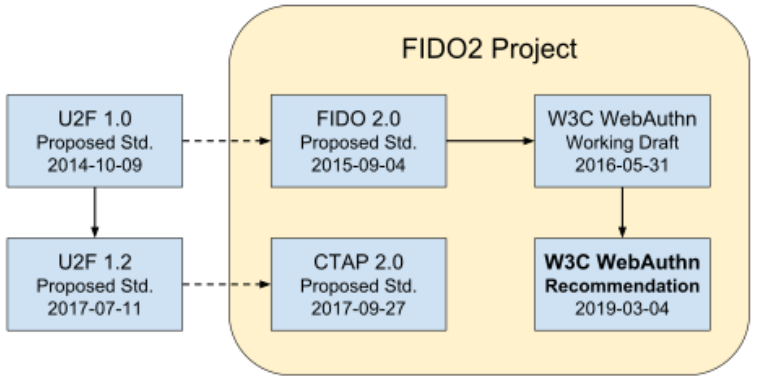
The U2F protocol: two parts (simplified)



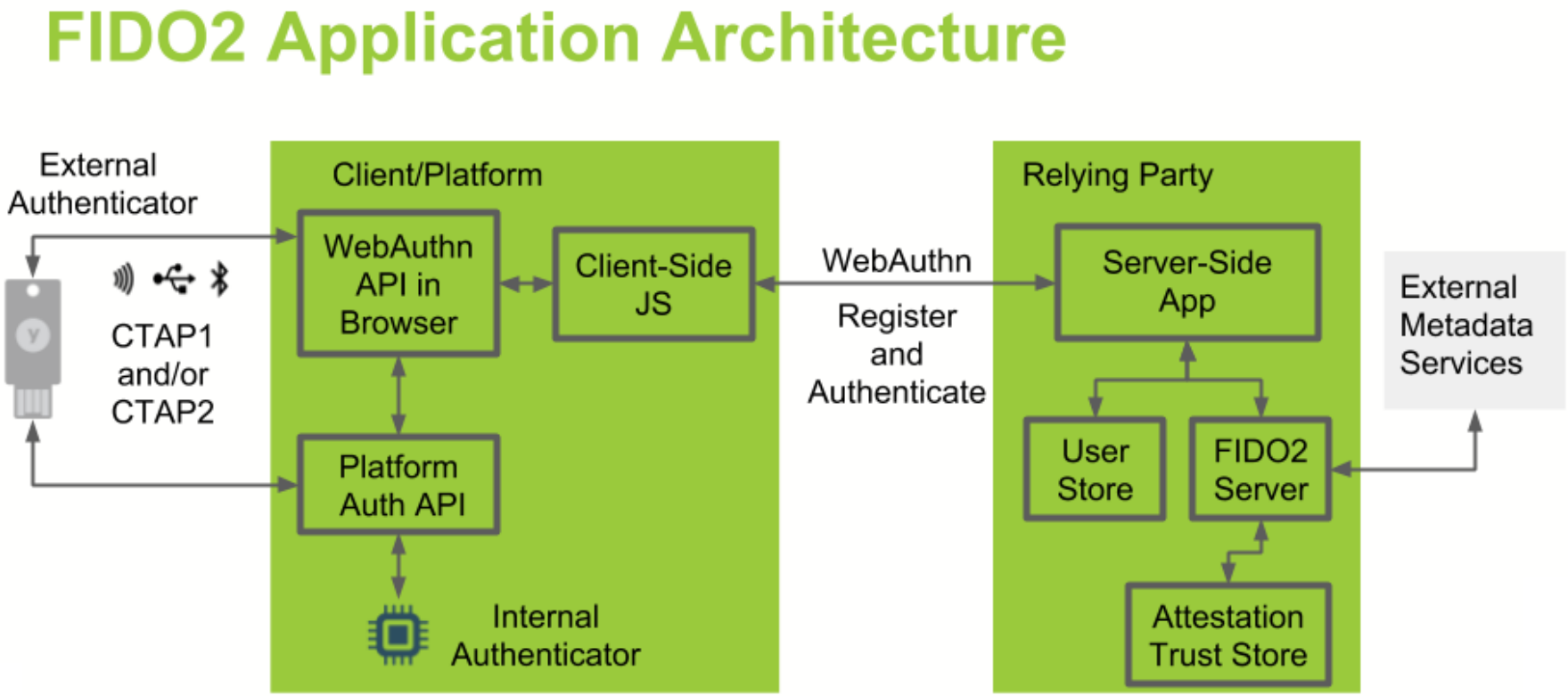
FIDO 2

- **WebAuthn** (the client API)
  - A browser JS API that describes an interface for creating and managing public key credentials.
- **CTAP** (the authenticator API), Client to Authenticator Protocols





FIDO 2

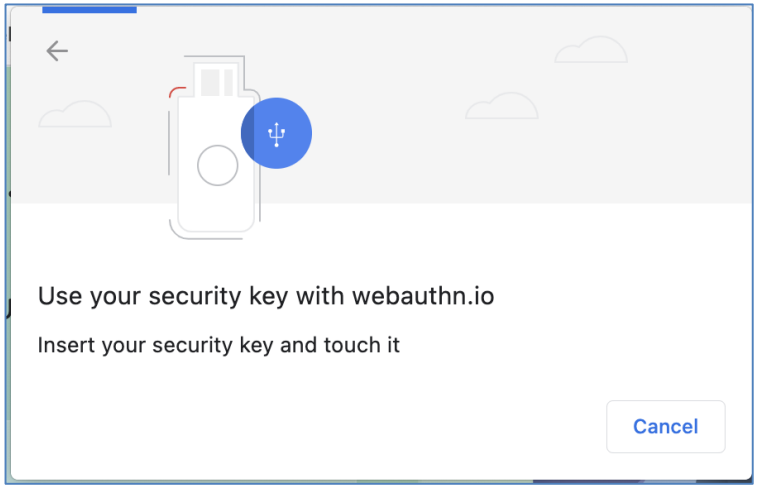

FIDO2 Application Architecture
External Authenticator
CTAP1 and/or CTAP2
Client/Platform
WebAuthn API in Browser
Client-Side JS
Platform Auth API
Internal Authenticator
WebAuthn
Register and Authenticate
Relying Party
Server-Side App
User Store
FIDO2 Server
Attestation Trust Store
External Metadata Services

WebAuthn

A browser Javascript API:

- Javascript from web site can register a U2F device,

  and later user can authenticate to web site with U2F device



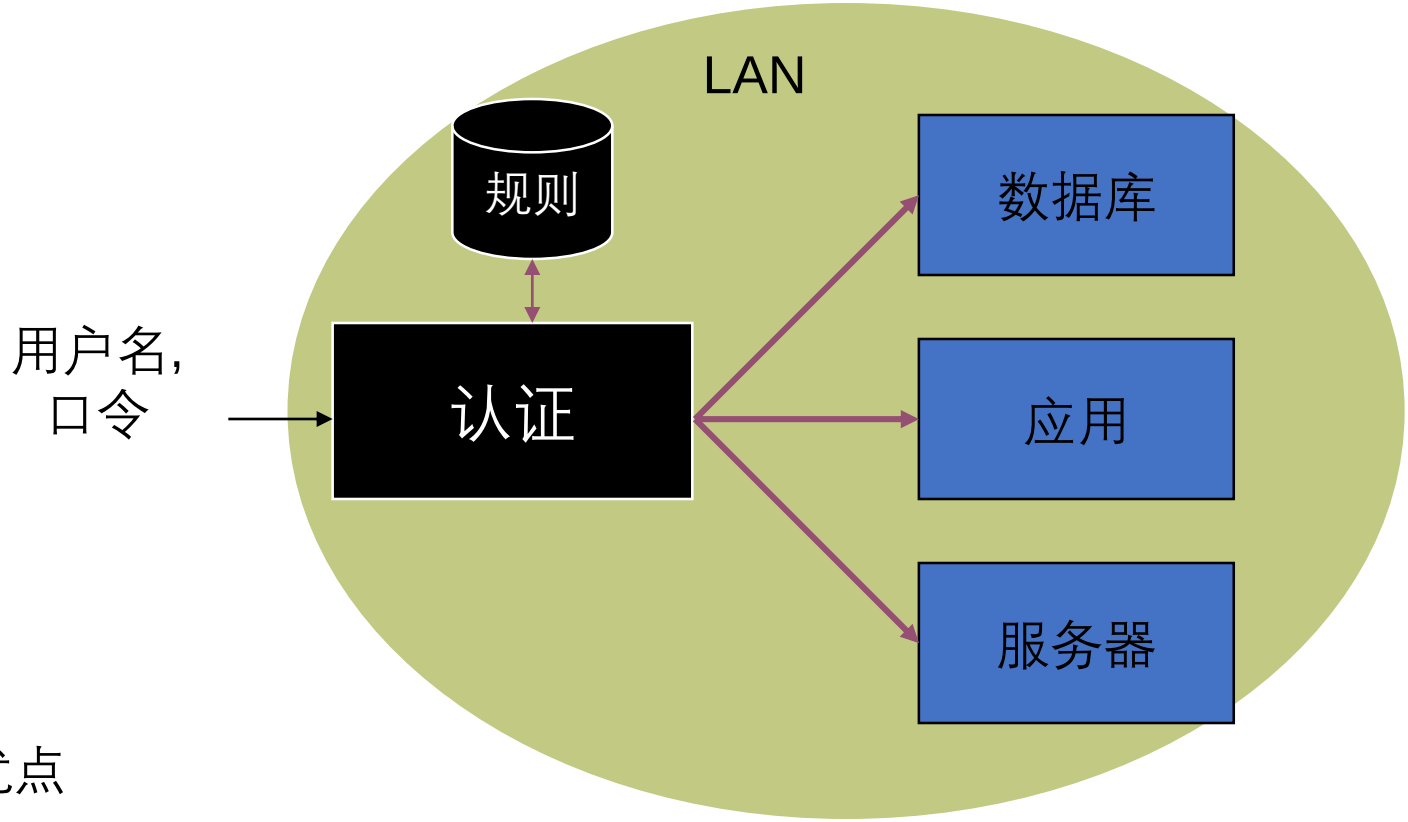
Web Authentication

- Security
  - Strong cryptography
  - Unphishable
  - Resists data-breach and brute force attacks
  - Test of user presence
  - Attestation
- Usability
  - Passwordless
  - One- or two-factor
  - In-device, biometric

大纲

- 用户认证
  - 口令认证，Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介（KPC和CA）

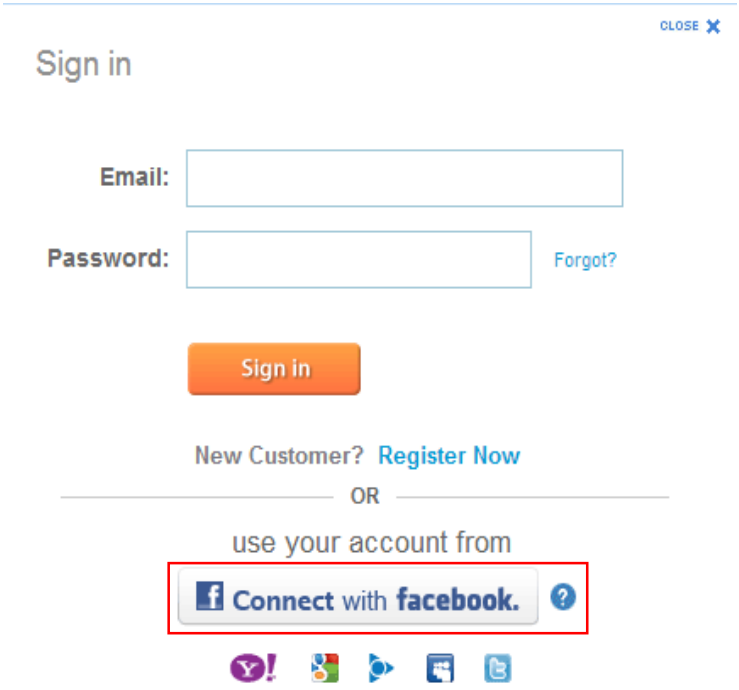
单点登录(Single Sign-on)系统



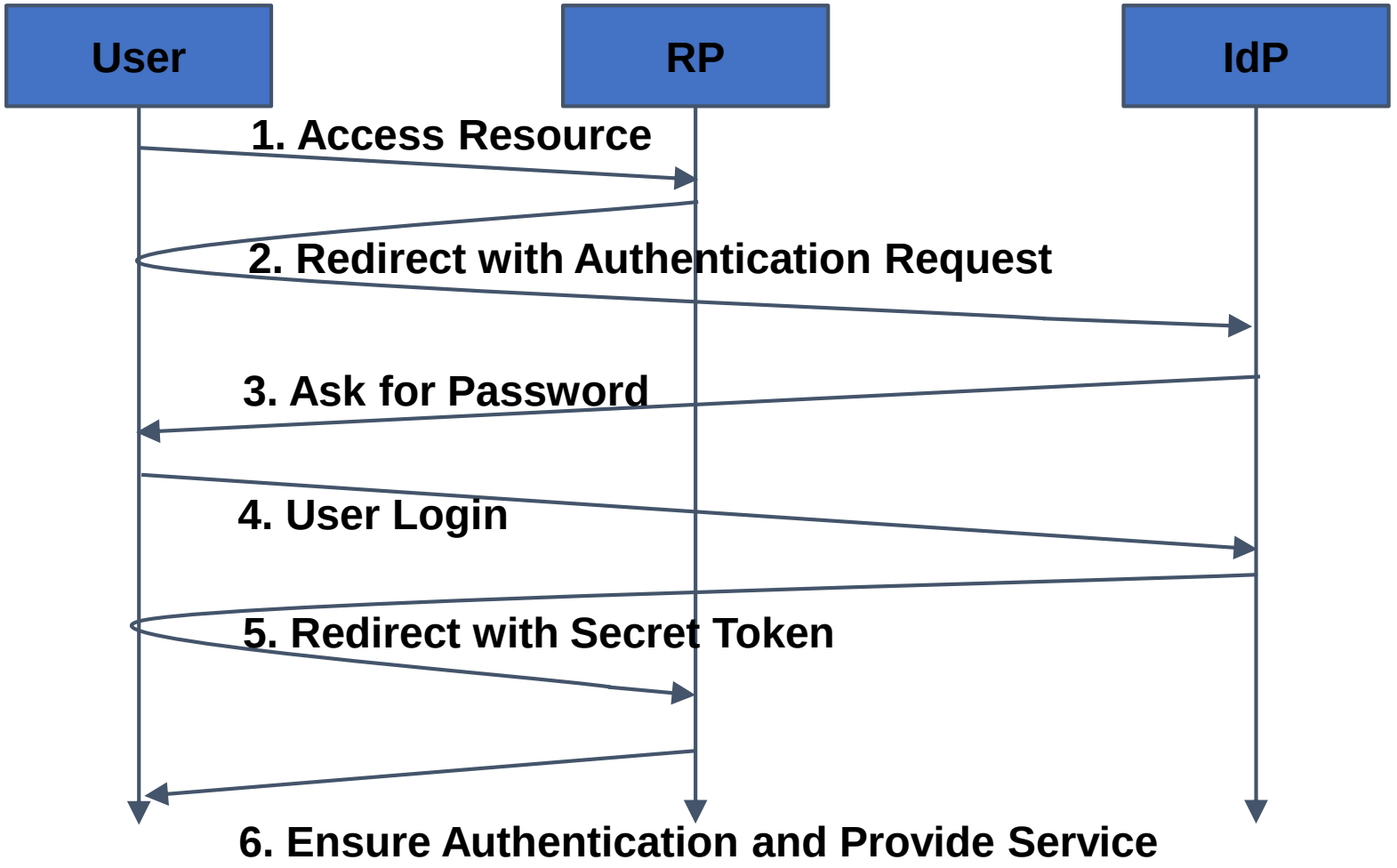
- 优点
  - 用户登录一次
  - 无需在多个站点或应用程序进行多次身份验证
  - 可以设置集中的策略

Web的单点登录系统

- 涉及的实体
  - IdP (Identity party, 如Facebook and Google)
  - RP (Relying party, 如 NYTimes)
  - User
- 例子: 用户通过Facebook、QQ提供的身份登录到第三方网站



Web的单点登录系统


User
RP
IdP
1. Access Resource
2. Redirect with Authentication Request
3. Ask for Password
4. User Login
5. Redirect with Secret Token
6. Ensure Authentication and Provide Service

大纲

- 用户认证
  - 口令认证，Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介（KPC和CA）

可信中介(Trusted Intermediaries)

对称密钥问题:

- 两个实体如何通过网络建立共享密钥?

方法:

- 可信的密钥分配中心（key distribution center, KDC）充当实体之间的中介

公钥问题:

- 当Alice获得Bob的公钥（从网站，电子邮件，U盘）时，她怎么知道这是Bob的公钥，而不是Trudy的公钥

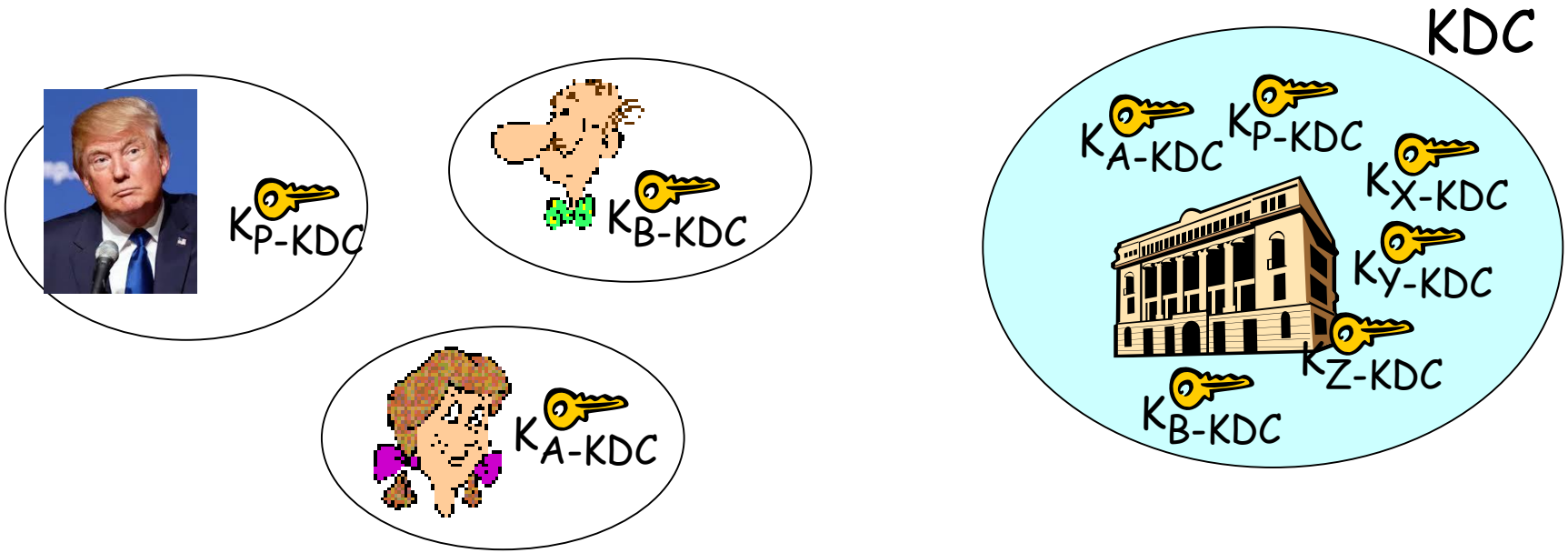
方法:

- 可信认证中心(certification authority, CA)

密钥分发中心(KDC)

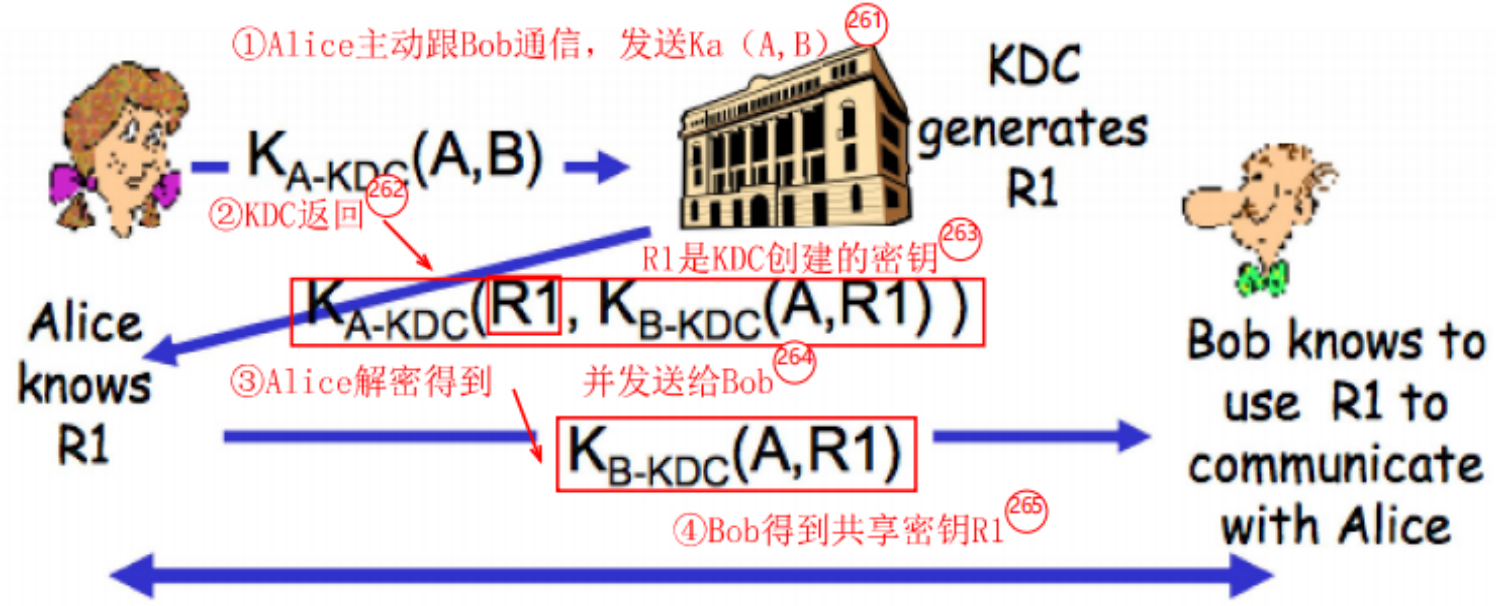
对称密钥咩有公钥私钥之分

- Alice，Bob需要共享对称密钥
- KDC: 服务器与每个注册用户（许多用户）共享不同的密钥
- Alice，Bob知道自己的对称密钥 $K_{A\text{-}KDC}$, $K_{B\text{-}KDC}$ ， 用于与KDC进行通信。



密钥分发中心(KDC)

Q: KDC如何让Bob，Alice确定共享对称密钥以相互通信。



Alice和Bob通信：将R1用作共享对称加密的*session key*

为什么要②③大费周章？为什么不直接让KDC分别通知AB会话密钥R1即可？
266
——KDC会有更大的开销。（开两个端口发两个密文）

使用KDC的Ticket 和标准

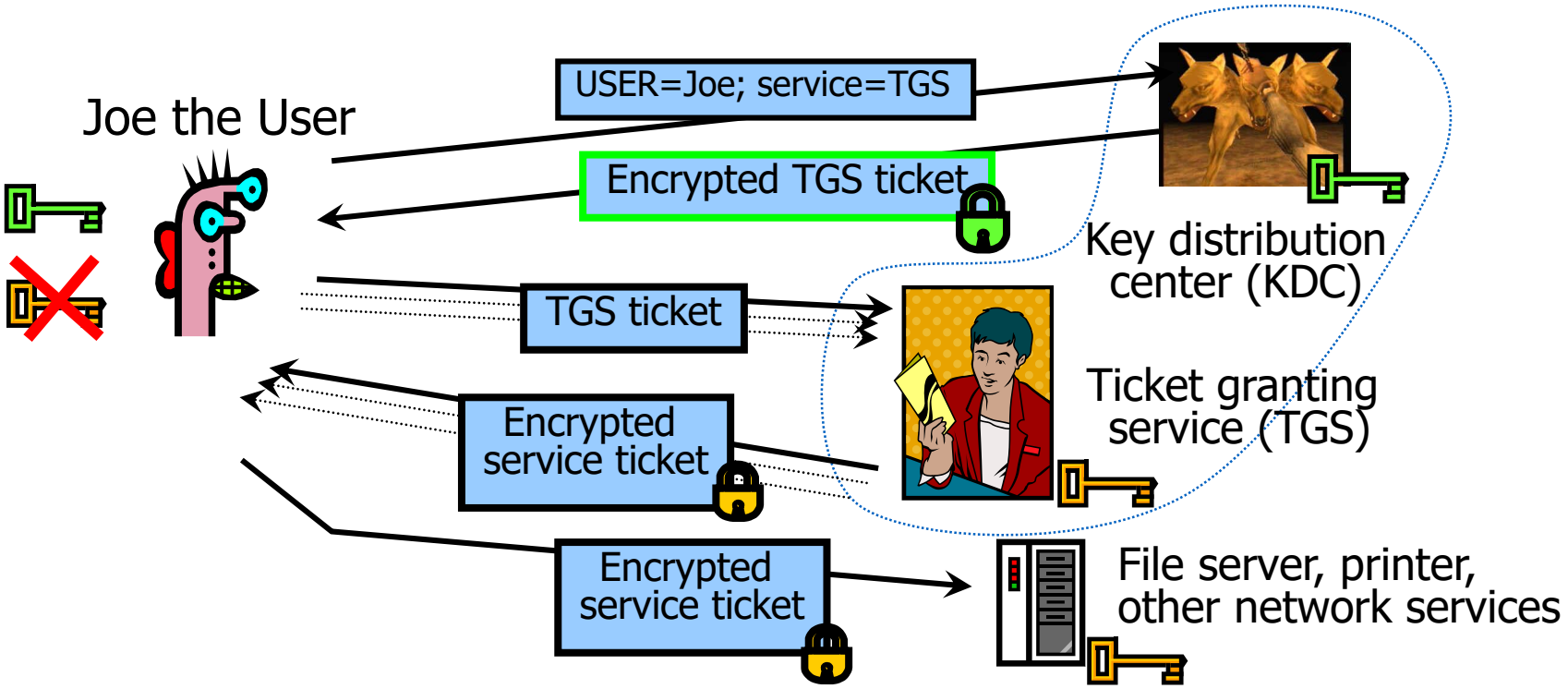
- Ticket
  - 在$K_{A\text{-}KDC}(R1, K_{B\text{-}KDC}(A,R1))$中, $K_{B\text{-}KDC}(A,R1)$ 也被称为ticket
  - Ticket具有有效期(expiration time)
- KDC 在 Kerberos中使用: 基于共享密钥的认证标准
  - 用户注册密码
  - 从密码派生共享密钥

Kerberos

- 来自MIT的可信密钥服务器系统
  - 最著名和最广泛实施的**可信第三方密钥分发系统**
- 在分布式网络中提供集中式的基于对称密钥的第三方认证
  - 允许用户访问通过网络的分布式服务
  - 无需信任所有站点
  - 而是全部信任中央认证服务器
- 两个版本: 4 & 5
- 广泛使用
  - Red Hat 7.2 and Windows Server 2003 or higher

Two-Step 认证

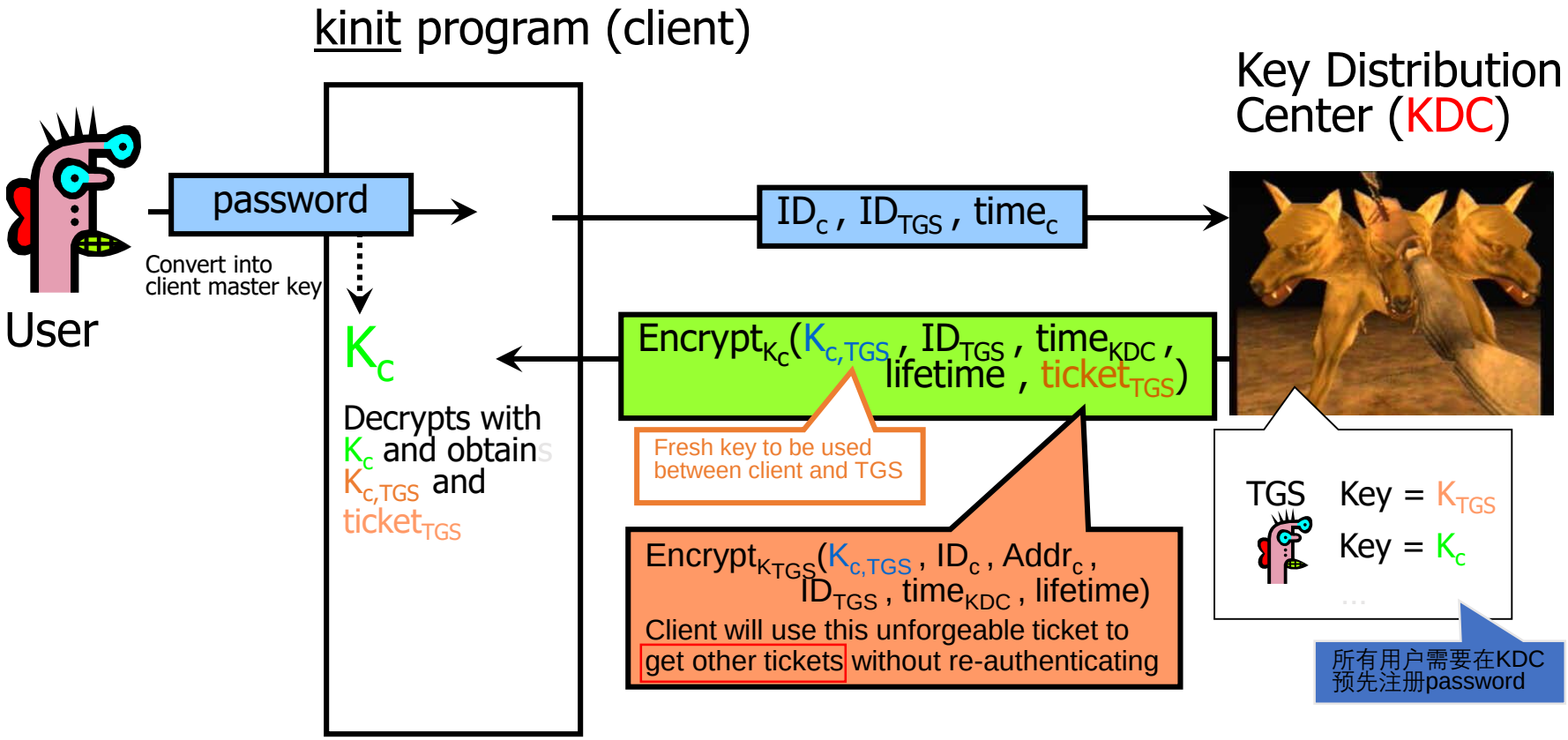
- 向KDC证明身份一次，以获得特殊的TGS ticket
- 使用TGS ticket 服务TGS 获取任何网络服务的tickets



Kerberos中的对称密钥

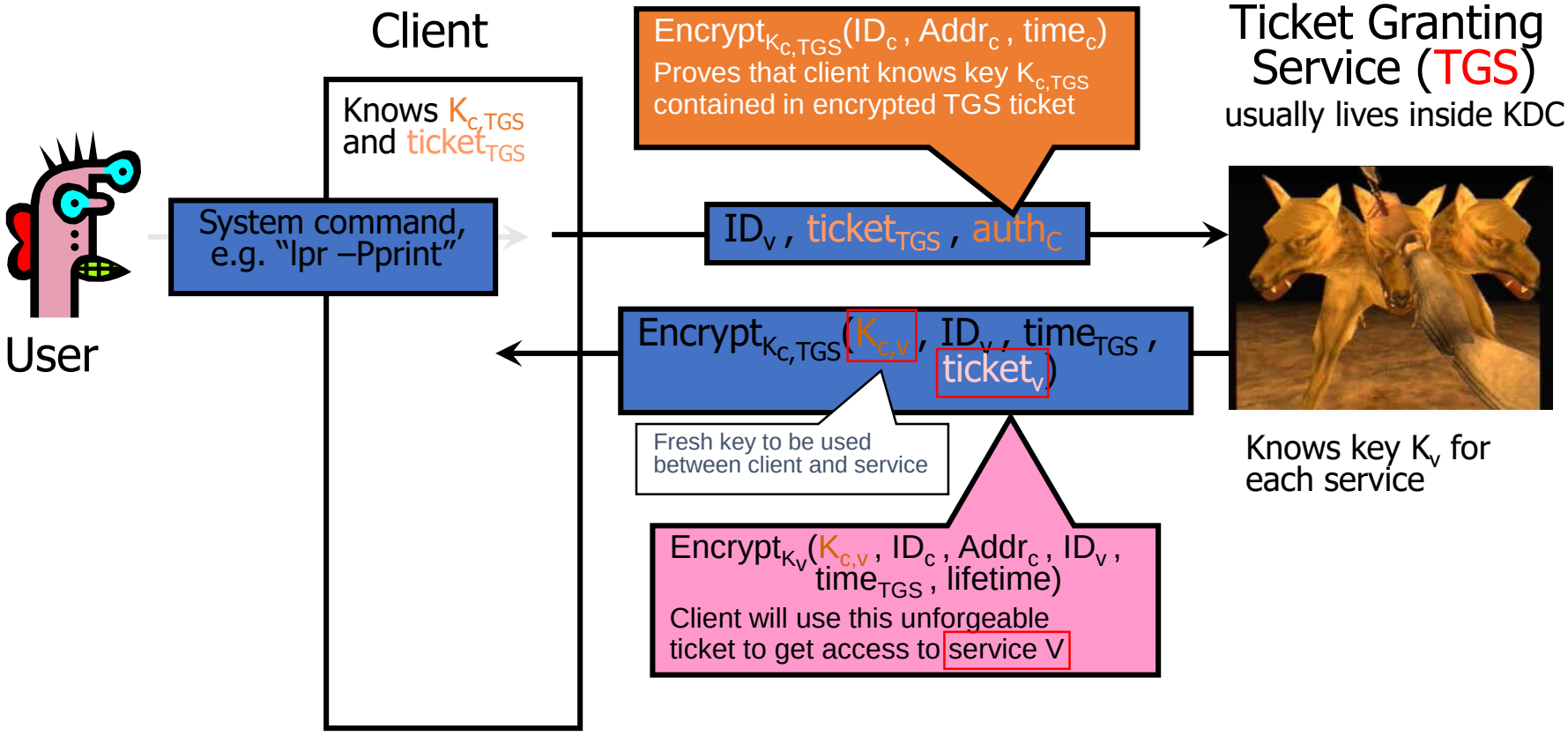
- K_c 是客户端C的长期密钥
  - 源自用户的密码
  - 由客户端和KDC所知
- K_{TGS} TGS的长期密钥
  - 由KDC和TGS (ticket granting service)所知
- K_v 网络服务V的长期密钥
  - 由 V 和TGS所知
  - 每个服务有单独的密钥
- $K_{c,TGS}$ 是客户端C和TGS之间的短期密钥
  - 由KDC创建，由C和TGS所知
- $K_{c,v}$ C和V之间的短期密钥
  - 由TGS创建，由C和V所知

“Single Logon” 认证



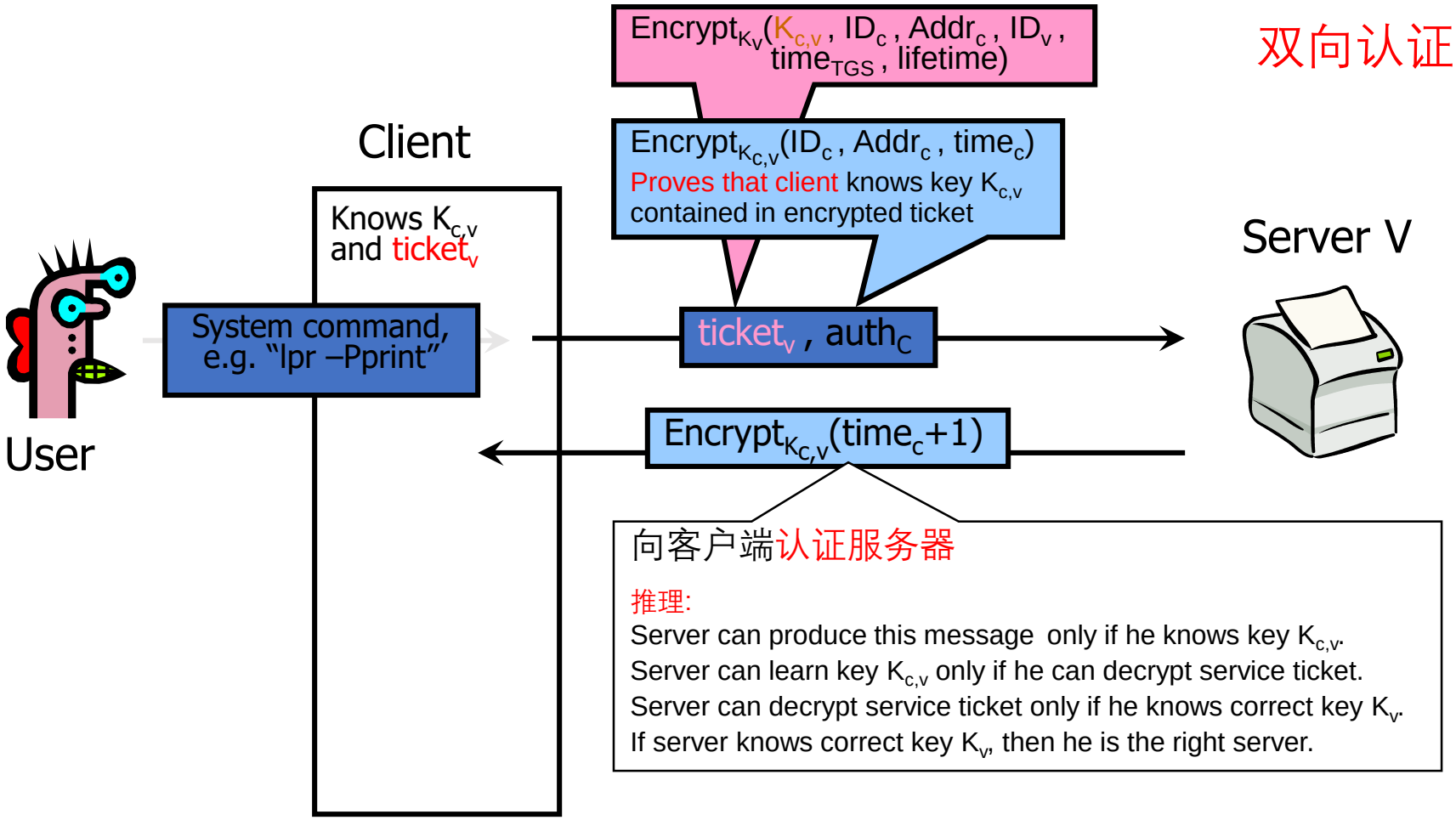
- 客户只需获得一次TGS ticket（比如每天早上）
  - ticket是加密的; 客户端无法伪造或篡改它

获得一个服务Ticket



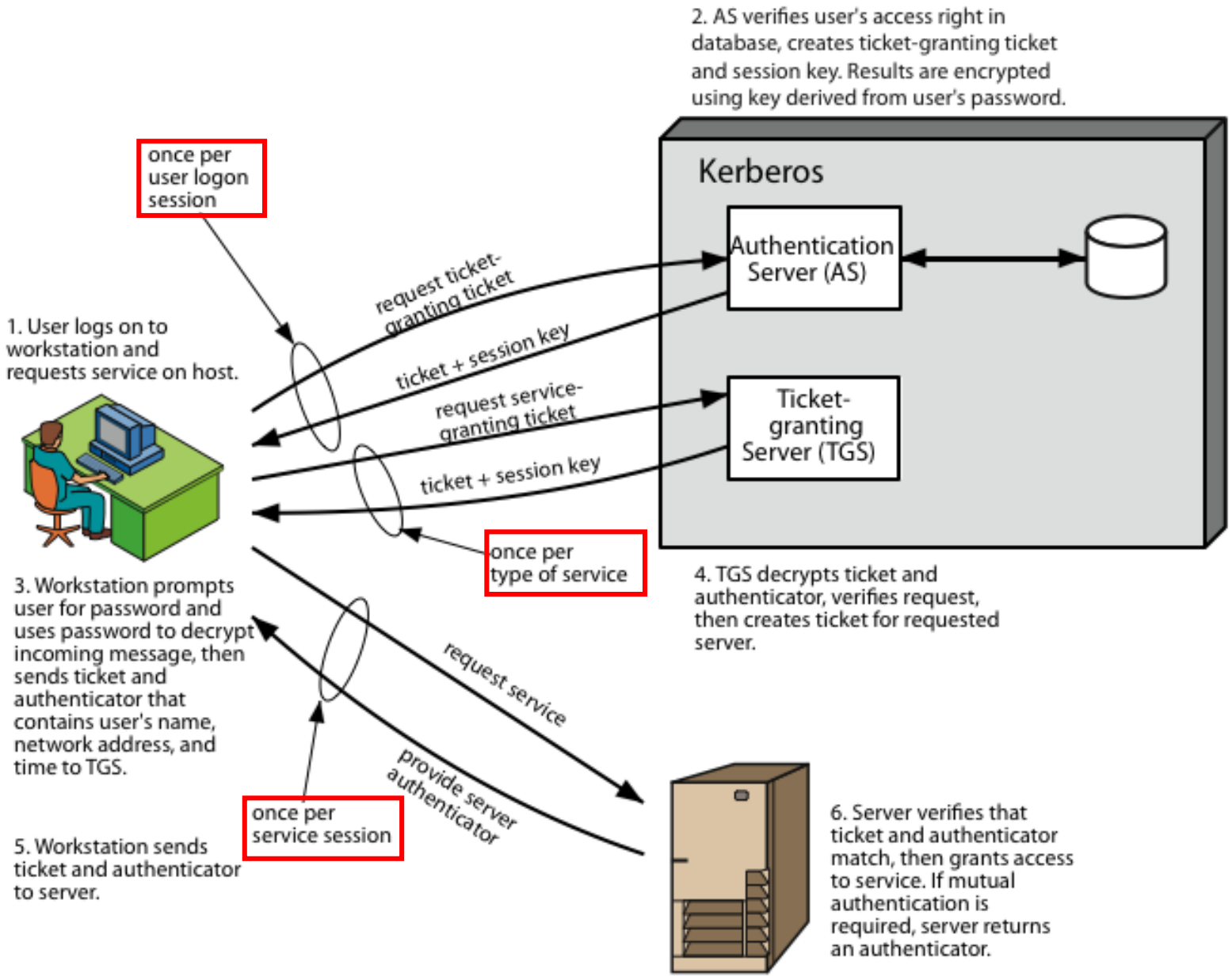
- 客户使用TGS票据来获取每个网络服务的服务票据和短期密钥
  - 每个服务有一张加密的，不可伪造的票据

获得服务



- 对于每个服务请求，客户端使用该服务的短期密钥以及他从TGS收到的票据

Kerberos 概述


2. AS verifies user's access right in database, creates ticket-granting ticket and session key. Results are encrypted using key derived from user's password.
once per user logon session
Kerberos
Authentication Server (AS)
request ticket-granting ticket
ticket + session key
1. User logs on to workstation and requests service on host.
request service-granting ticket
Ticket-granting Server (TGS)
ticket + session key
once per type of service
4. TGS decrypts ticket and authenticator, verifies request, then creates ticket for requested server.
3. Workstation prompts user for password and uses password to decrypt incoming message, then sends ticket and authenticator that contains user's name, network address, and time to TGS.
request service
provide server authenticator
once per service session
5. Workstation sends ticket and authenticator to server.
6. Server verifies that ticket and authenticator match, then grants access to service. If mutual authentication is required, server returns an authenticator.

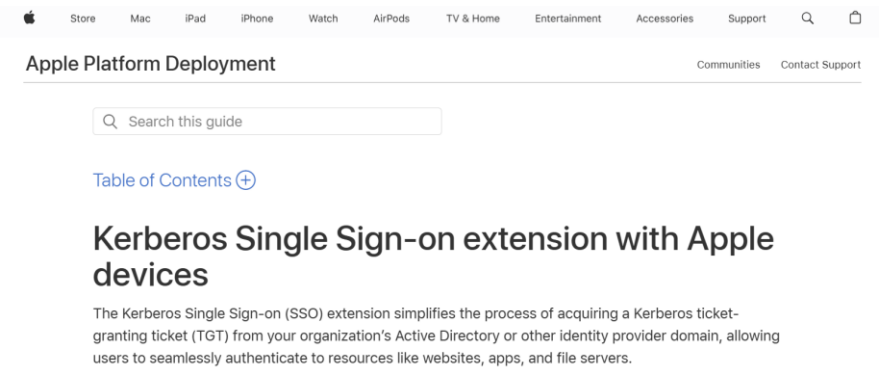
Kerberos的重要思想

- 短期会话密钥(session keys)
  - 长期秘密仅用于派生出短期密钥
  - 每对用户-服务器具有单独的会话密钥
    - … 但多个用户-服务器会话重复使用相同的密钥
- 身份证明基于认证者(authenticators)
  - 客户端使用短期会话密钥加密身份、地址和当前时间
    - 还可以防止重放（如果时钟全局同步）
  - 服务器分别获知此密钥（通过客户端无法解密的加密票据）并验证用户身份
- 仅有对称加密

Kerberos的实际应用

- 电子邮件、FTP、网络文件系统和许多其他应用程序已被kerberized
  - 对最终用户来说，Kerberos的使用是透明的
  - 透明度对可用性(usability)非常重要!

Single Sign-on?



https://support.apple.com/en-sg/guide/deployment/depe6a1cda64/web

可信中介(Trusted Intermediaries)

对称密钥问题:

- 两个实体如何通过网络建立共享密钥?

方法:

- 可信的密钥分配中心（key distribution center, KDC）充当实体之间的中介

公钥问题:

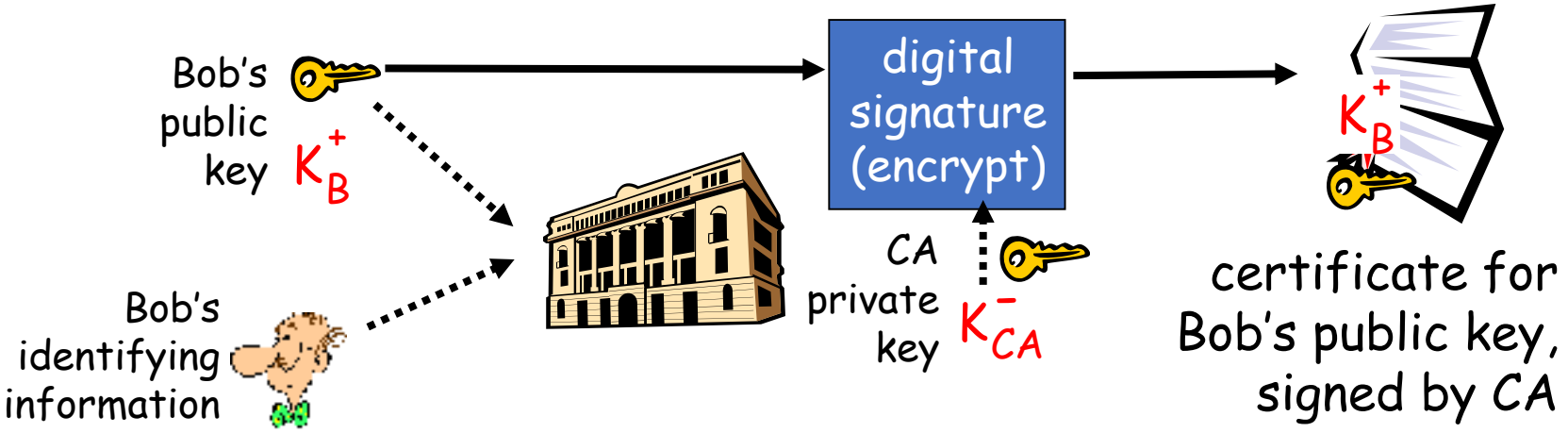
- 当Alice获得Bob的公钥（从网站，电子邮件，U盘）时，她怎么知道这是Bob的公钥，而不是Trudy的公钥

方法:

- 可信认证中心(certification authority, CA)

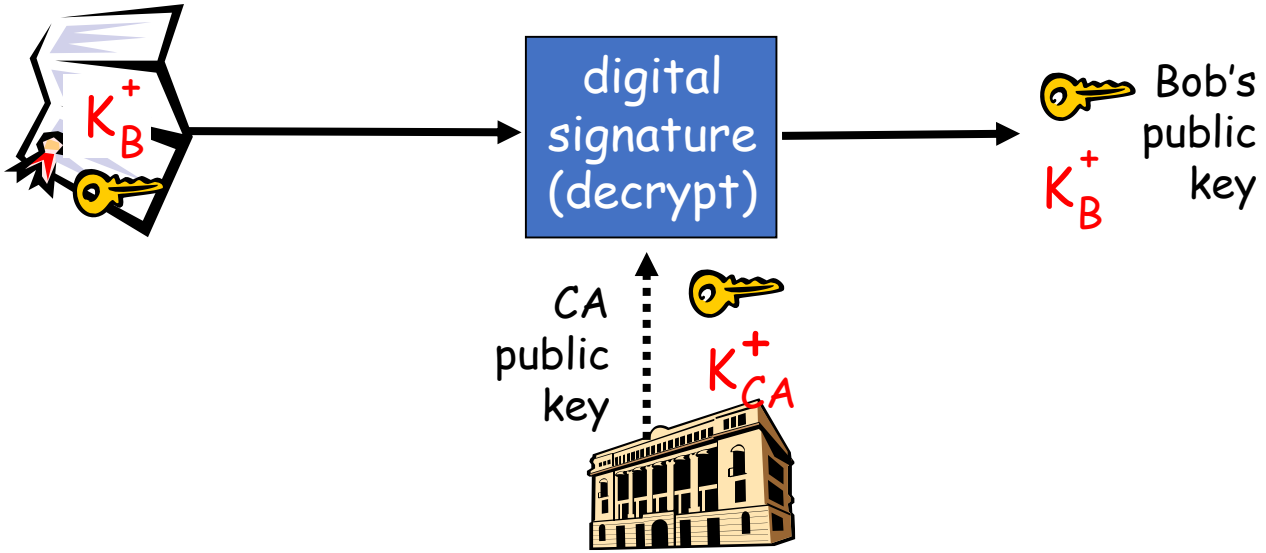
CA

- Certification authority (CA): 将公钥绑定到特定实体E
- E (person, router) 向CA注册公钥
  - E 向CA提供身份证明(“proof of identity”)
  - CA创建证书将E和它的公钥绑定。
  - 包含E的公钥的证书，由CA数字签名(CA称”这是E的公钥”)

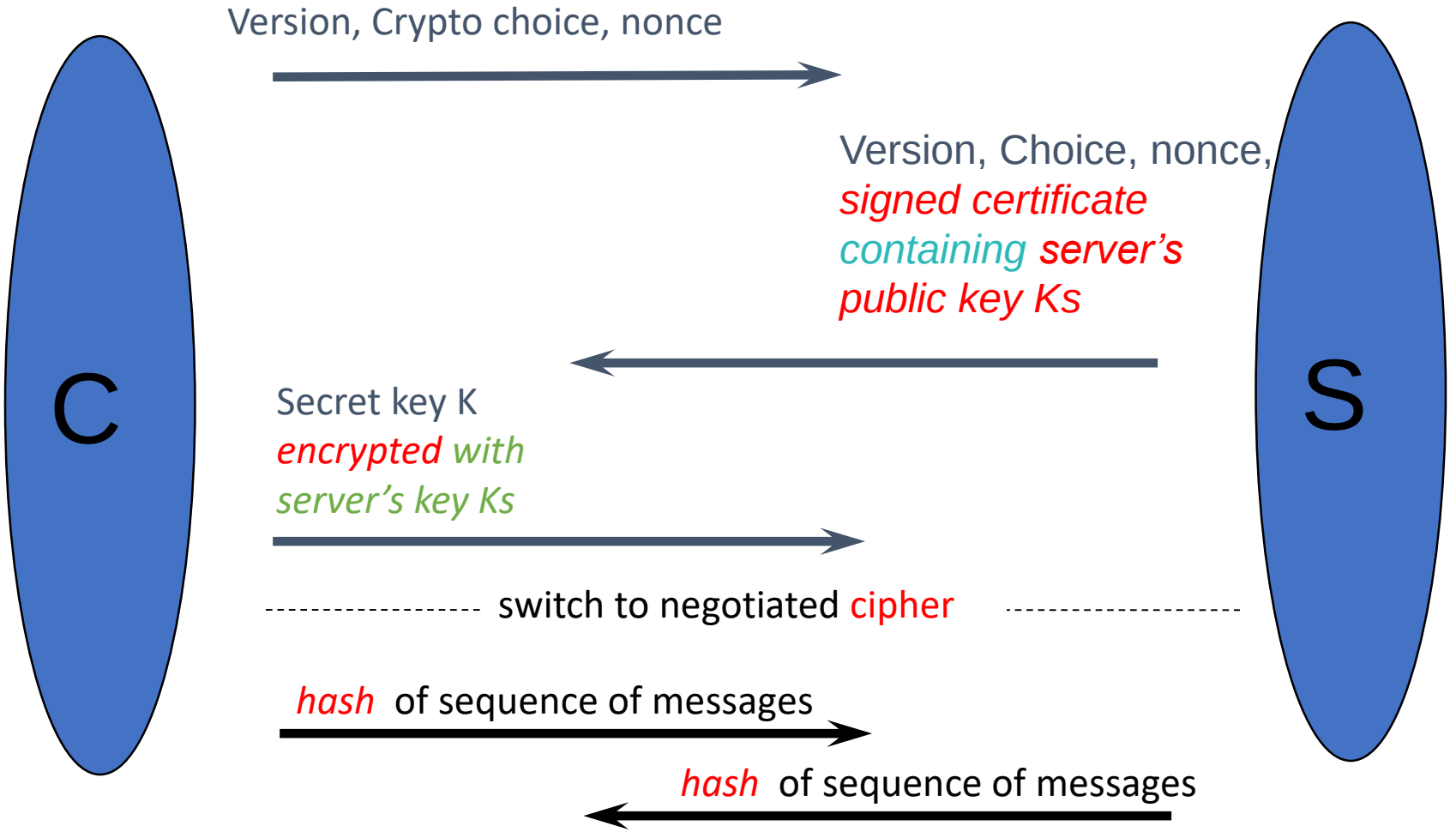


CA

- 当Alice想要Bob的公钥时:
  - 获得Bob的证书（从Bob或其它地方）
  - 将CA的公钥应用于Bob的证书，获取Bob的公钥
- CA是广泛使用的X.509标准的核心
  - SSL/TLS：部署在每个Web浏览器中
  - S/MIME（安全/多用途Internet邮件扩展）和IP Sec等



SSL的一般流程


Version, Crypto choice, nonce
Version, Choice, nonce,
signed certificate
containing server's
public key Ks
C
S
Secret key K
encrypted with
server's key Ks
switch to negotiated cipher
hash of sequence of messages
hash of sequence of messages

SSL/HTTPS中的认证

- 公司要求CA（例如Verisign）提供证书
- CA创建证书并对其签名
- 证书安装在服务器中（例如，Web服务器）
- 浏览器安装了根证书
  - 如：Windows根证书列表 https://gallery.technet.microsoft.com/Trusted-Root-Certificate-7ece659b
  - 浏览器验证证书并信任正确签名的证书

多KDC/CA 域

Secret keys:

- KDCs share pairwise key
- topology of KDC: tree with shortcuts

Public keys:

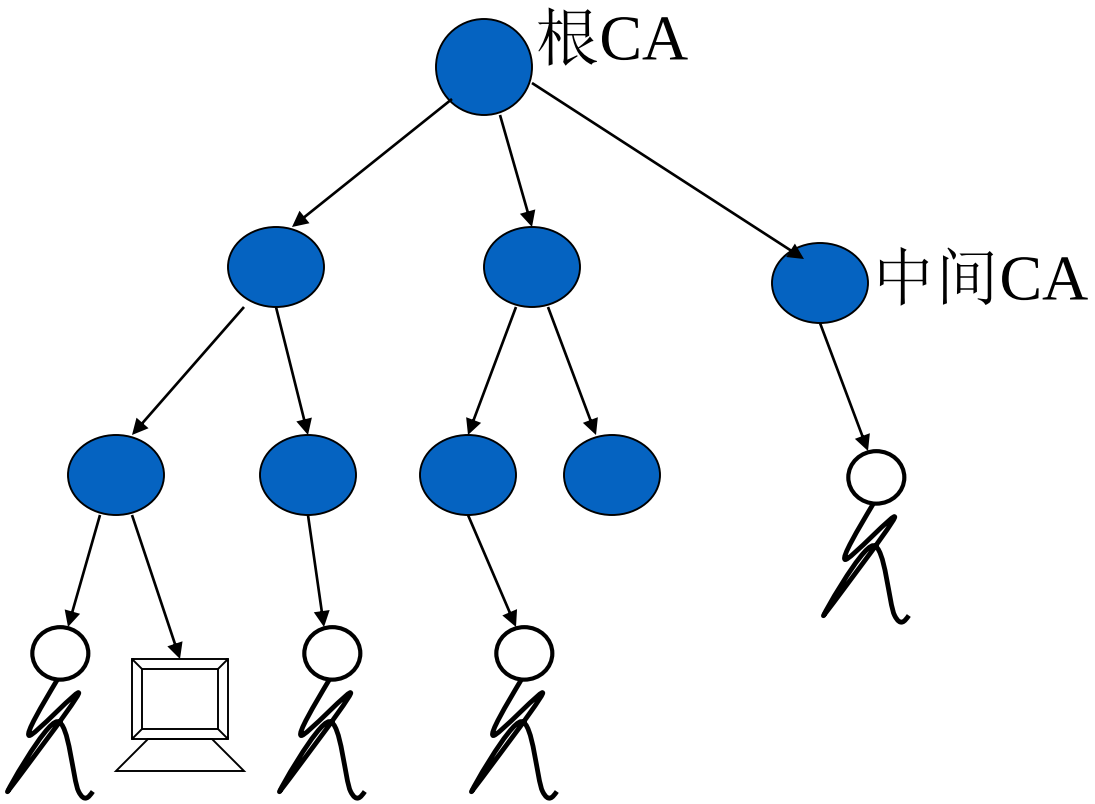
- cross-certification of CAs
- example: Alice with CA_A, Boris with CA_B
  - Alice gets CA_B's certificate (public key p_1), signed by CA_A
  - Alice gets Boris' certificate (its public key p_2), signed by CA_B (p_1)

CA信任关系

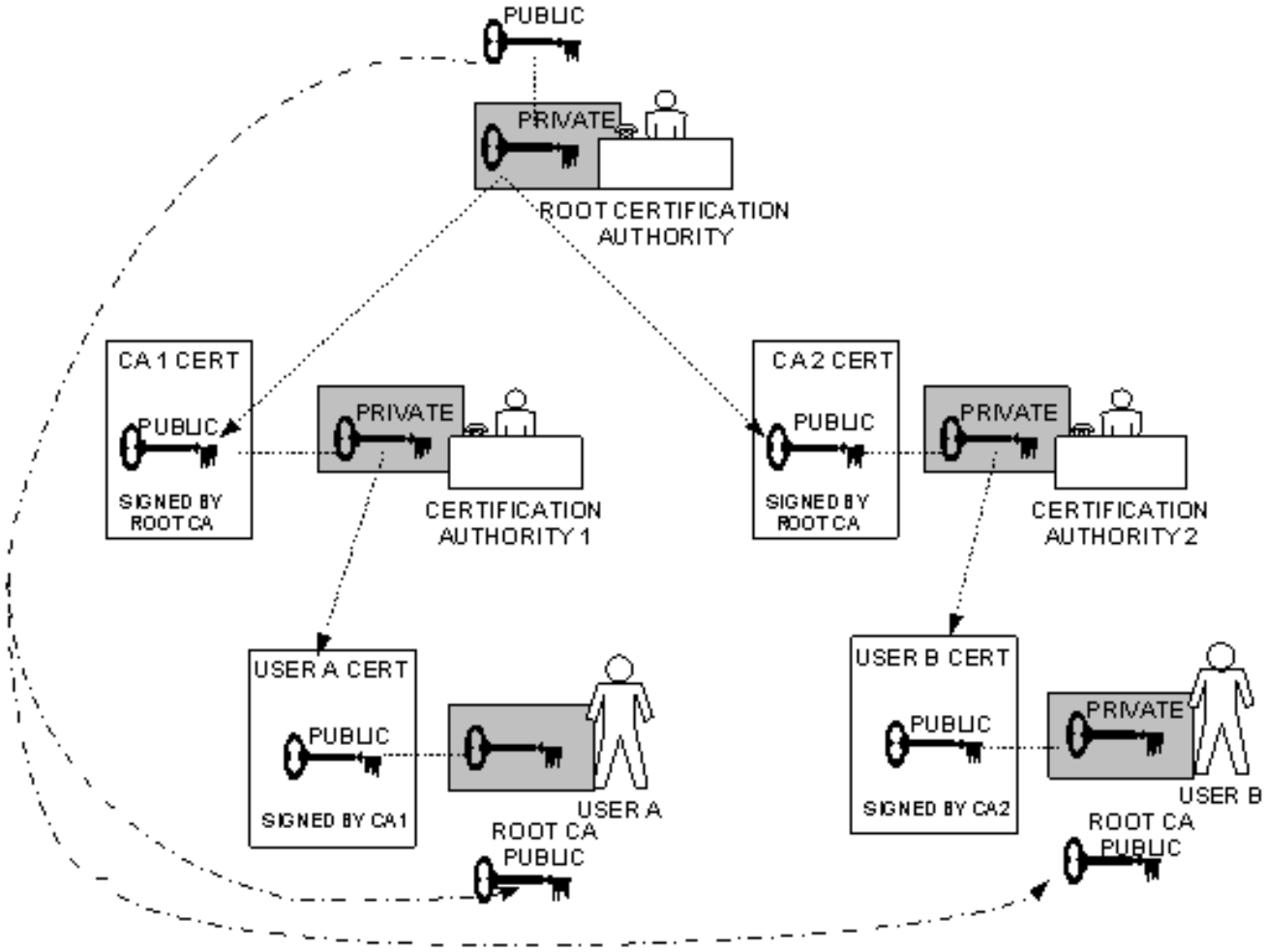
- 当一个安全个体看到另一个安全个体出示的证书时，他是否信任此证书？
  - 信任难以度量，总是与风险联系在一起
- 可信CA
  - 如果一个个体假设CA能够建立并维持一个准确的“个体-公钥属性”之间的绑定，则他可以信任该CA，该CA为可信CA
- 信任模型
  - 基于层次结构的信任模型
  - 交叉认证
  - 以用户为中心的信任模型
  - 浏览器信任列表认证模型

CA层次结构

- 对于一个运行CA的大型权威机构而言，签发证书的工作不能仅仅由一个CA来完成
- 它可以建立一个CA层次结构



CA层次结构


PUBLIC
PRIVATE
ROOT CERTIFICATION AUTHORITY
CA 1 CERT
PUBLIC
SIGNED BY ROOT CA
PRIVATE
CERTIFICATION AUTHORITY 1
CA 2 CERT
PUBLIC
SIGNED BY ROOT CA
PRIVATE
CERTIFICATION AUTHORITY 2
USER A CERT
PUBLIC
SIGNED BY CA1
USER A
USER B CERT
PUBLIC
SIGNED BY CA2
PRIVATE
USER B
ROOT CA PUBLIC
ROOT CA PUBLIC

CA层次结构的建立

- 根CA具有一个自签名的证书
- 根CA依次对它下面的CA进行签名
- 层次结构中叶子节点上的CA用于对安全个体进行签名
- 对于个体而言，它需要信任根CA，中间的CA可以不必关心(透明)；同时它的证书是由底层的CA签发的
- 在CA的机构中，要维护这棵树
  - 在每个节点CA上，需要保存两种cert

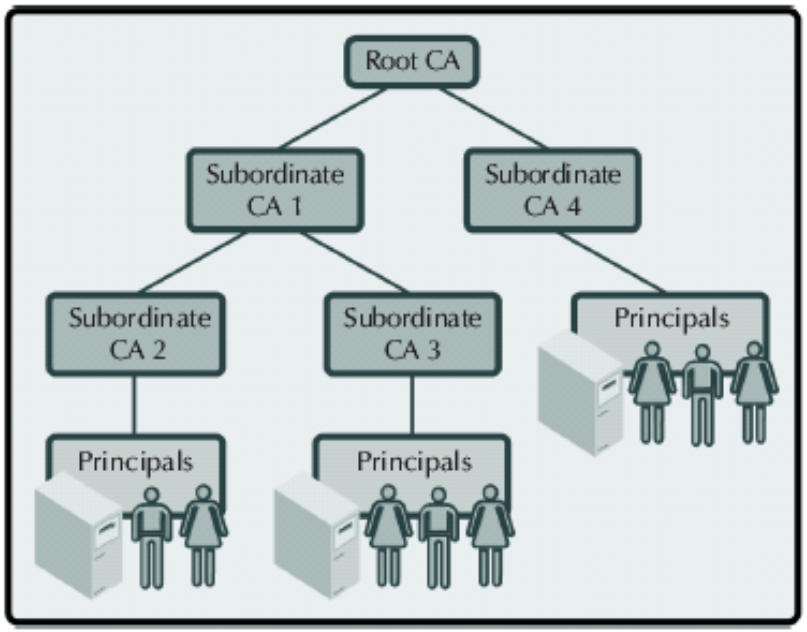
    (1) Forward Certificates: 其他CA发给它的certs

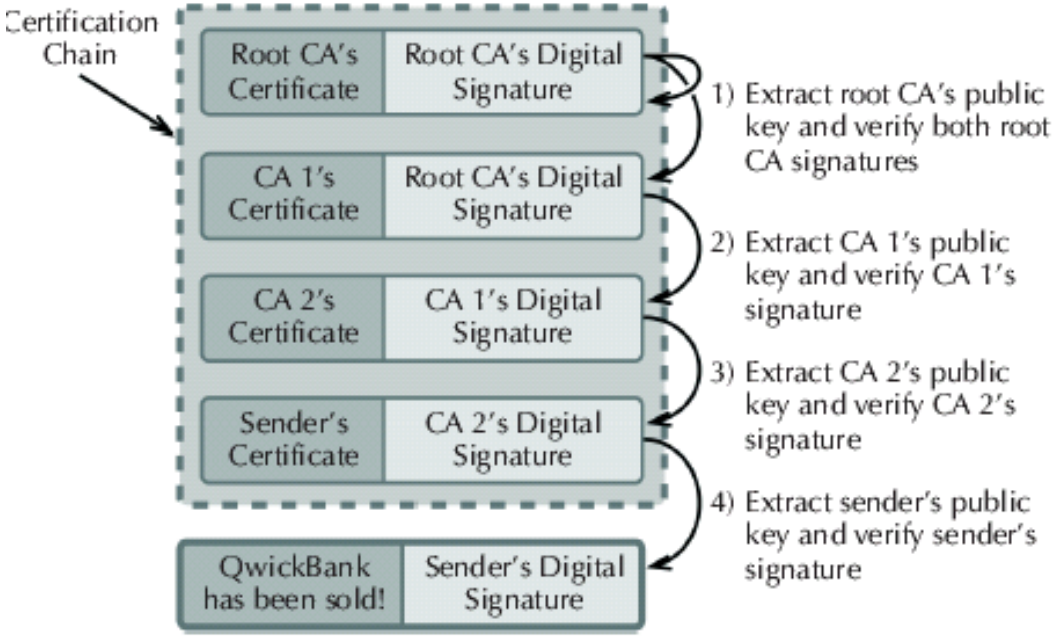
    (2) Reverse Certificates: 它发给其他CA的certs

层次结构CA中证书的验证

- 假设个体A看到B的一个证书
- B的证书中含有签发该证书的CA的信息
- 沿着层次树往上找，可以构成一条证书链，直到根证书
- 验证过程：
  - 沿相反的方向，从根证书开始，依次往下验证每一个证书中的签名。其中，根证书是自签名的，用它自己的公钥进行验证
  - 一直到验证B的证书中的签名
  - 如果所有的签名验证都通过，则A可以确定所有的证书都是正确的，如果他信任根CA，则他可以相信B的证书和公钥

证书链的验证示例


Root CA
Subordinate CA 1
Subordinate CA 4
Subordinate CA 2
Subordinate CA 3
Principals
Principals
Principals


Certification Chain
Root CA's Certificate
Root CA's Digital Signature
CA 1's Certificate
Root CA's Digital Signature
CA 2's Certificate
CA 1's Digital Signature
Sender's Certificate
CA 2's Digital Signature
QwickBank has been sold!
Sender's Digital Signature
1) Extract root CA's public key and verify both root CA signatures
2) Extract CA 1's public key and verify CA 1's signature
3) Extract CA 2's public key and verify CA 2's signature
4) Extract sender's public key and verify sender's signature

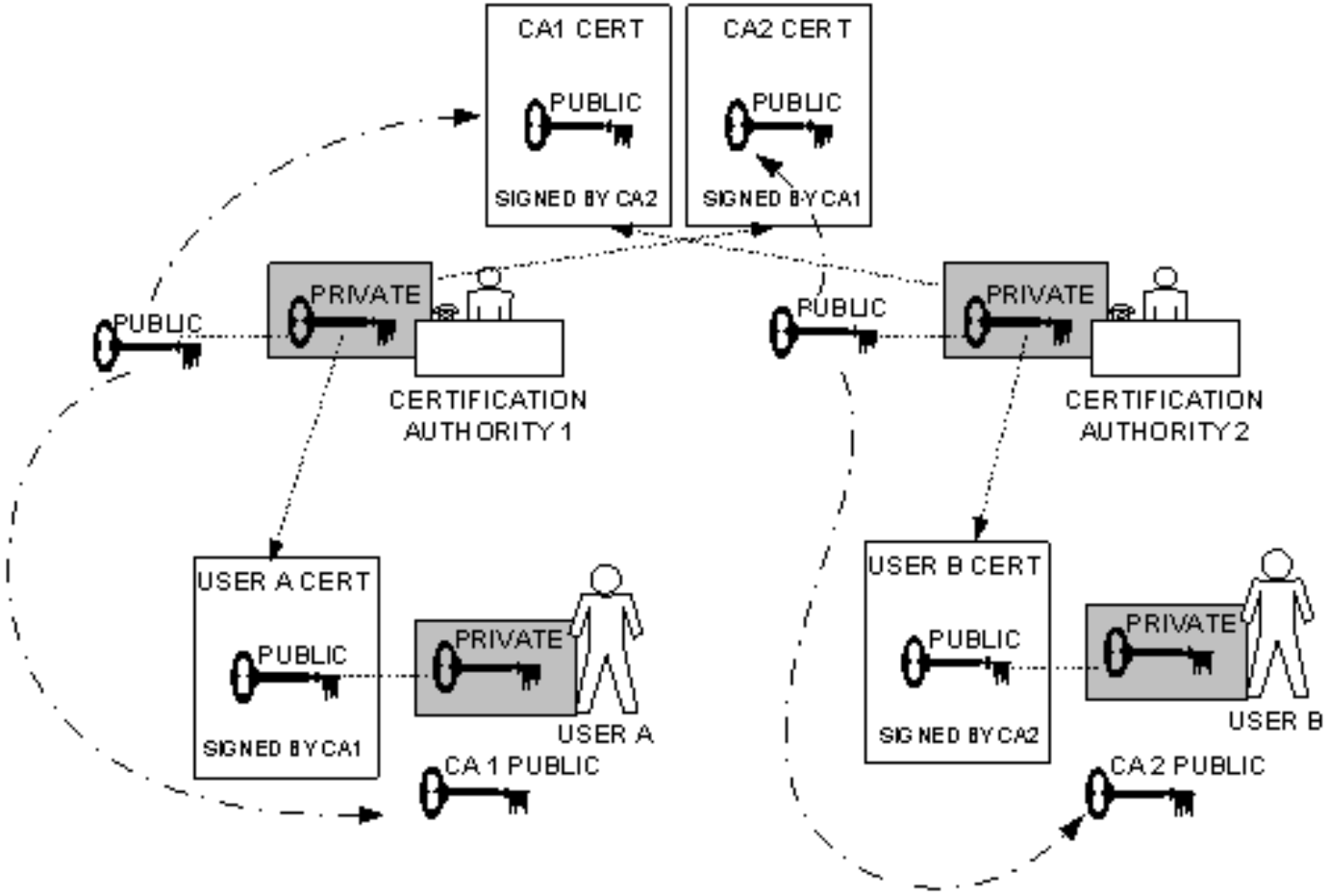
交叉认证

- 两个不同的CA层次结构之间可以建立信任关系
  - 单向交叉认证
    - 一个CA可以承认另一个CA在一定名字空间范围内的所有被授权签发的证书
  - 双向交叉认证
- 交叉认证可以分为
  - 域内交叉认证(同一个层次结构内部)
  - 域间交叉认证(不同的层次结构之间)
- 交叉认证的约束
  - 名字约束
  - 路径长度约束
  - 策略约束

交叉认证

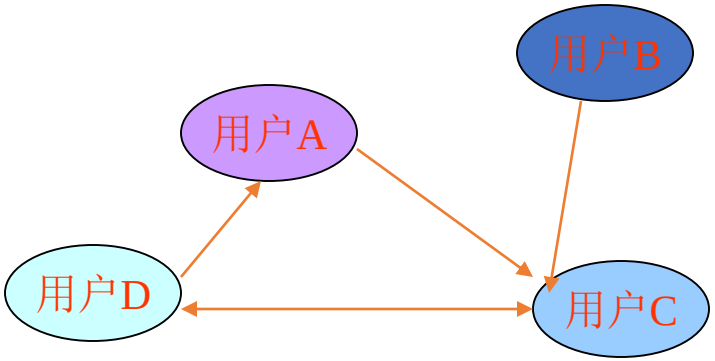
- 交叉认证是把以前无关的CA连接到一起的认证机制
  - 当两者隶属于不同的CA时，可以通过信任传递的机制来完成两者信任关系的建立。
- CA签发交叉认证证书是为了形成非层次的信任路径
  - 一个双边信任关系需要两个证书，它们覆盖每一方向中的信任关系
  - 这些证书必须由CA之间的交叉认证协议来支持。当某证书被证明是假的或者令人误解的时候，该协议将决定合作伙伴的责任

交叉认证

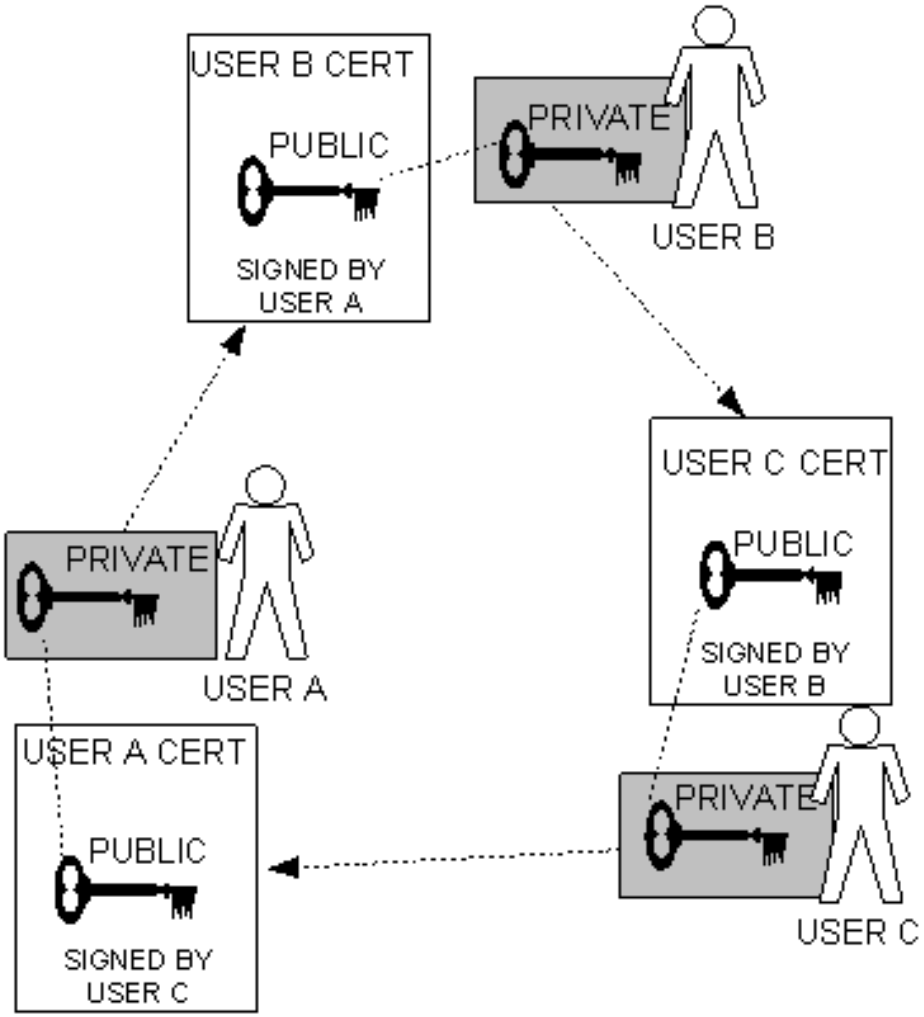

CA1 CERT
PUBLIC
SIGNED BY CA2
CA2 CERT
PUBLIC
SIGNED BY CA1
PUBLIC
PRIVATE
CERTIFICATION AUTHORITY 1
PUBLIC
PRIVATE
CERTIFICATION AUTHORITY 2
USER A CERT
PUBLIC
SIGNED BY CA1
PRIVATE
USER A
CA 1 PUBLIC
USER B CERT
PUBLIC
SIGNED BY CA2
PRIVATE
USER B
CA 2 PUBLIC

以用户为中心的认证

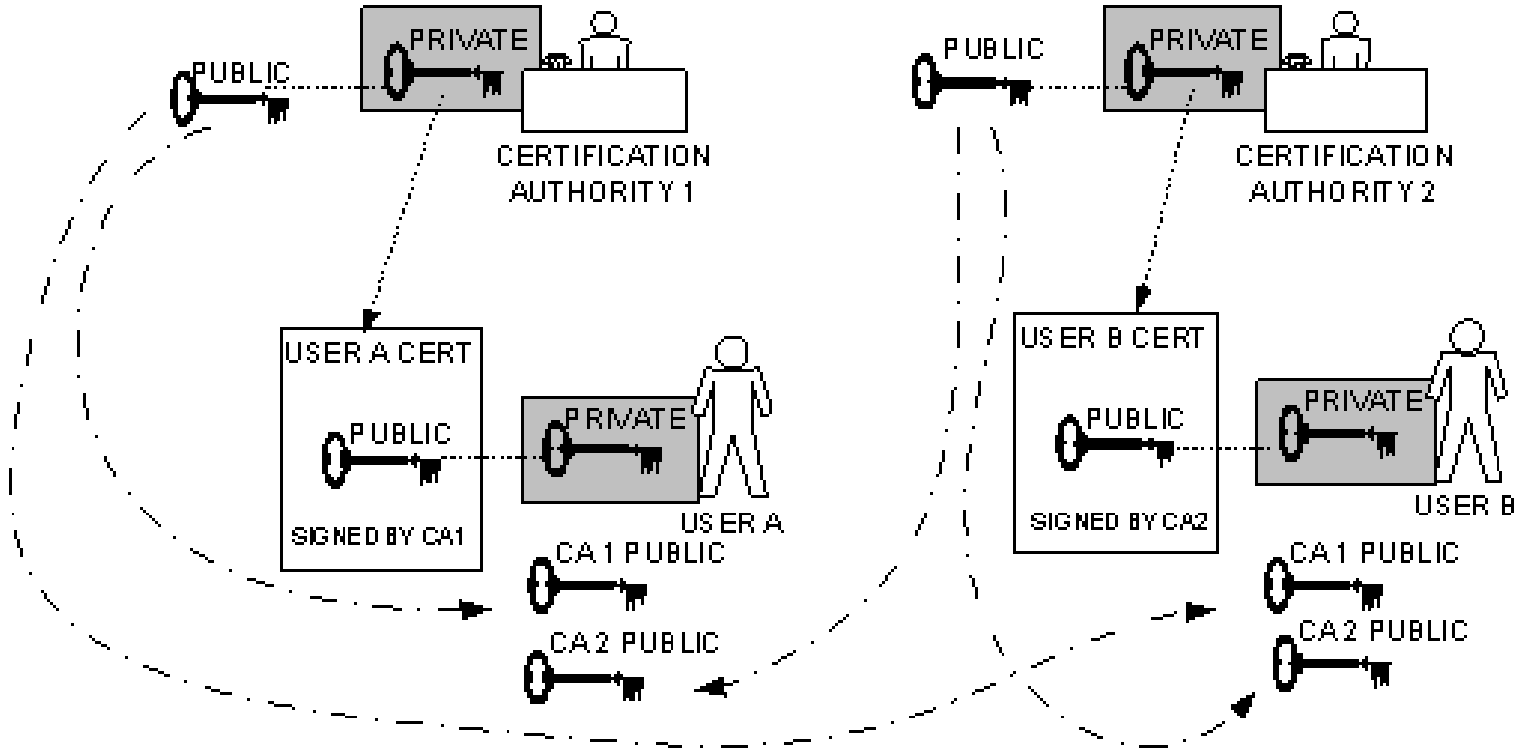
- 不存在集中式CA,用户之间相互颁发证书
- 每个用户为自己颁发的证书提供担保
- 自组织认证模型
  - eg: PGP
- 信任关系按照自然人的信任关系传播



以用户为中心的认证


USER B CERT
PUBLIC
SIGNED BY
USER A
PRIVATE
USER B
USER C CERT
PUBLIC
SIGNED BY
USER B
PRIVATE
USER A
USER A CERT
PUBLIC
SIGNED BY
USER C
PRIVATE
USER C

浏览器信任列表认证


PUBLIC
PRIVATE
CERTIFICATION AUTHORITY 1
USER A CERT
PUBLIC
SIGNED BY CA1
PRIVATE
USER A
CA 1 PUBLIC
CA 2 PUBLIC
PUBLIC
PRIVATE
CERTIFICATION AUTHORITY 2
USER B CERT
PUBLIC
SIGNED BY CA2
PRIVATE
USER B
CA 1 PUBLIC
CA 2 PUBLIC

思考

考虑KDC和CA服务器。

假设KDC发生故障，对各方安全通信的能力有何影响？也就是说，谁可以和不能沟通?

再假设一个CA发生故障。这种失败的影响是什么?

KDC的ticket有几个小时的生命周期，如果kdc故障时间短就还好，如果长时间，用户就不能得到服务。

CA证书周期相对较长，而且ca证书可能预先安装在服务器上，影响可能不大。但有些ca有证书撤销列表，除了验证签名还需要验证一条链，而验证撤销列表的时间是定期的（半小时左右），如果故障的话有问题，但没那么严重