

2550 Intro to cybersecurity

L5

abhi shelat

How U2F foils phishing



User,sk



My browser

1. In the beginning, I register with G and setup 2FA.



User,pk

How U2F foils phishing

2. I am tricked into clicking on fake G login, who tries a PITM attack.



User,sk



My browser

Fake Website

Com-settingssecurity.tk



User,pk

How U2F foils phishing

2. I am tricked into clicking on fake G login, who tries a PITM attack.



User,sk



My browser

Fake Website

Com-settingssecurity.tk



User,pk

{login, challenge ch}

How U2F foils phishing

2. I am tricked into clicking on fake G login, who tries a PITM attack.



User,sk



My browser

Fake Website

Com-settingssecurity.tk



User,pk

{login, challenge ch}

{login, challenge ch}

How U2F foils phishing

2. I am tricked into clicking on fake G login, who tries a PITM attack.



User,sk



My browser

Fake Website

Com-settingssecurity.tk



User,pk

{login, ch, url, tls_id}

{login, challenge ch}

{login, challenge ch}

My browser knows the origin is “com-settingssecurity.tk” instead of google.com, and passes this string as url.

How U2F foils phishing

2. I am tricked into clicking on fake G login, who tries a PITM attack.



User,sk



My browser

Fake Website

Com-settingssecurity.tk



User,pk

$\{\text{login}, \text{ch}, \text{url}, \text{tls_id}\}$

$\{\text{login}, \text{challenge ch}\}$

$\{\text{login}, \text{challenge ch}\}$

My browser knows the origin is “com-settingssecurity.tk” instead of google.com, and passes this string as **url**.

$s \leftarrow \text{Sign}_{sk}(\text{ch}, \text{url}, \text{tls_id})$

Sign challenge using sk

The 2FA key signs this with url=com-settings...

How U2F foils phishing

2. I am tricked into clicking on fake G login, who tries a PITM attack.



User,sk



My browser

Fake Website

Com-settingssecurity.tk



User,pk

$\{\text{login}, ch, \text{url}, \text{tls_id}\}$

$\{\text{login}, \text{challenge } ch\}$

$\{\text{login}, \text{challenge } ch\}$

My browser knows the origin is “com-settingssecurity.tk” instead of google.com, and passes this string as **url**.

$s \leftarrow \text{Sign}_{sk}(ch, \text{url}, \text{tls_id})$

Sign challenge using sk

$\{s\}$

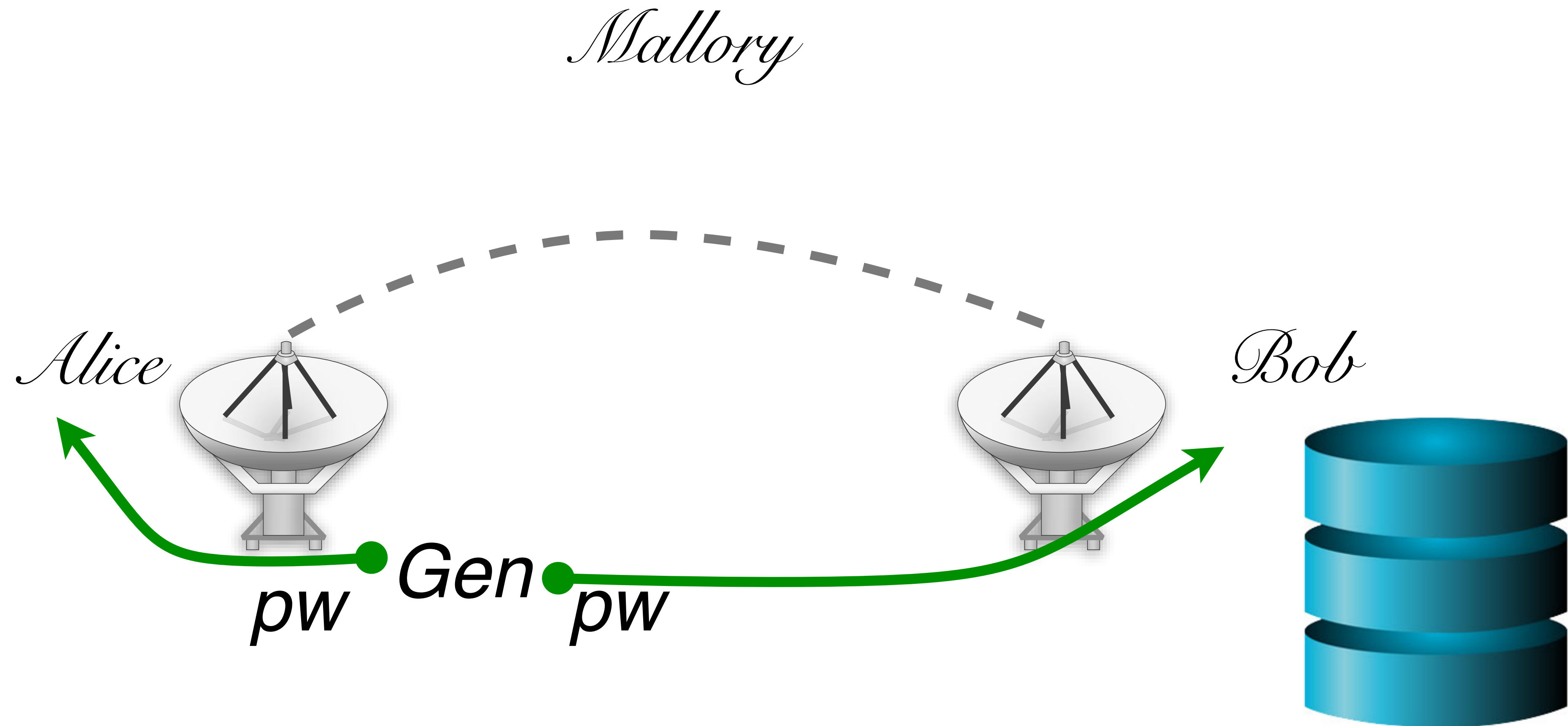
$\text{Verify}_{pk}(ch, s, \text{url}, \text{tls_id})$

The 2FA key signs this with url=com-settings...

Google reject the authentication and detects the attack!

Future without passwords?

Password Security game

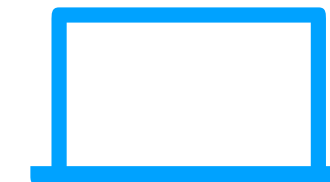


More realistic picture of the world

Alice
pw



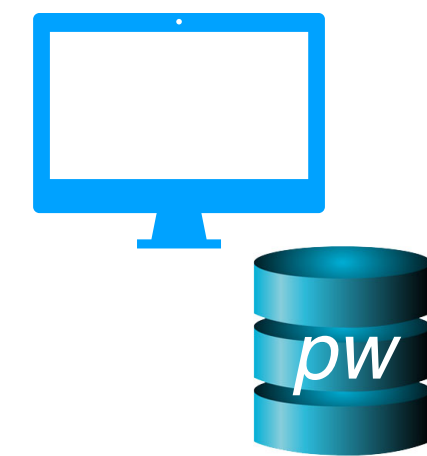
Neu



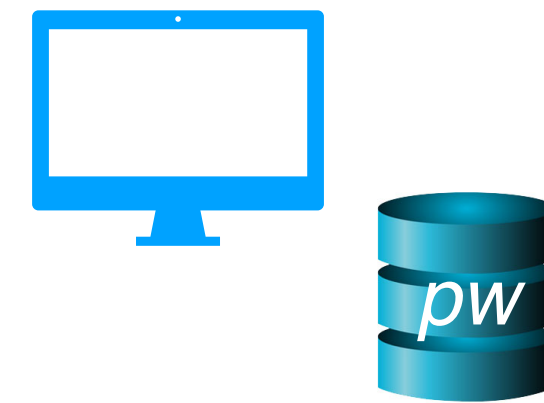
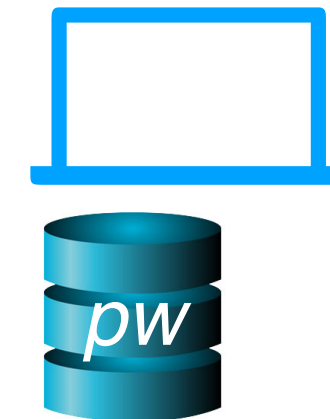
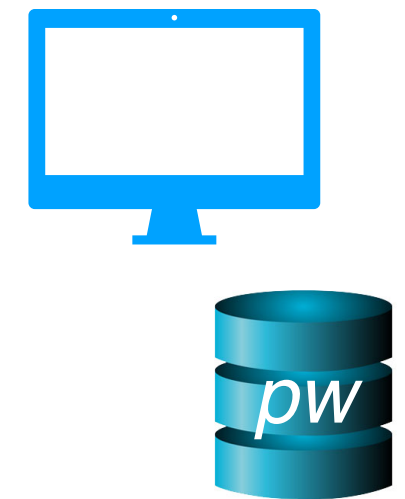
More realistic picture of the world

What are the problems with this solution?

Alice
pw

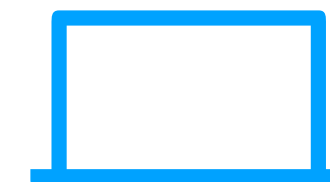
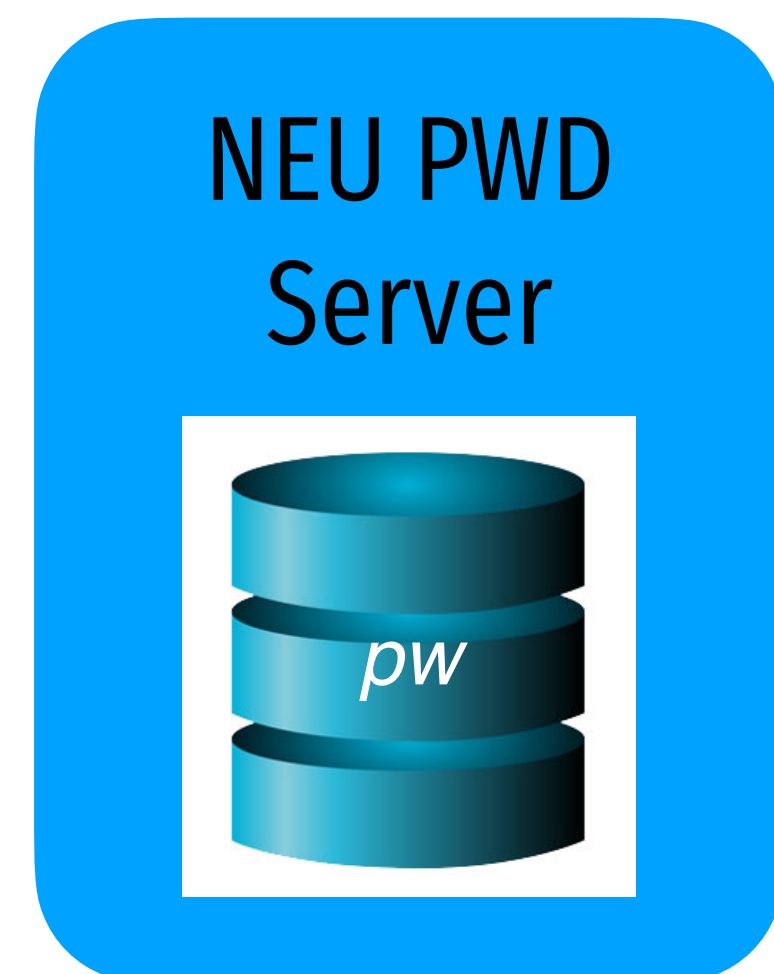


New



The problem of distributed authentication

Alice
pw

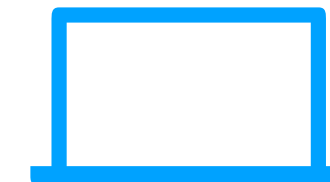
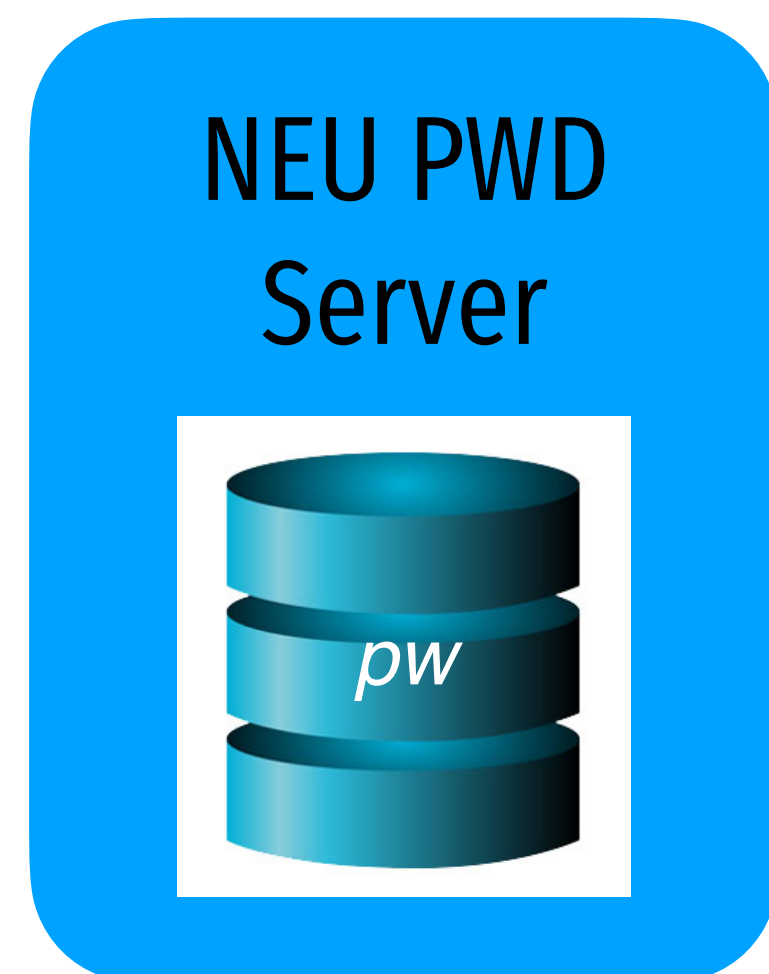


Distributed authentication: Attacker model

What can attacker do?

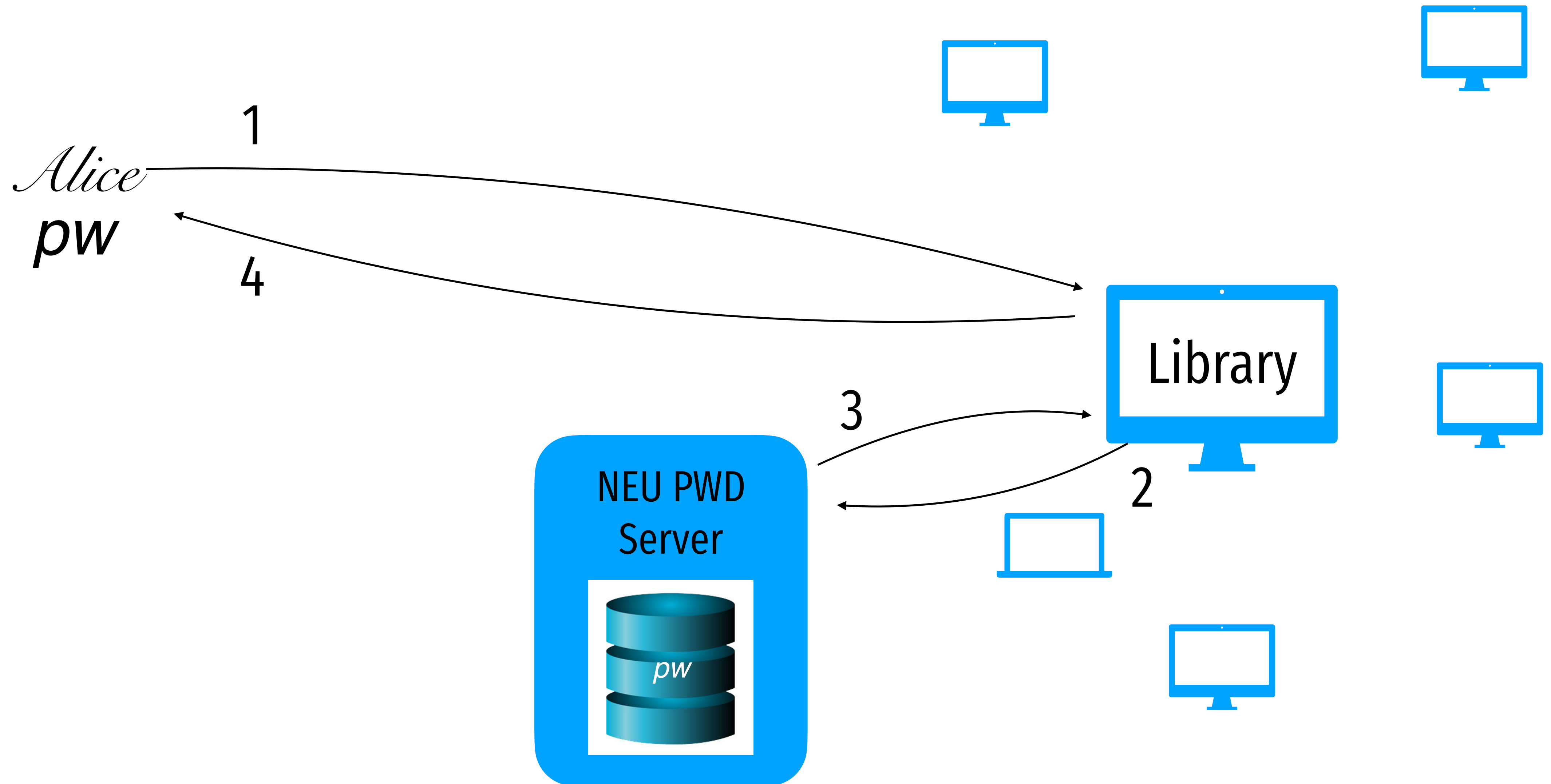


Alice
pw



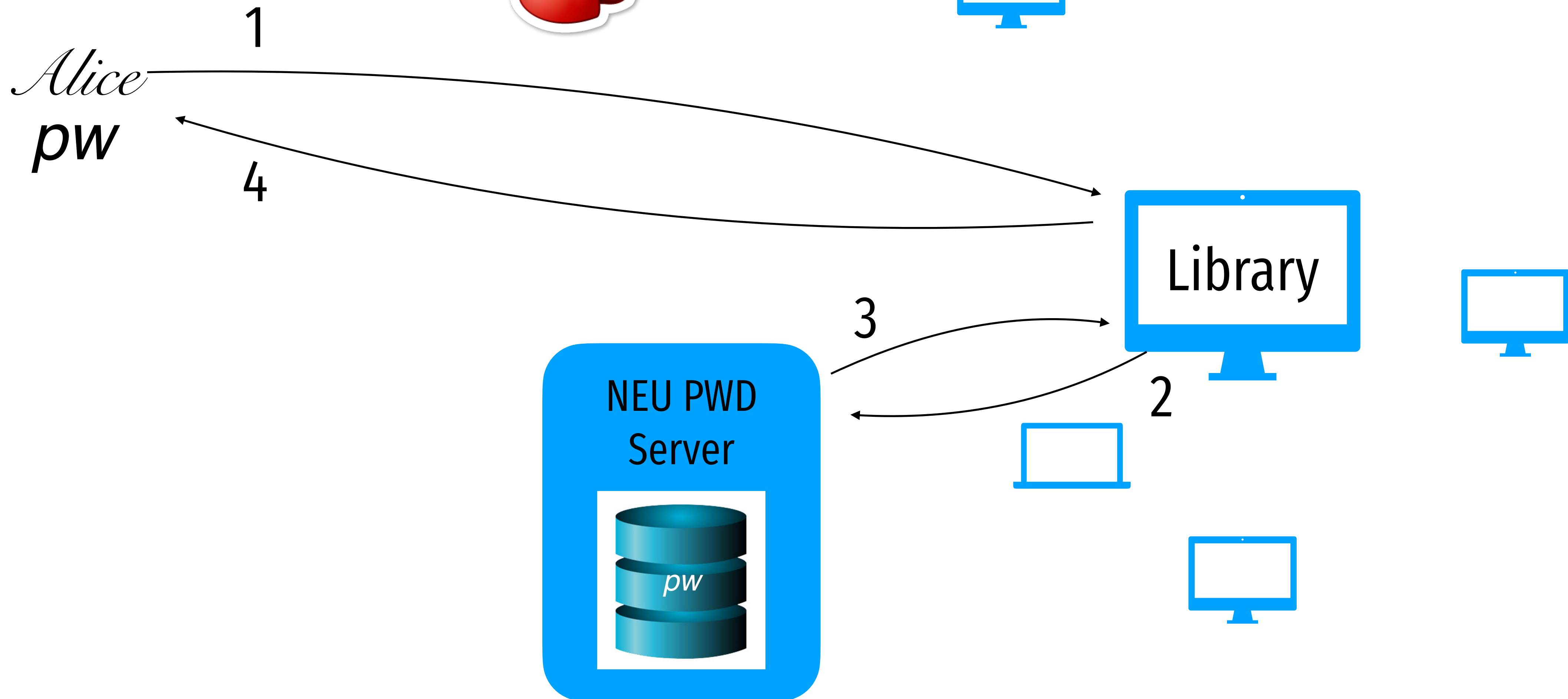
Distributed authentication: Bad Solution

What can attacker do?

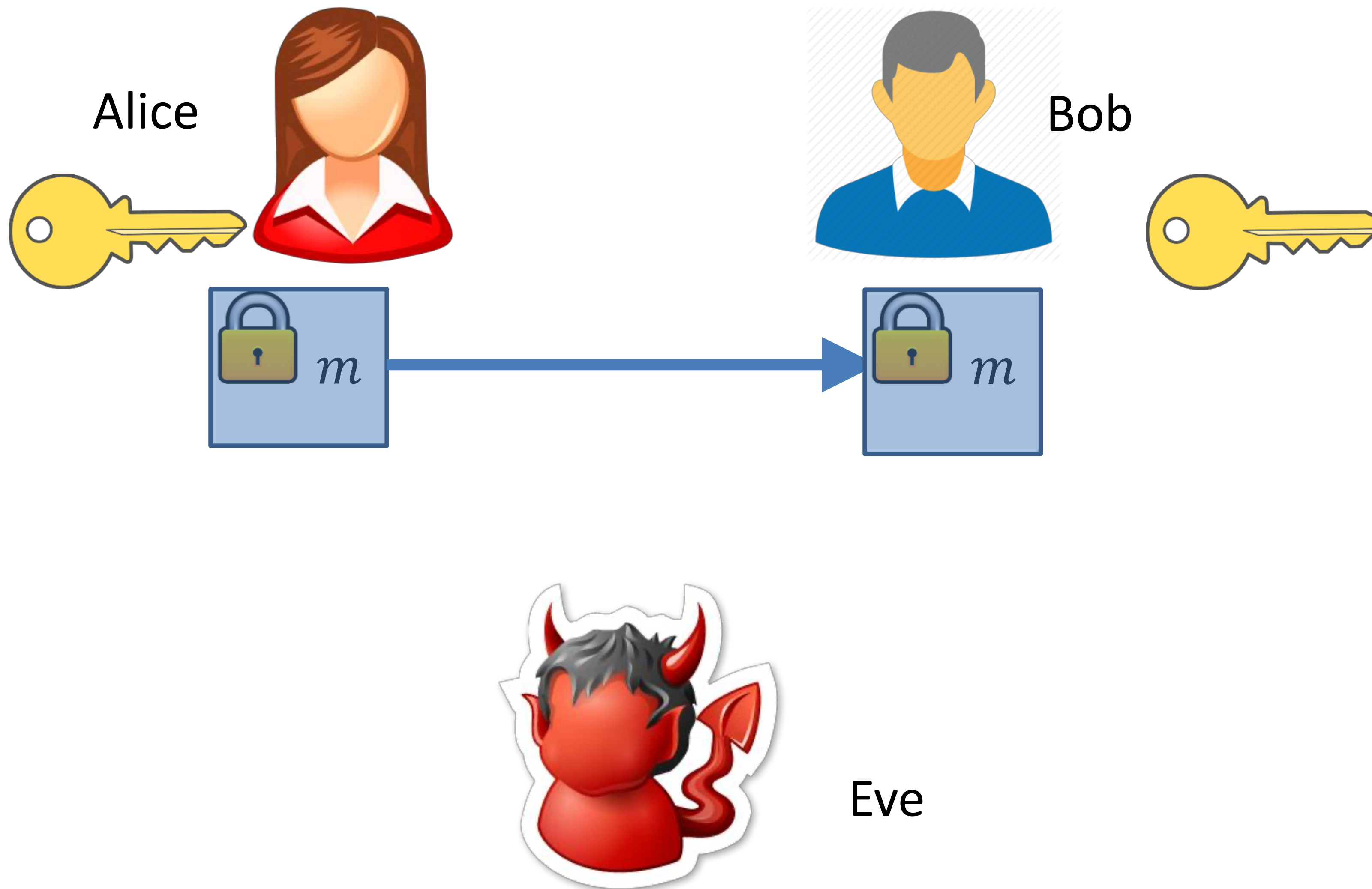


Distributed authentication: Bad Solution

What can attacker do?

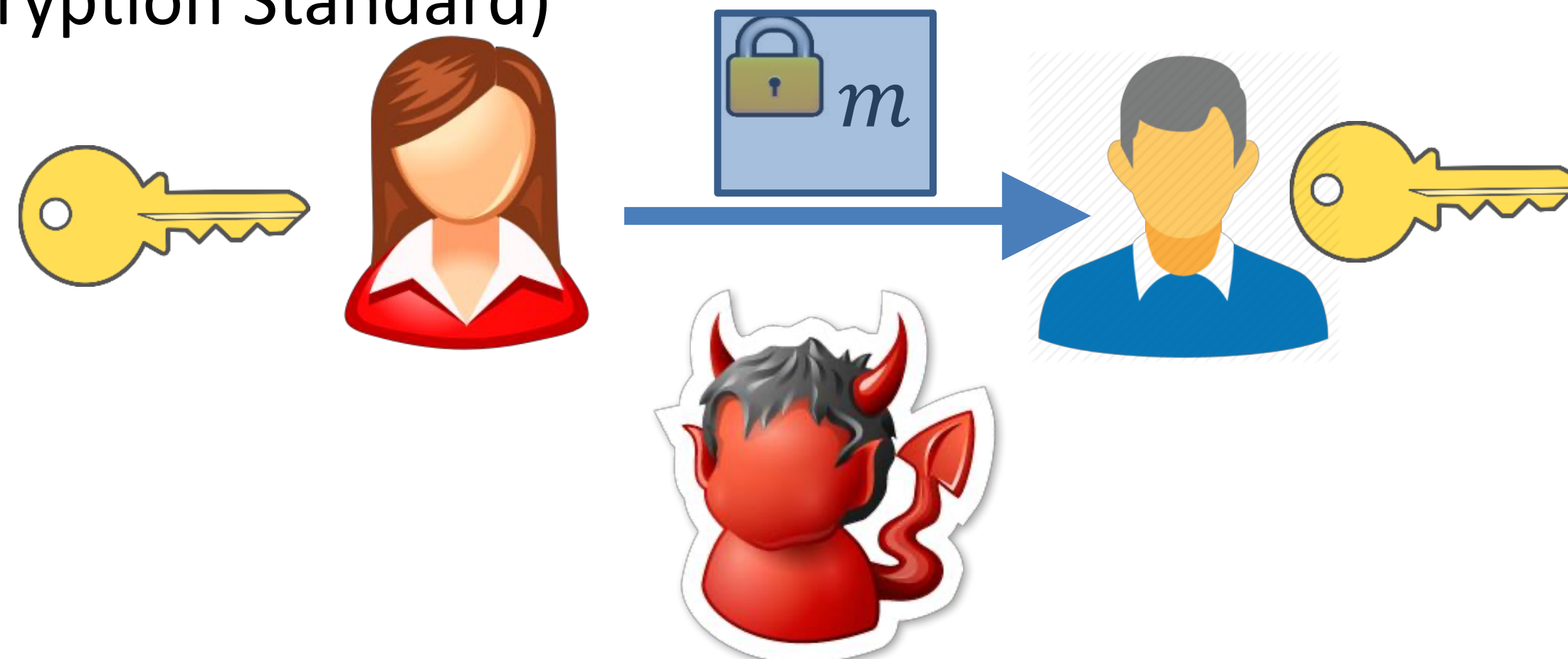


Basic tool: symmetric encryption



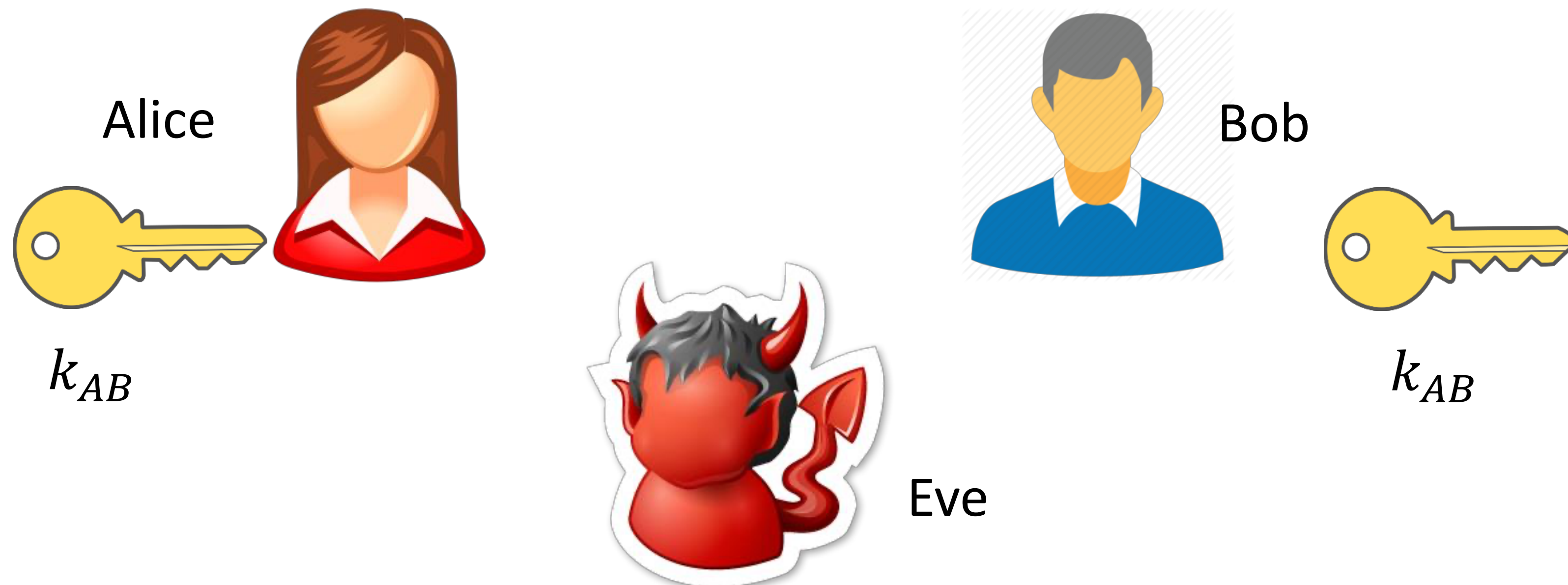
Basic tool: symmetric encryption

- **Gen**: generates secret key k
- **Enc**: given k and m output a ciphertext c
Denote $Enc_k(m)$, $E_k(m)$, $\{m\}_k$
- **Dec**: given k and c output a message m
- Security (informal):
Whatever Eve can learn on m given c can be learned without c
- Examples:
 - DES (Data Encryption Standard)
 - AES (Advanced Encryption Standard)

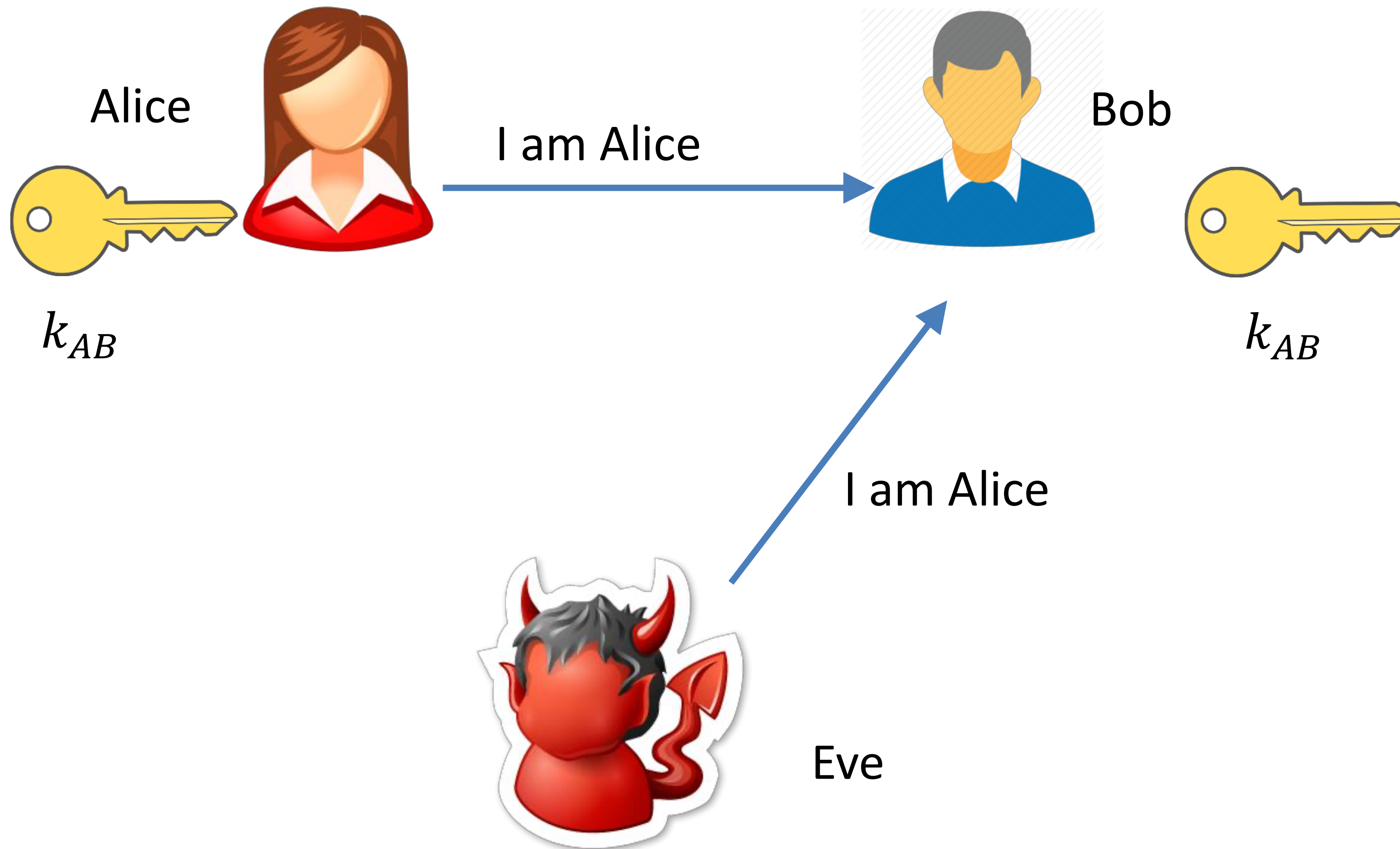


Authentication from Encryption

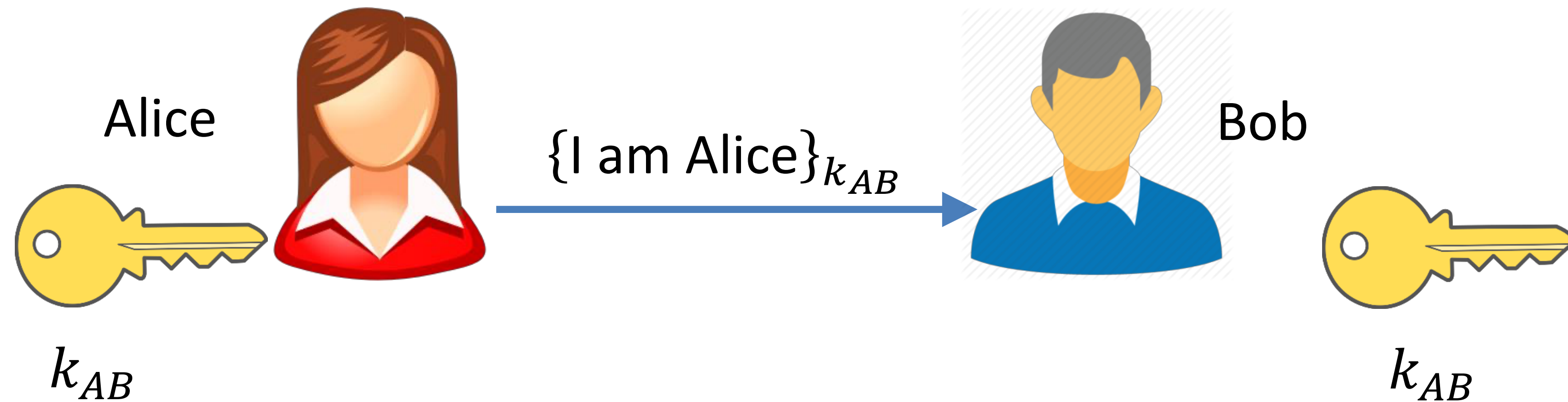
- Alice and Bob share a key
- They communicate over an insecure channel
- Alice wants to prove her identity to Bob
- Eve's goal: impersonate Alice



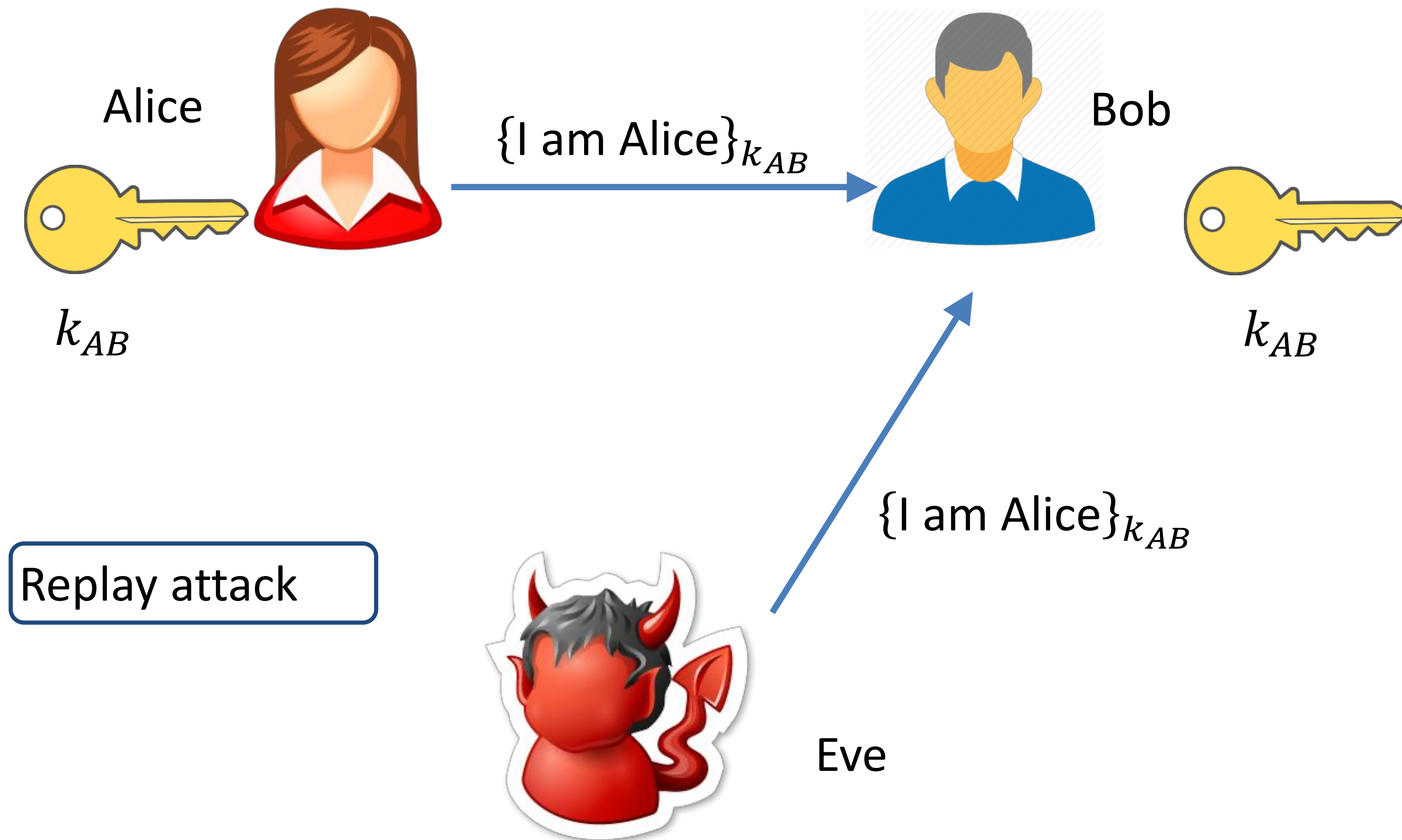
Attempt #1



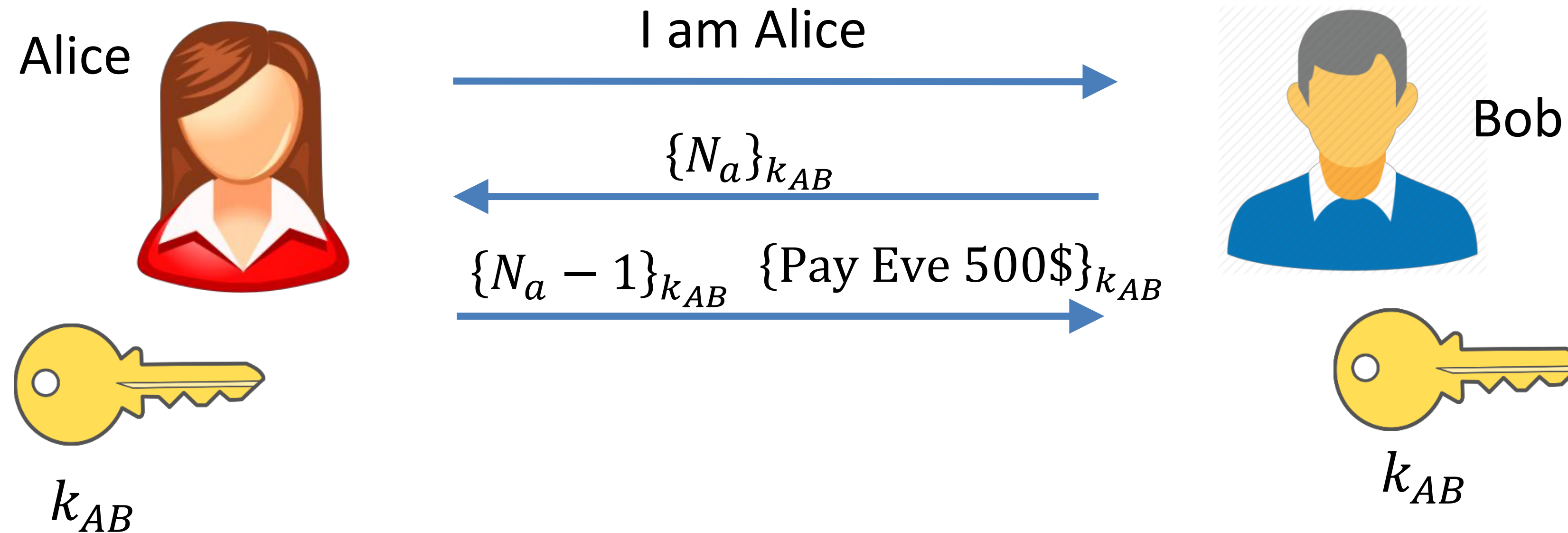
Attempt #2: use the key



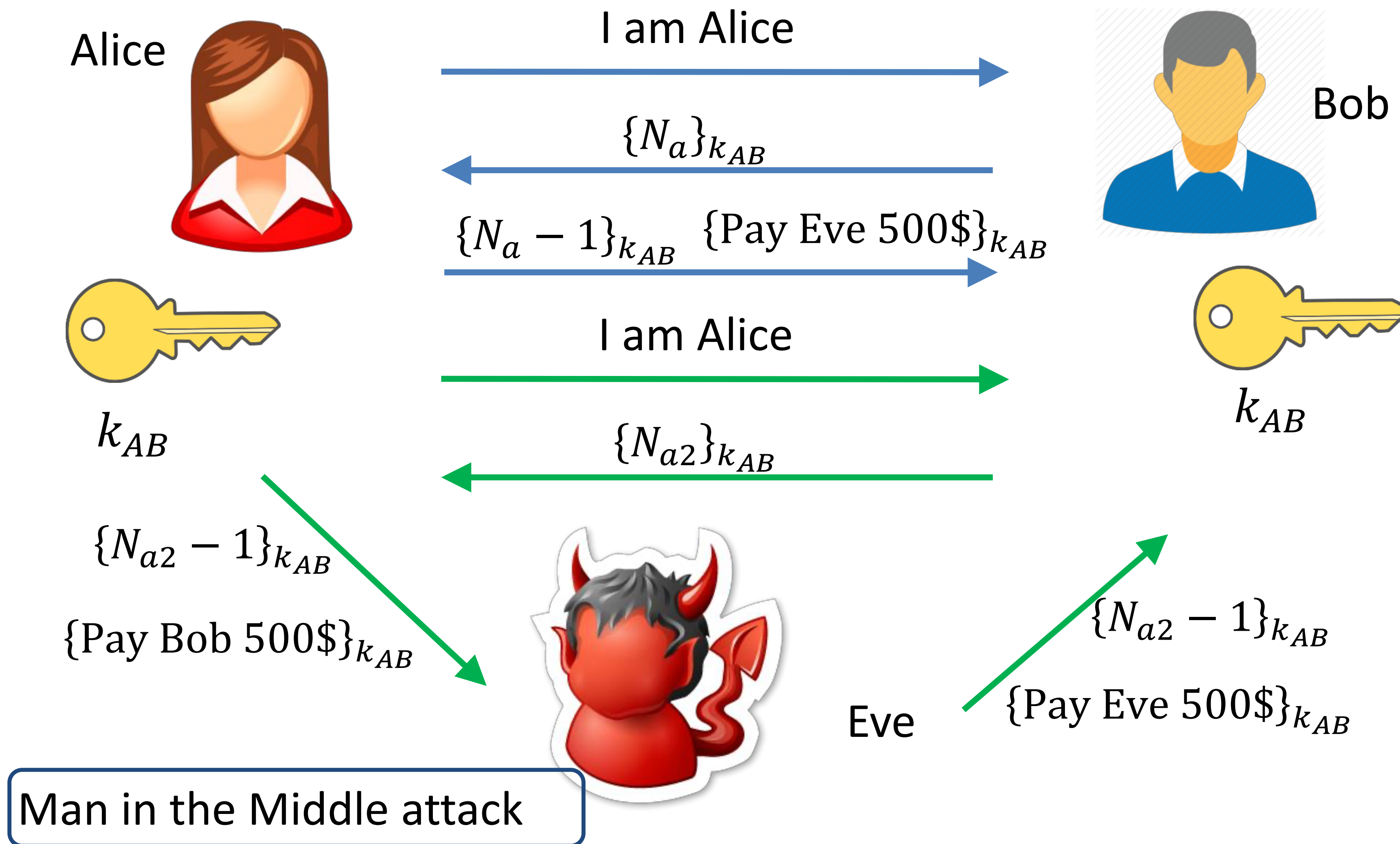
Attempt #2: use the key



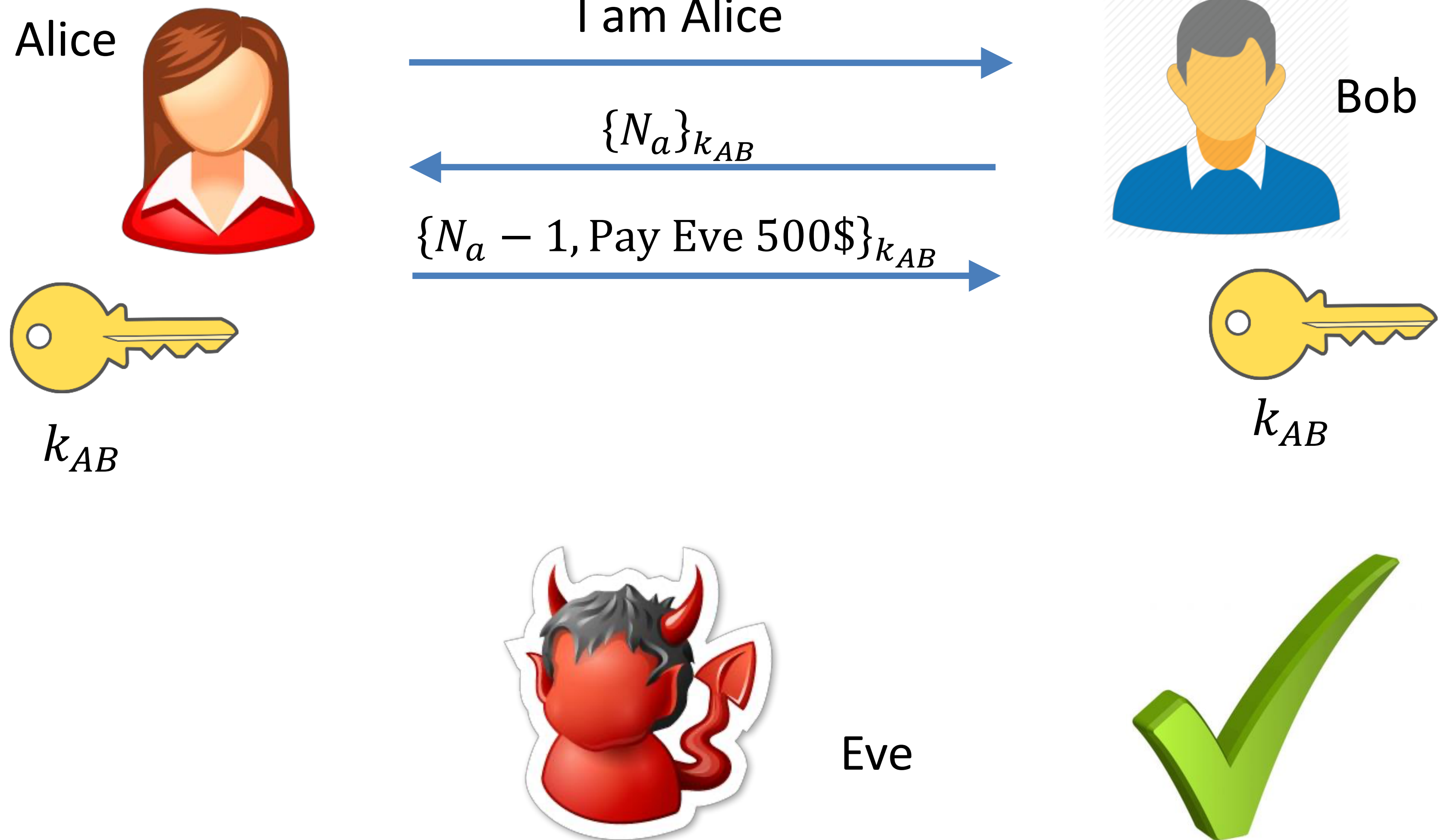
Attempt #3: use nonce



Attempt #3: use nonce



Attempt #4



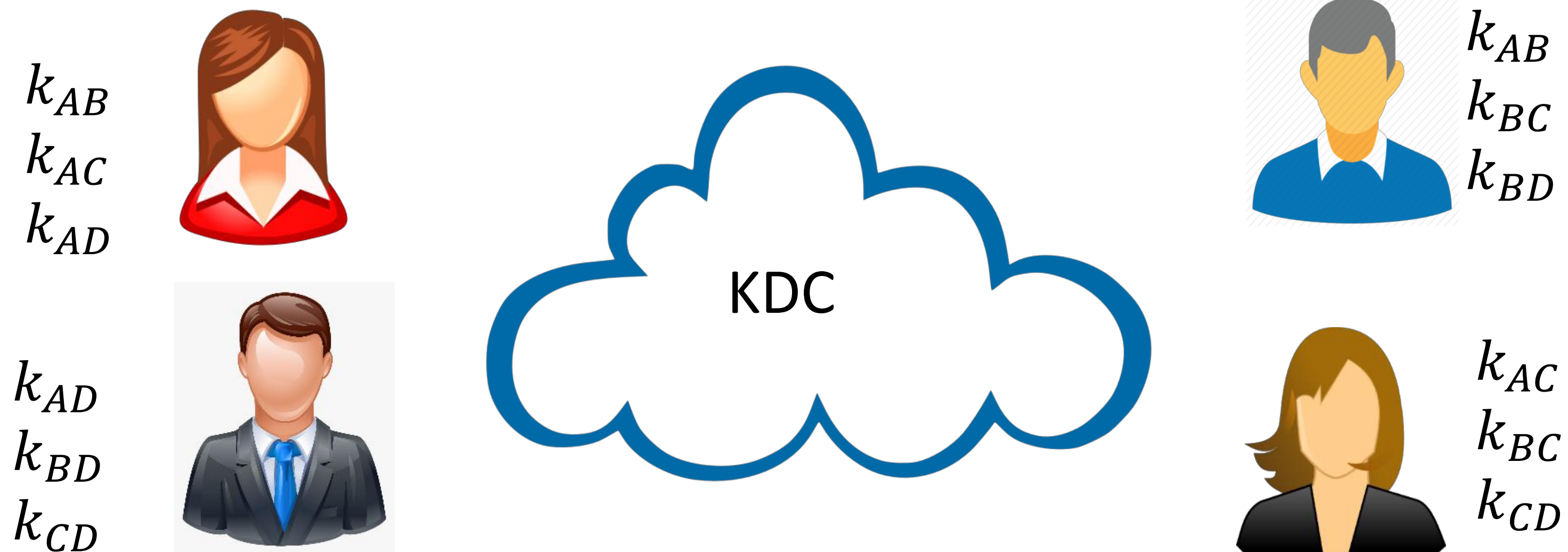
Key establishment

- The protocol worked because Alice and Bob shared a key
- How do parties agree on a key?
 - Run a key agreement protocol (later in the semester)
 - Use a trusted third party (this lecture)
- Key distribution center (KDC):
 - Shares a key with each entity
 - Single point of failure
 - Reasonable assumption for organizations
 - Not useful for open environments (e.g. the Internet)



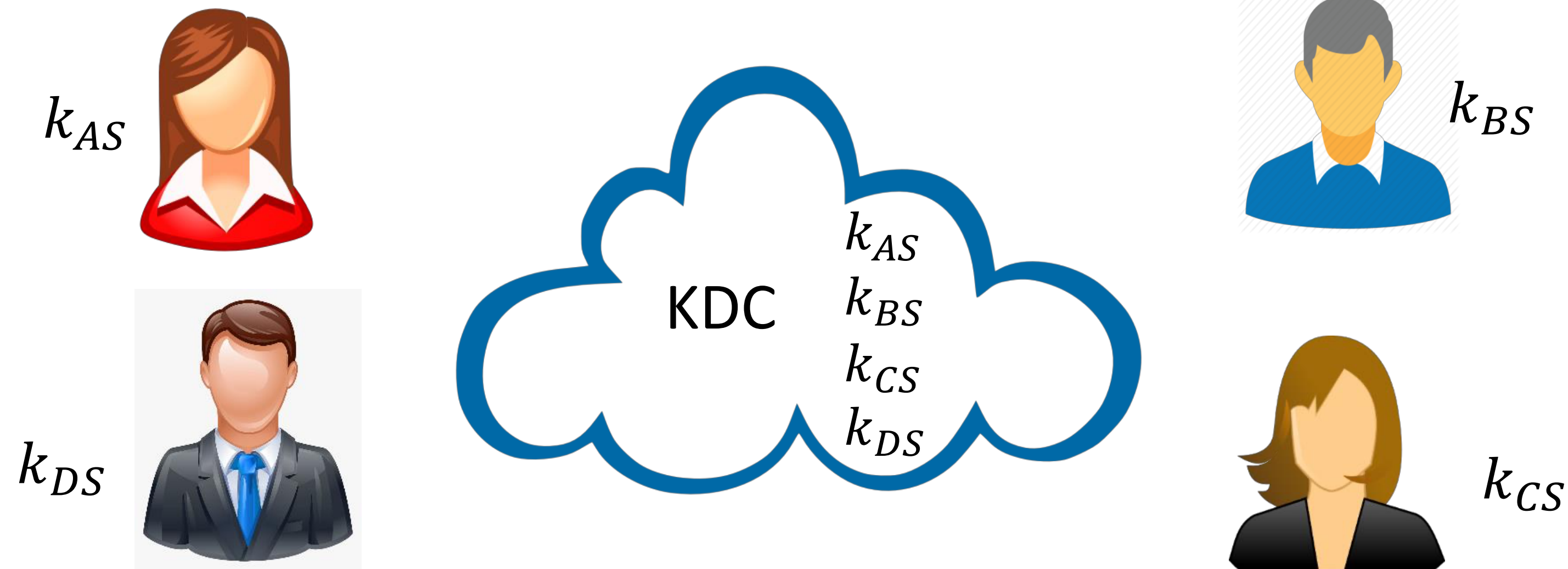
Naïve solution

- KDC generates a key for each pair
- Number of keys $n(n - 1)$, number of key pairs $\frac{n(n-1)}{2} = \binom{n}{2}$
- Drawbacks:
 - Quadratic number of keys
 - Adding new users is complex
- May be useful for static small networks

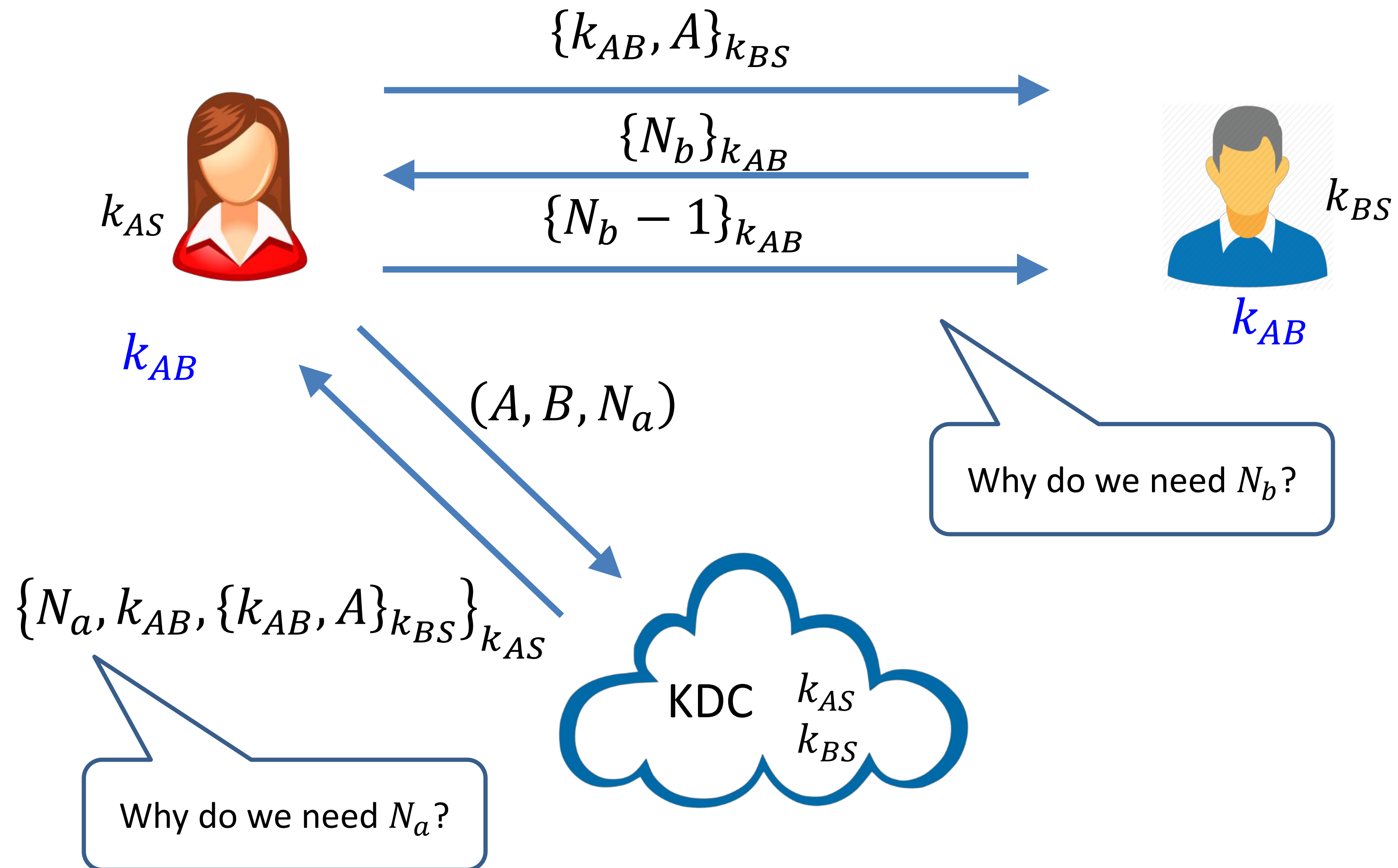


Desire: solution with linear keys

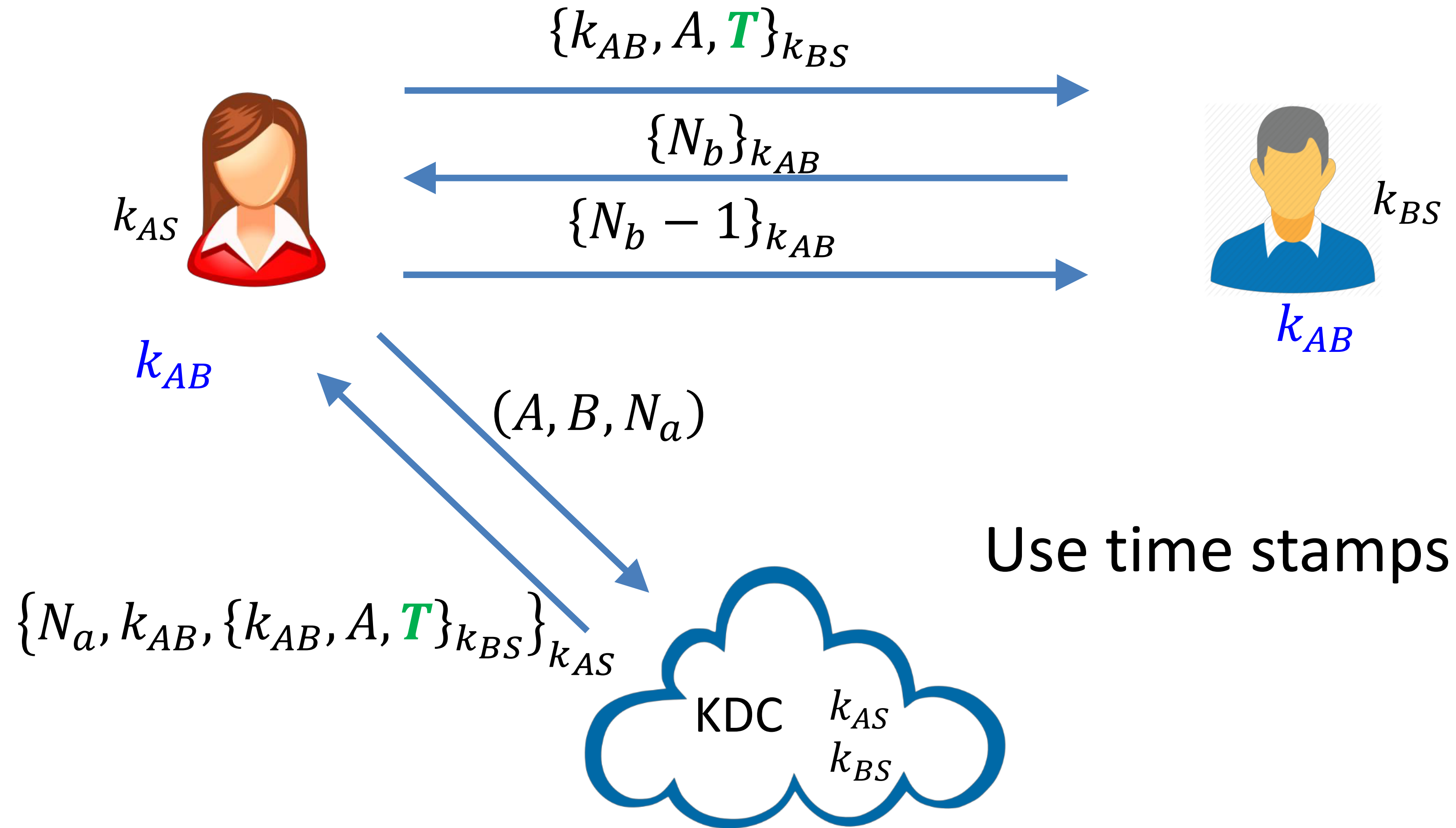
- KDC shares a key with each user
- Number of keys $2n$
- Number of key pairs n
- These are long-term keys
- Alice and Bob establish a fresh session key



Needham-Schroeder Protocol (1978)



Fixed Needham-Schroeder

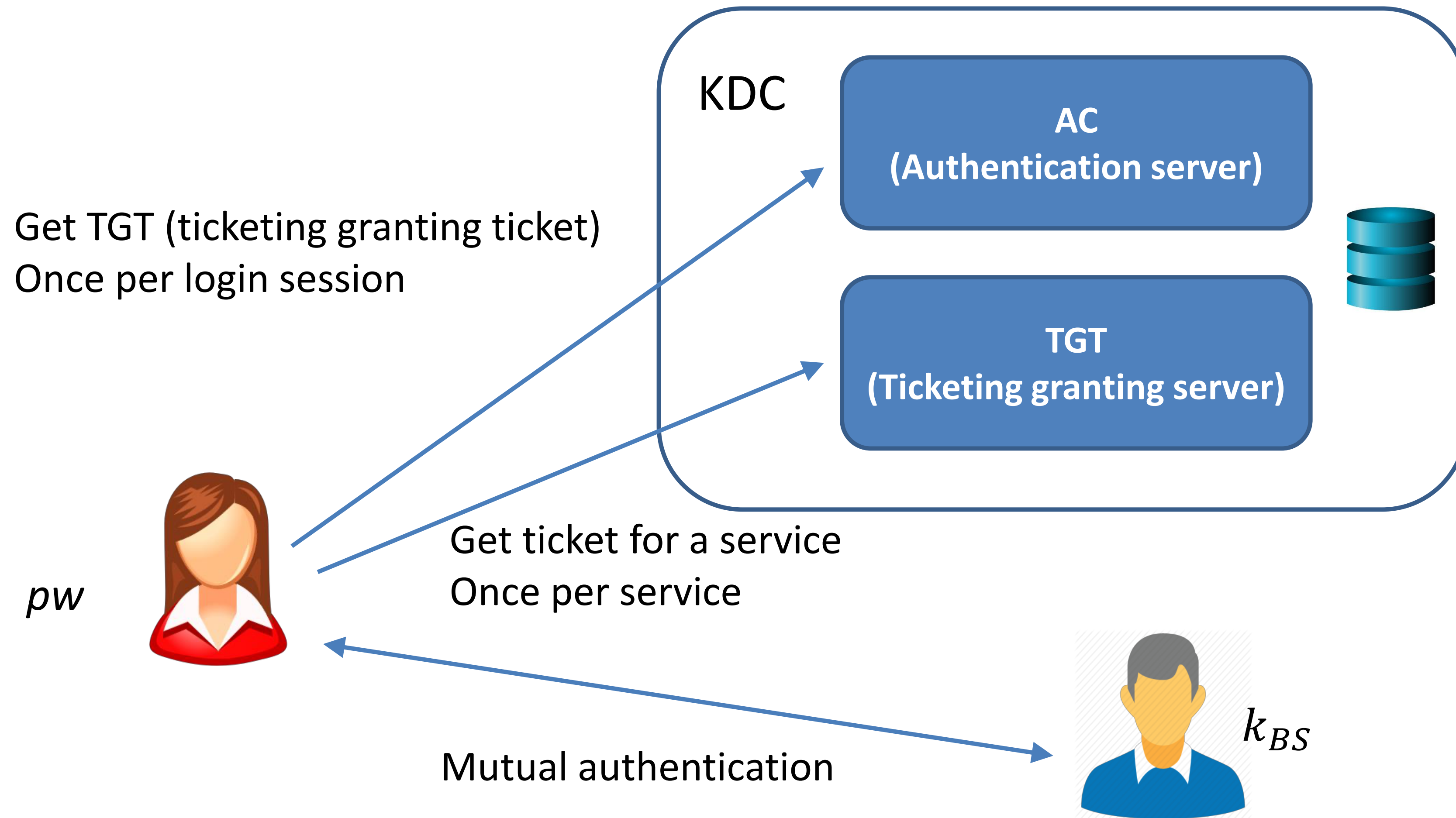


Kerberos

- Developed in MIT in the '80s
- Based on Needham-Schroeder
 - Versions 1-3 not published
 - Version 4 not secure
 - Version 5 published in 1993
- Widely used nowadays:
 - The basis of Microsoft's active directory
 - Many Unix versions



Kerberos



Kerberos

- Passwords are not sent over the network
- Alice's key k_{AS} is a hash of her password
- Kerberos weaknesses:
 - KDC is a single point of failure
 - DoS the KDC and the network ceases to function
 - Compromise the KDC leads to network-wide compromise
 - Time synchronization is a very hard problem

“Single Sign on”

Sign up with your identity provider

You'll use this service to log in to your network

 Sign up with Google

 Sign up with Microsoft

OR

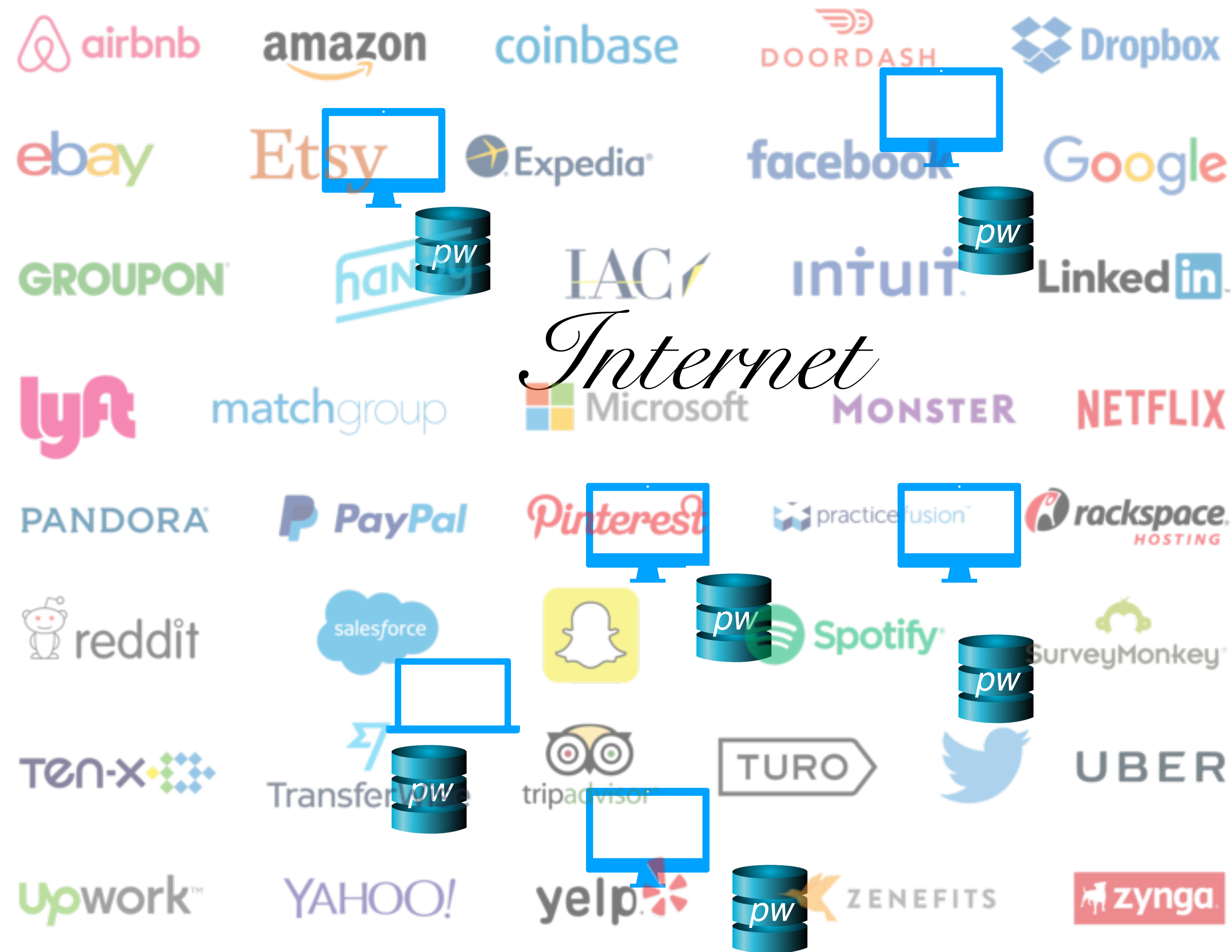
|Enter your email...



Sign up with Email

Same problem as before

Alice
pw



“Single Sign on”

Alice
pw

Sign up with your identity provider

You'll use this service to log in to your network

 Sign up with Google

 Sign up with Microsoft

OR

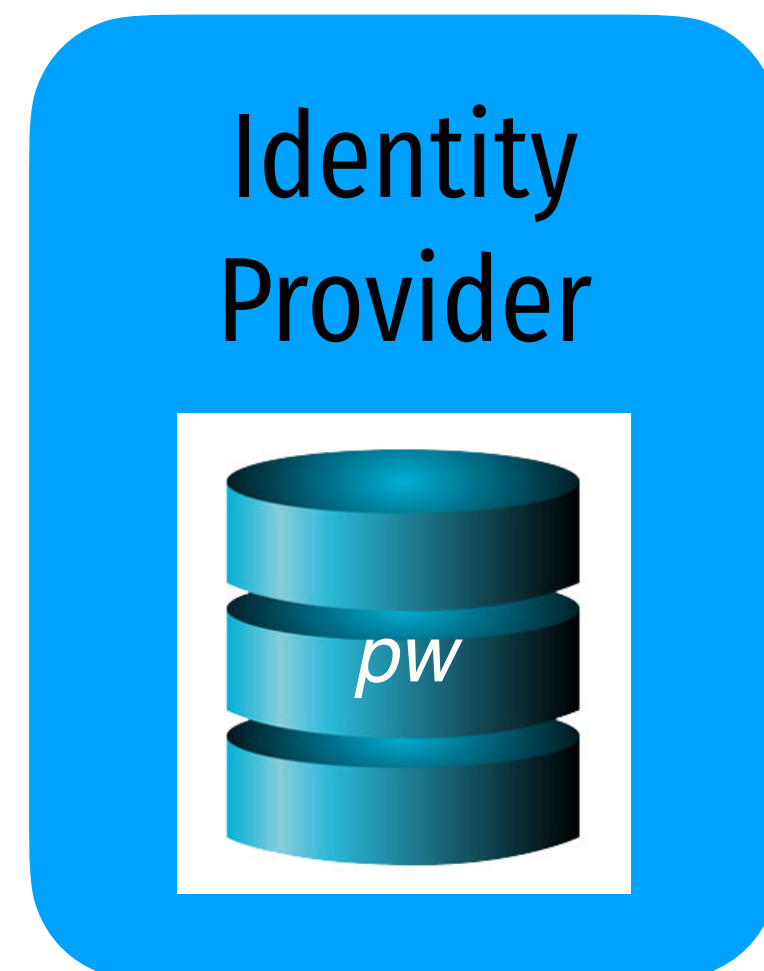


Oauth

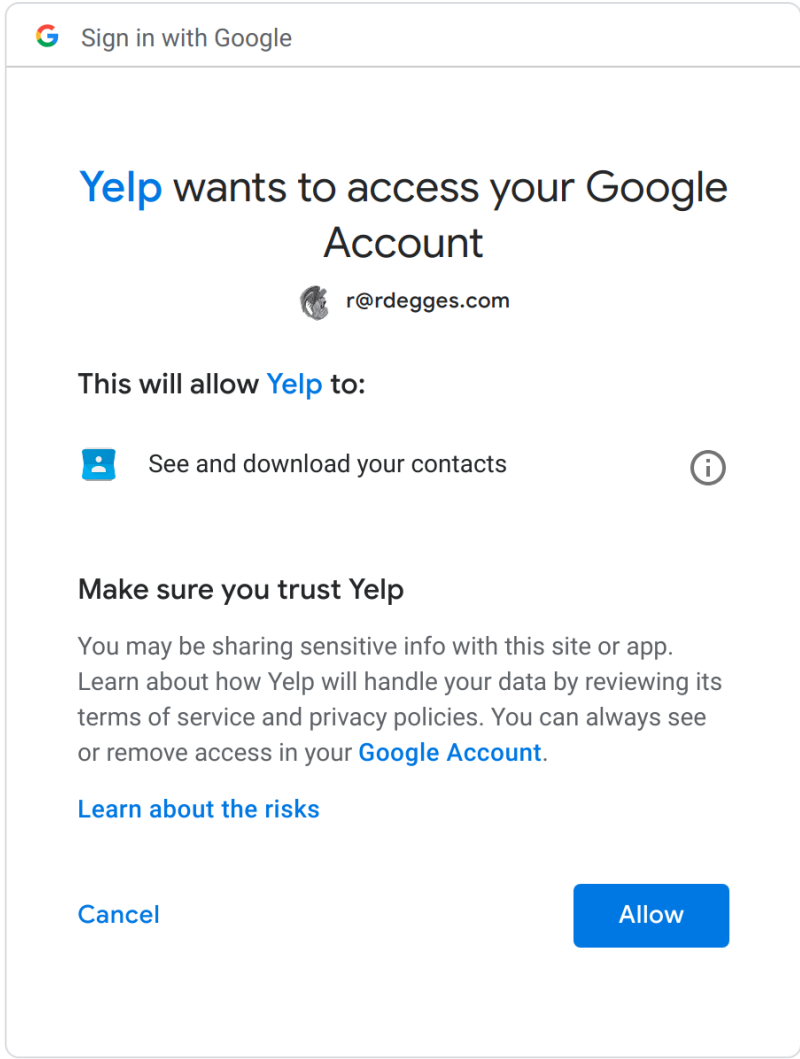
“I want to use your service”



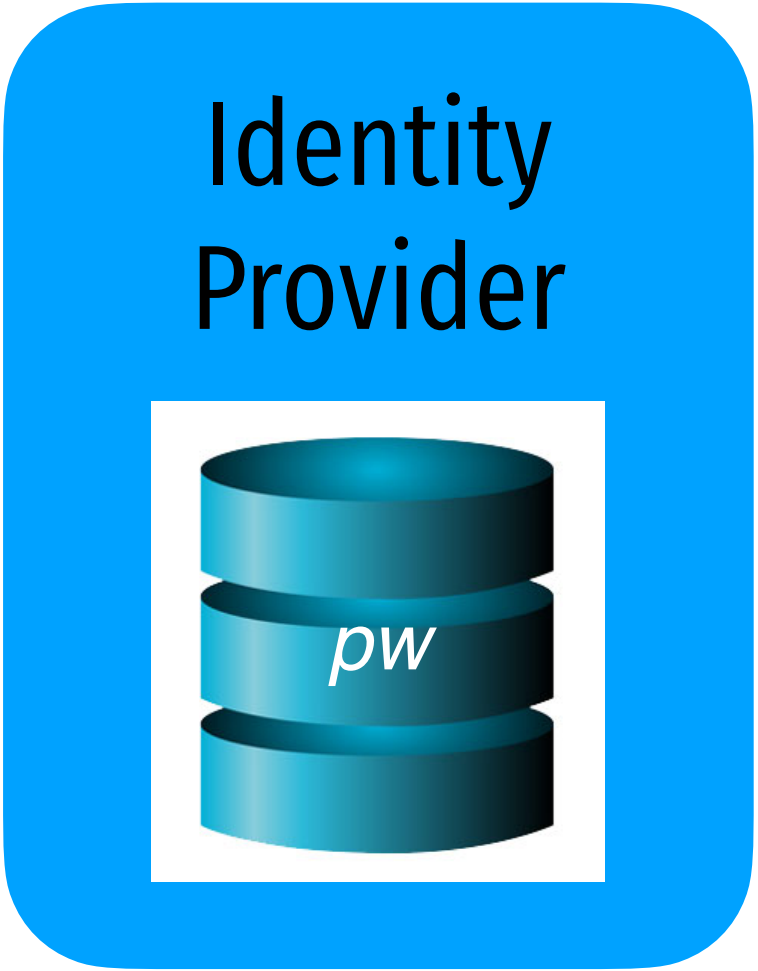
1. Authenticate



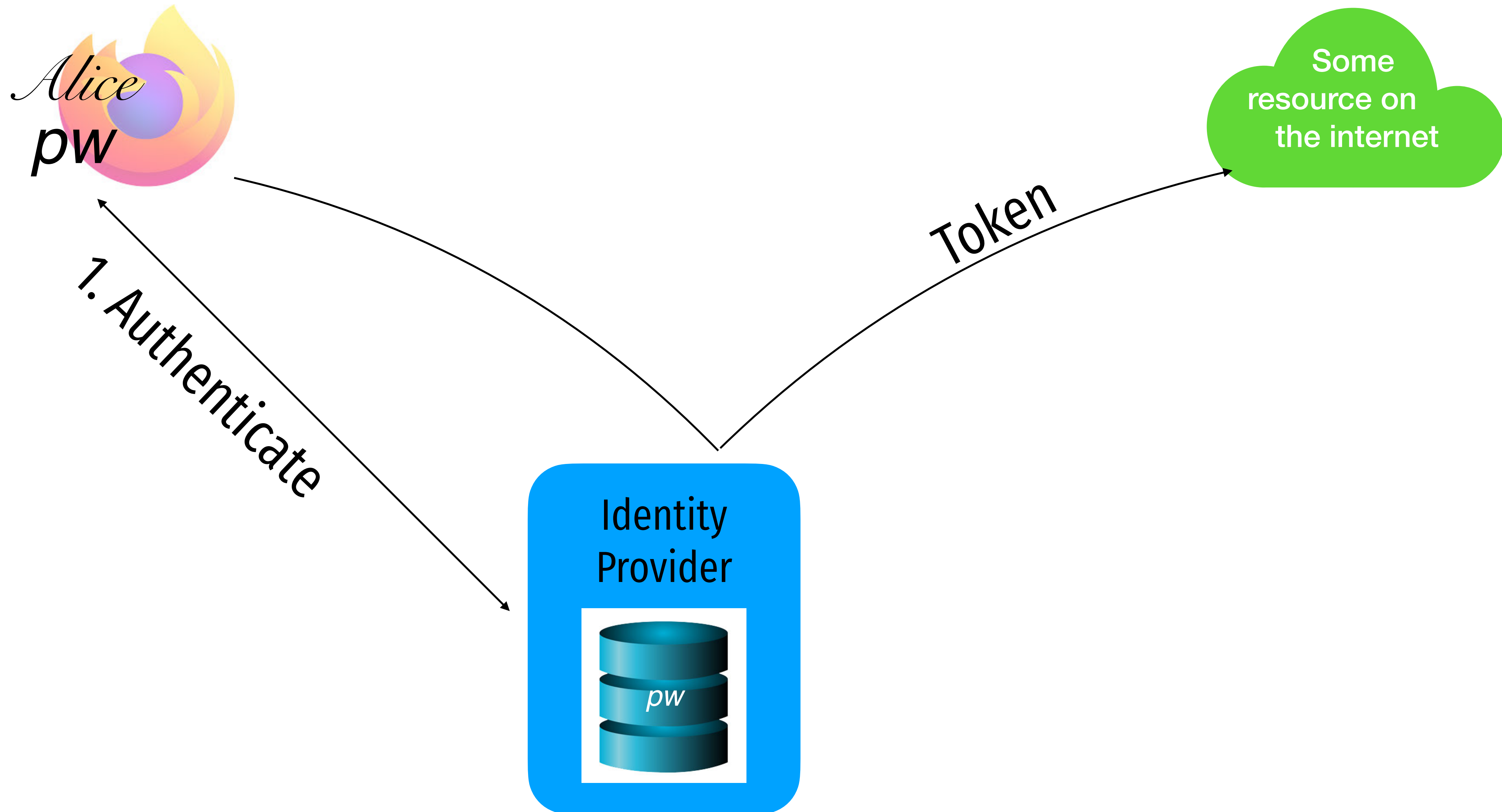
Oauth



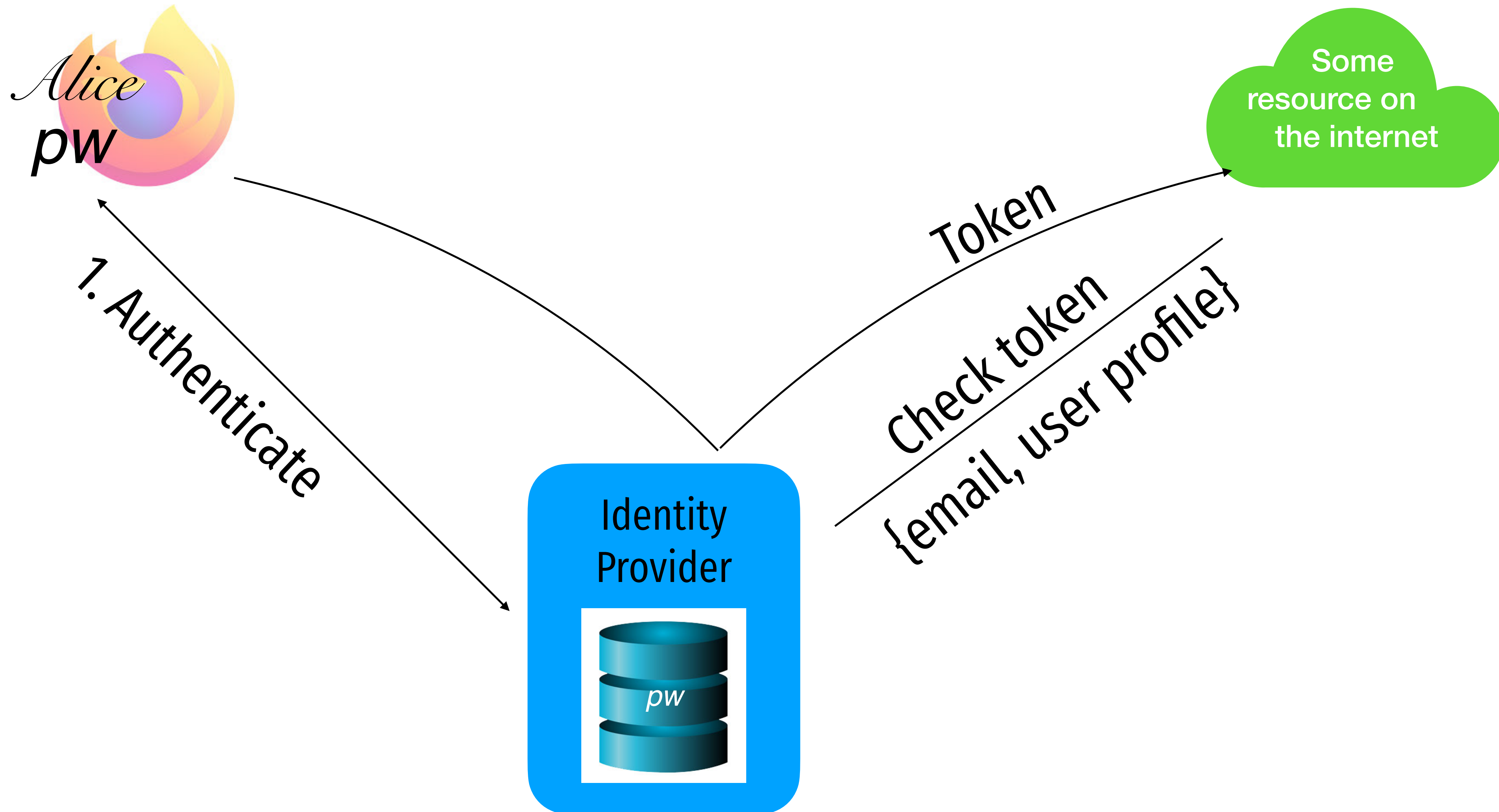
1. Authenticate



Oauth



Oauth



In one breath

The relying party sends the user to the issuer, who checks her identity and sends her back with a one-time code. The relying party trades that code, privately, for a signed ID token that says who she is.



Oauth parties



User

Alice, in her browser

Wants to use PrintShop without creating yet another password.
Acts through her browser.

Spec name: End-User / Resource Owner



Relying Party

l4.neucrypt.org

The app Alice is signing in to. It relies on someone else to tell it who Alice is.

Spec name: Client



Identity Issuer

issuer.example

Already knows Alice and holds her password. Issues signed statements about who she is.

*Spec name: OpenID Provider /
Authorization Server*

The problem OAuth solves

Without it

l4.neucrypt.org asks Alice for her issuer password directly.

l4.neucrypt.org now holds a secret that unlocks everything Alice has at the issuer.

Alice can't limit what l4.neucrypt.org does, and can't revoke it without changing her password.

With it

Alice types her password only at the issuer, on the issuer's own page.

The issuer hands l4.neucrypt.org a signed, short-lived statement: "this is Alice."

l4.neucrypt.org never sees the password. Alice can revoke access at the issuer anytime.



The relying party registers with the issuer

A one-time setup, done by l4.neucrypt.org developers in the issuer's developer console. The issuer hands back three things l4.neucrypt.org will use in every sign-in.



client_id

Public name for l4.neucrypt.org.
Safe to put in URLs.



client_secret

l4.neucrypt.org password to the issuer. Stays on PrintShop's server.

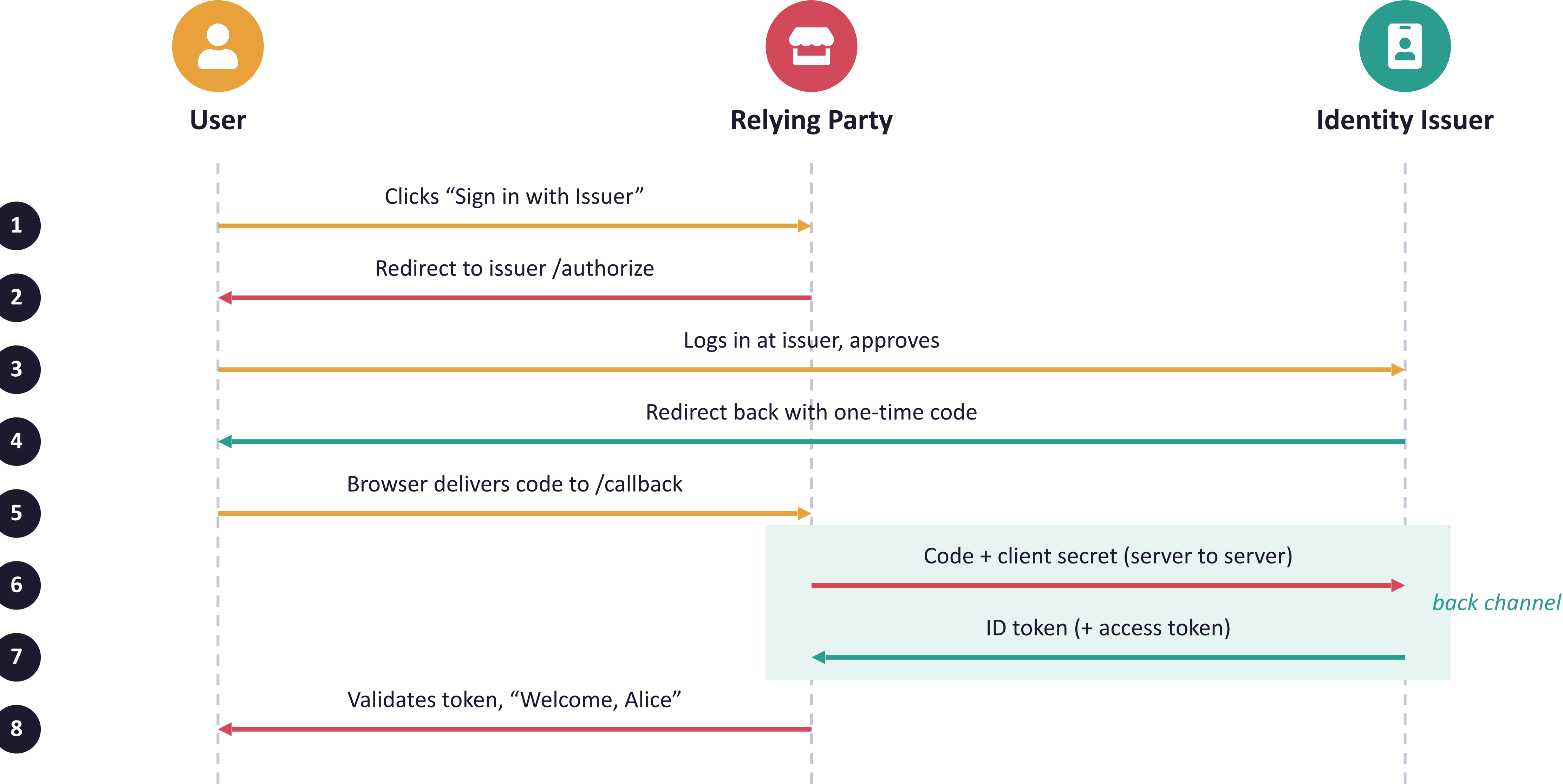


redirect_uri

The only address the issuer will send users back to.

Example: client_id = print-shop-123, redirect_uri = https://l4.neucrypt.org/callback

The whole flow at a glance





l4.neucrypt.org sends Alice to the issuer

Alice clicks “Sign in with Issuer”

l4.neucrypt.org doesn't show a password box. It responds with an HTTP redirect.

The redirect URL is the request

It names who's asking (client_id), where to return (redirect_uri), and what's wanted (scope).

state is a random value l4.neucrypt.org remembers

It will check for the same value on the way back, which blocks forged responses.

```
GET https://issuer.example/authorize
  ?response_type=code
  &client_id=l4.neucrypt.org-123
  &redirect_uri=
    https://l4.neucrypt.org/callback
  &scope=openid profile email
  &state=af0ifjsldkj
```

Everything here is visible in the browser, so nothing in this URL is secret.



Alice proves who she is, to the issuer only

issuer.example

Sign in

PrintShop would like to know your name and email.

Allow

The address bar says issuer.example

Alice types her password on the issuer's own page.
PrintShop can't see or intercept it.

The issuer may add extra checks

MFA, passkeys, device checks: all invisible to
l4.neucrypt.org, all handled here.

Consent

The issuer shows what l4.neucrypt.org asked for (the scope)
and lets Alice approve or refuse.



The issuer sends Alice back with a code

```
HTTP/1.1 302 Found
Location: https://l4.neucrypt.org
/callback
?code=Sp1x10BeZQQYbYS6WxSbIA
&state=af0ifjsldkj
```

Only to the registered `redirect_uri`

The issuer checks this against registration, so the code can't be sent to an attacker's site.

l4.neucrypt.org checks state first

Same value it stored in step 2? If not, reject.

One-time and short-lived

Typically valid for seconds to a minute, and useless without l4.neucrypt.org client secret.

The code is not a token. It's a claim ticket.



l4.neucrypt.org trades the code for tokens

Request (l4.neucrypt.org server → issuer)

```
POST /token
Host: issuer.example
Authorization: Basic
    base64(client_id:secret)

grant_type=authorization_code
&code=Sp1x10BeZQQYbYS6WxSbIA
&redirect_uri=
    https://l4.neucrypt.org/callback
```

Response (issuer → PrintShop server)

```
{
  "id_token": "eyJhbGciOiJSUzI1...\"",
  "access_token": "SlAV32hkKG",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

This call never touches the browser. The client secret proves it's really l4.neucrypt.org redeeming the code.



Read and verify the ID token

A signed JWT. Decoded payload:

```
{
  "iss": "https://issuer.example",
  "sub": "248289761001",
  "aud": "print-shop-123",
  "iat": 1790000000,
  "exp": 1790003600,
  "name": "Alice Liddell",
  "email": "alice@example.com"
}
```



Signature verifies with the issuer's published public key



iss is the issuer we sent Alice to



aud is our own client_id, not someone else's



exp hasn't passed

Then: look up the user by iss + sub (never by email alone), start a session, and show “Welcome, Alice.”

Every piece defends against something



Login at the issuer

l4.neucrypt.org never holds Alice's password



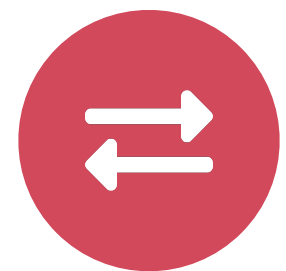
Registered redirect_uri

Codes can't be delivered to an attacker's site



state parameter

Forged or replayed callbacks are rejected



Code, then exchange

Tokens never pass through the browser



Client secret

A stolen code alone is worthless

Where to go next

PKCE

A per-login secret the RP invents on the spot. Lets mobile apps and SPAs, which can't keep a client secret, use this same flow safely. Now recommended for everyone.

nonce

A random value in the request that comes back inside the ID token, tying the token to this specific login.

Refresh tokens

Let the RP get new access tokens later without sending Alice back through the issuer.

Avoid the implicit flow

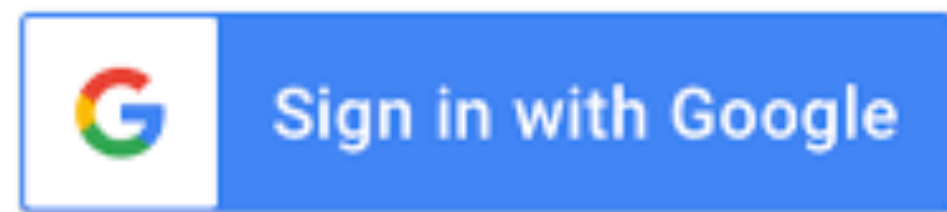
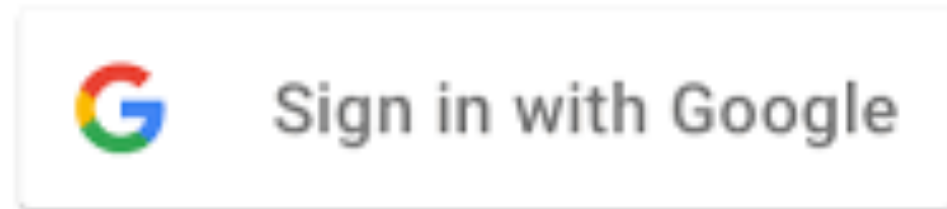
An older shortcut that returned tokens straight in the redirect. Deprecated for exactly the reasons on the last slide.

In one breath

The relying party sends the user to the issuer, who checks her identity and sends her back with a one-time code. The relying party trades that code, privately, for a signed ID token that says who she is.



Attacks against “Login with...” services



Use Sign in with Apple on your Apple device

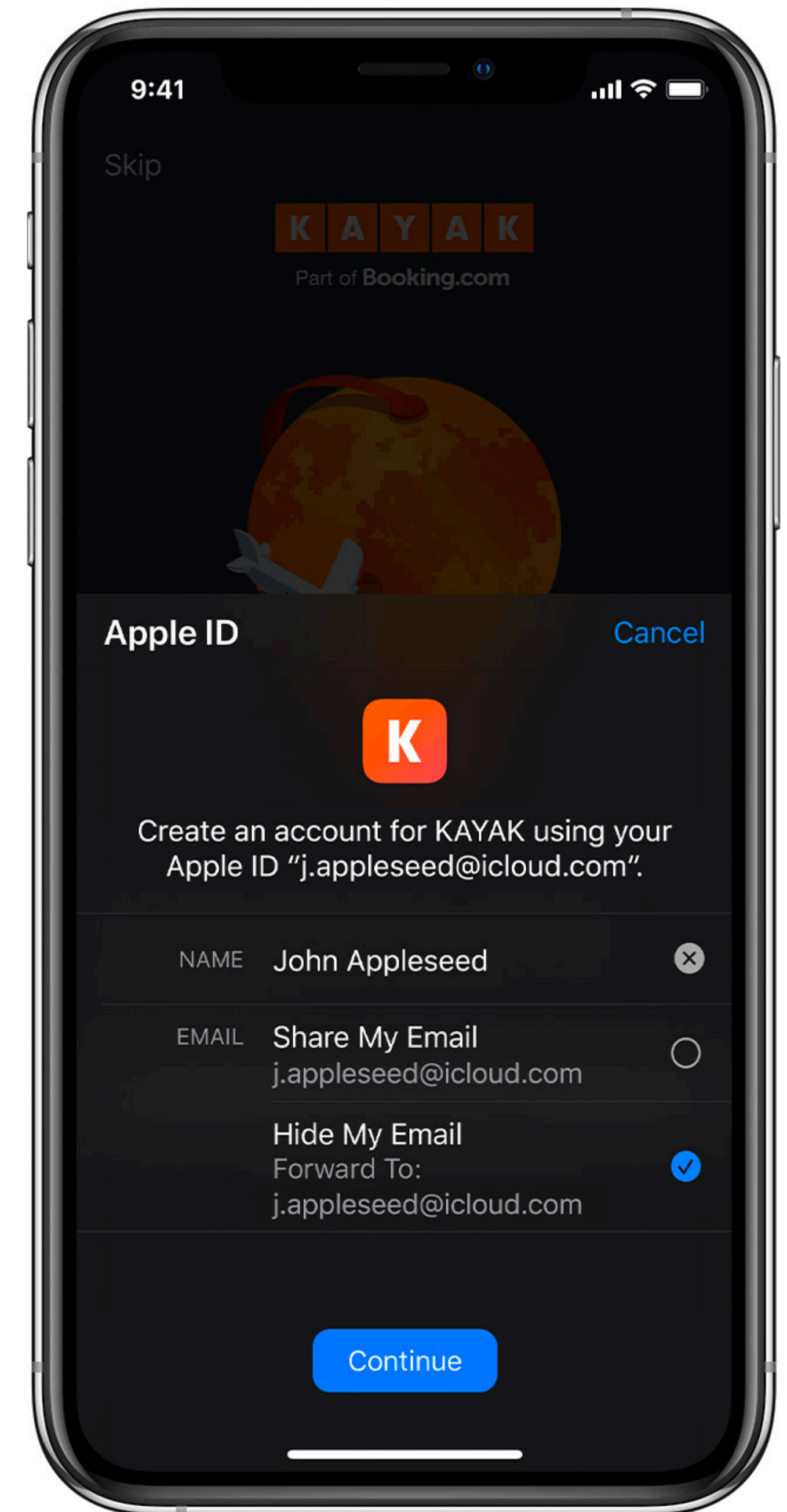
Using Sign in with Apple is quick and easy on any Apple device with the latest software. Make sure you're [signed in with your Apple ID](#) on your device.

1. Tap the Sign in with Apple button on the participating app or website.

If the app or site has not requested any information to set up your account, check that your Apple ID is correct and go to Step 4.

If you're asked to provide your name and email address, Sign in with Apple automatically fills in the information from your Apple ID. You can edit your name if you like and choose Share My Email or [Hide My Email](#).

Tap Continue and confirm with a quick Face ID, Touch ID, or device passcode to sign in. If you don't have Face ID, Touch ID, or a passcode set up, enter your Apple ID password.



Authentication:

Authorization

After Authenticating a subject, what next?

Access Control

- Policy specifying how entities can interact with resources
 - i.e., Who can access what?
 - Requires authentication and authorization
- Access control primitives

Principal User of a system

Subject Entity that acts on behalf of principals

Software program

Object Resource acted upon by subjects

Files

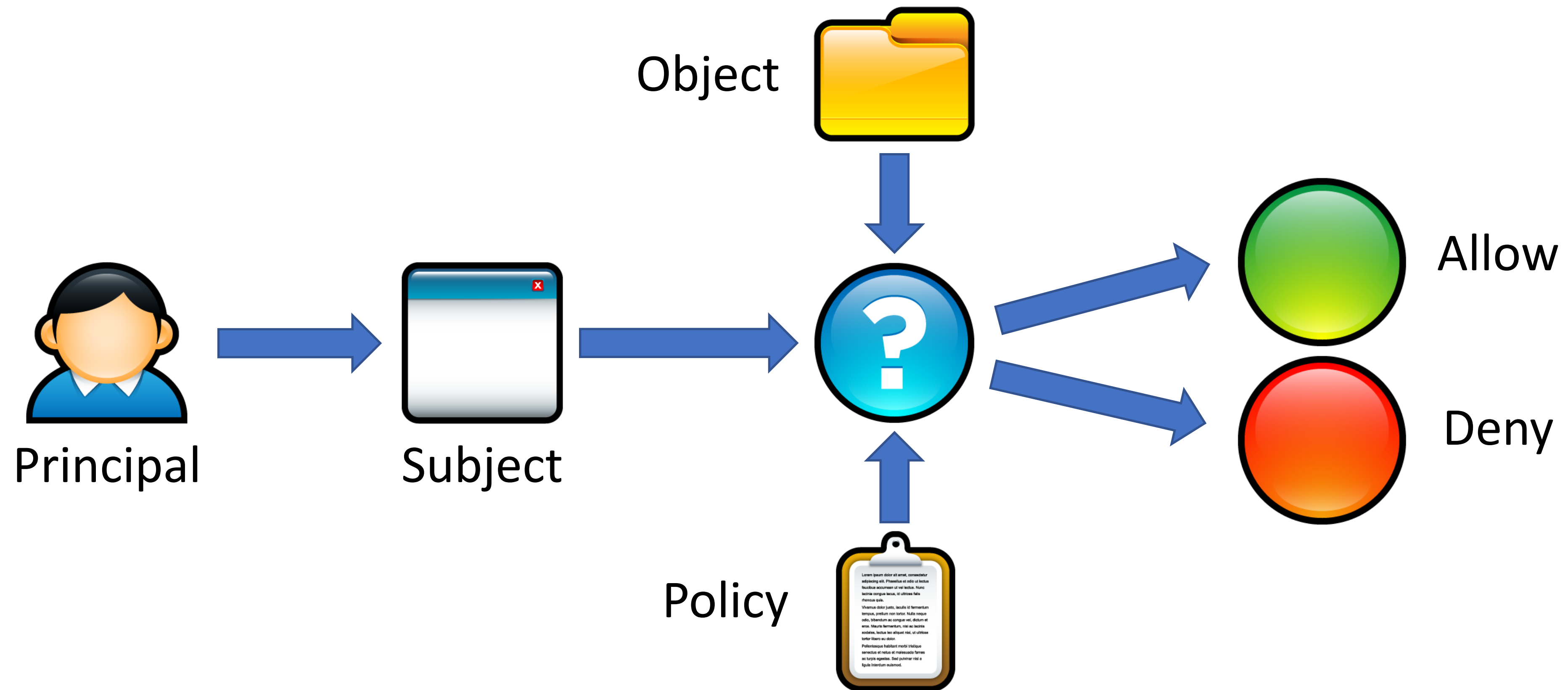
Sockets

Devices

OS APIs

Access Control Check

- Given an access request from a **subject**, on behalf of a **principal**, for an **object**, return an access control decision based on the **policy**



Access Control Models

- Discretionary Access Control (DAC)
 - The kind of access control you are familiar with
 - Access rights propagate and may be changed at subject's discretion

Access Control Models

- Discretionary Access Control (DAC)
 - The kind of access control you are familiar with
 - Access rights propagate and may be changed at subject's discretion
- Mandatory Access Control (MAC)
 - Access of subjects to objects is based on a system-wide policy
 - Denies users full control over resources they create

Sources

1. Many slides courtesy of Wil Robertson: <https://wkr.io>
2. Many slides courtesy of Ran Cohen