Authentication protocols

Segurança Informática em Redes e Sistemas
2022/23

Ricardo Chaves

Ack: Miguel Pardal, Miguel P. Correia, Carlos Ribeiro

Roadmap

- Authentication
- Authentication with passwords
- Biometric authentication
- **Authentication protocols**

Examples of authentication protocols

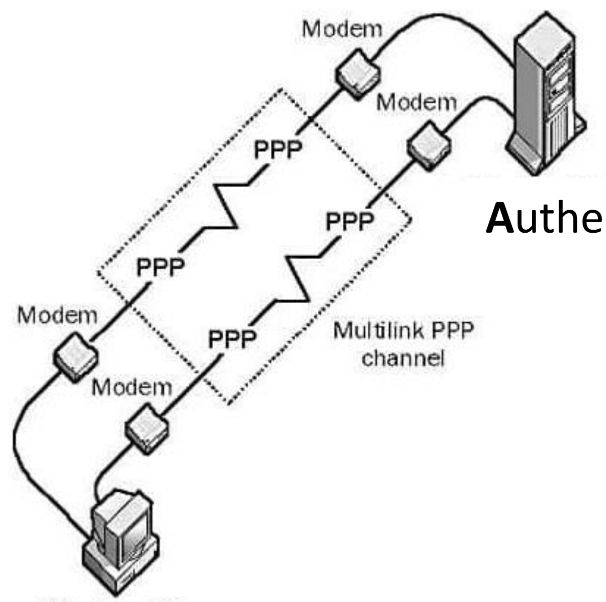
- With passwords
  - PAP, CHAP, MS-CHAP, Kerberos
- With one-time passwords
  - S/Key, SecurID
- With asymmetric keys
  - SSL, SSH, PGP (*later in the course*)

Roadmap

- Authentication
- Authentication with passwords
- Biometric authentication
- **Authentication protocols**
  - **With passwords**

PAP and CHAP

- Both protocols are used by PPP (Point-to-Point Protocol)
  - **Secret** shared between **authenticator (A)** and **user (U)** authenticated
  - Authentication is unidirectional (A authenticates U)



Authenticator

User

PAP

- PPP Authentication Protocol
  - Simple exchange of a pair UID/password

    U → A: username, password

    A →: OK/not OK
  - Insecure: the password is sent as is, in plaintext

CHAP

- Challenge Handshake Authentication Protocol

  U → A: username

  A → U: authID, challenge (authID: identifier for this attempt)

  U → A: Hash(authID, password, challenge)

  A →: OK/not OK

  – Authenticator may request authentication of the user at any moment

- *Problem with CHAP: **A** stores the passwords;*

  solution: next slide

MS-CHAP (Microsoft CHAP)

- **MS-CHAPv1** (RFC 2433)

  U → A: username

  A → U: authID, C

  U → A: **R**

  A → U: OK/not OK

  - C – challenge
  - **R** = $DES_{PH}(C)$ – cipher C with password hash PH
  - PH = NT-Hash or LM-Hash
    - NT-Hash = MD4(password)
    - LM-Hash based on DES
    - See "Windows Password Authentication" before

A stores only hash of the password

- **MS-CHAPv2** (RFC 2759)

  U → A: username

  A → U: authID, C_A

  U → A: C_U , **R1**

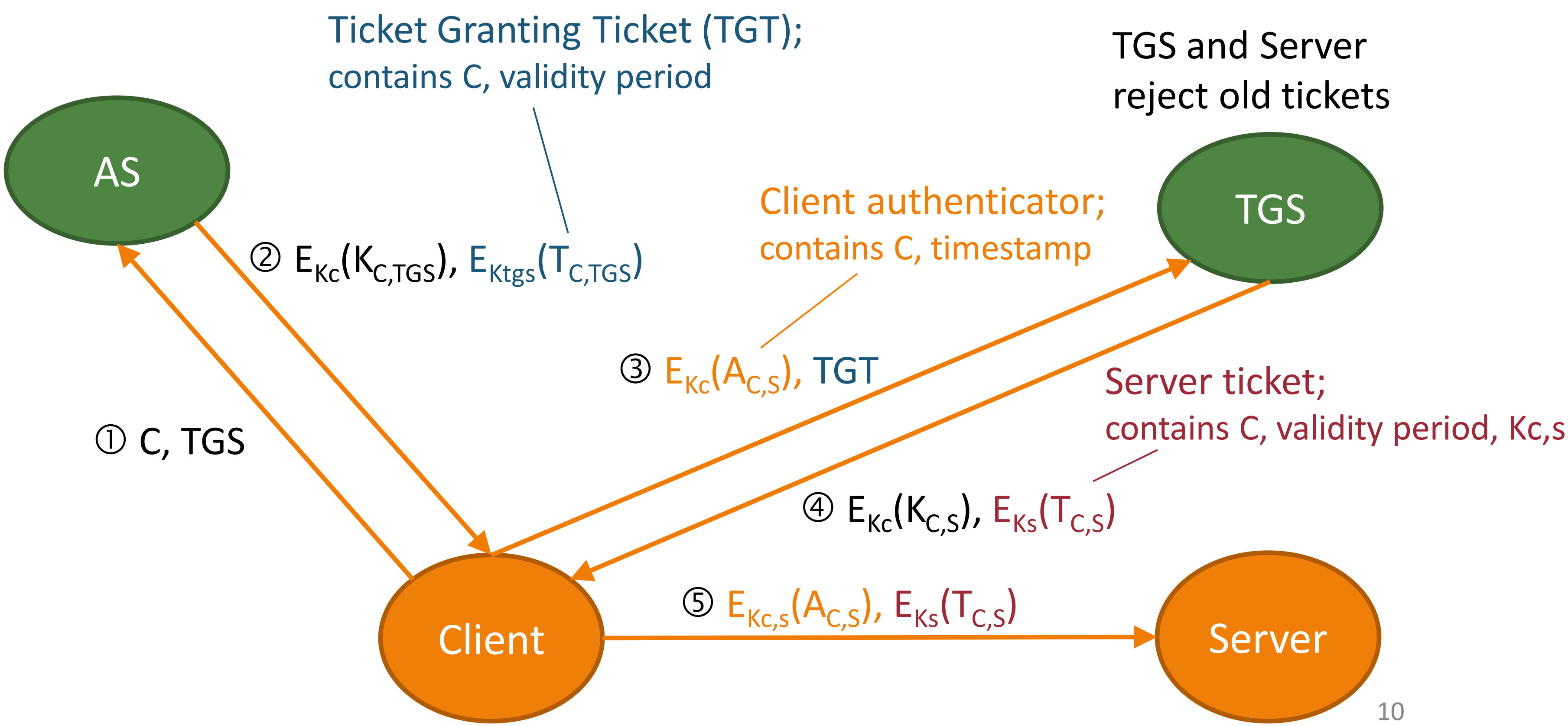
  A → U: OK/not OK, **R2**

  - C = SHA(C_U, C_A, username)
  - **R1** = $DES_{PH}(C)$
  - PH = MD4(password)
  - **R2** = SHA(SHA(MD4(PH), m1), C, m2)

- **Mutual authentication**
  - Authenticator shows knowledge of PH (in R2)
- Insecure LM-Hash is no longer allowed

Kerberos 5

- C, S, TGS – client, server, TGS identifiers
- Ktgs – TGS secret key, shared with AS
- **Kc – client secret key, shared with Kerberos**
- **Ks – server secret key, shared with Kerberos**
- Kc,s – session key, distributed to client and server by the protocol



Roadmap

- Authentication
- Authentication with passwords
- Biometric authentication
- **Authentication protocols**
  - **With one-time passwords**

S/Key

- Authentication protocol with **one-time passwords**
  - Password used only once, so sent in plaintext
- **Authenticator** gives **user** a sequence of one-time passwords

  OTP_1 = Hash(seed, password) ... OTP_n = Hash(OTP_{n-1})

  - For each user, the authenticator **stores** only: seed of the one-time password sequence; current index; OPT_{index} value
- **Authentication process:**
  - Authenticator sends index to the user
  - User sends P = $OPT_{index-1}$ to the authenticator (the one-time password)
  - Authenticator computes Hash(P) and compares with stored OPT_{index}
  - If values are equal, then success, server stores index-1 and $OPT_{index-1}$
  - The 1st time the authenticator sends index = n, next index = n-1, etc.

S/Key interaction

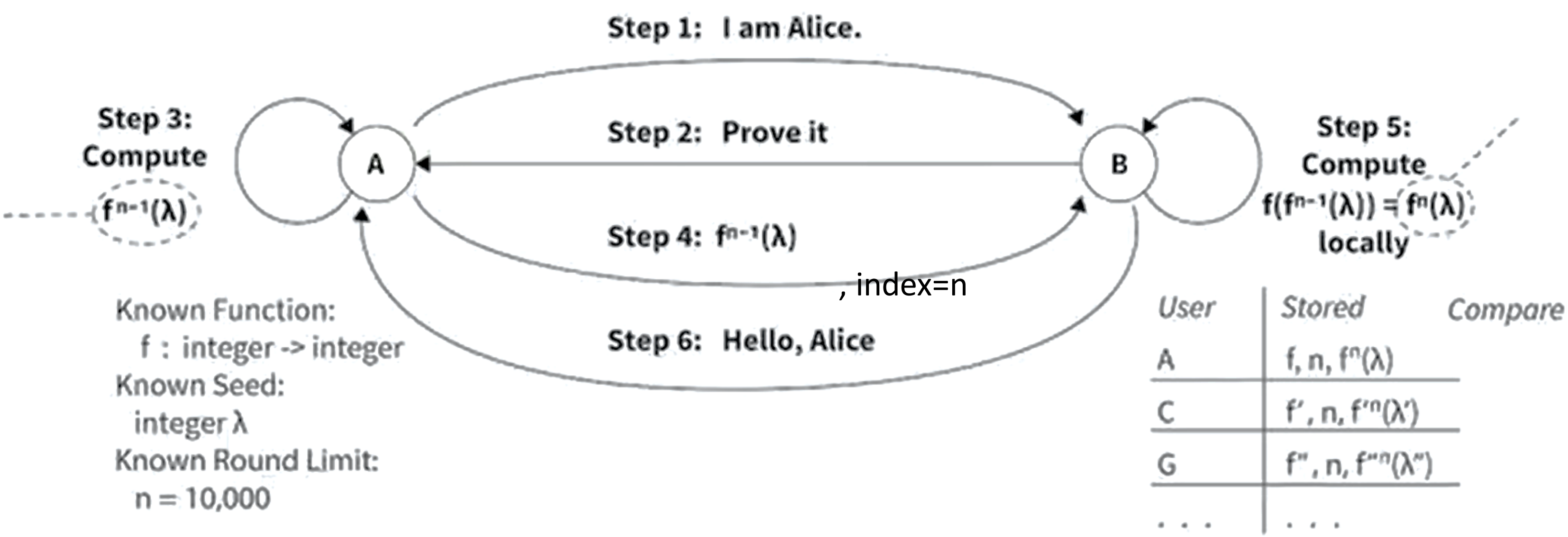


Image credits: Ed Amoroso, Tag Cyber

- Security:
  - Alice is the User, Bob is the Authenticator
  - All communication is in plaintext
  - Given $\text{OPT}_{\text{index-1}}$ it is not possible to get $\text{OPT}_{\text{index-2}}$ due to hash function
    - Easy to go in one direction but impossible in the opposite

RSA SecurID

- Personal authentication device from RSA Security (now Dell)
  - Also, in software for handheld devices and smartphones
- Generates a **unique number every minute**
  - They are essentially one-time passwords
  - They are computed using:
    - A **64-bit key** stored inside the card (shared with the RSA server)
    - The current date
    - A hash algorithm - **SecurID Hash**
    - Some versions allow inserting a PIN
- Authentication with a one-time password
  - The user generates a one-time password, combining a PIN with the card number
  - The RSA server performs the same operation an verifies if the values are equal
  - Server is synchronized with real time (RSA Security Time Synchronization)

Roadmap

- Authentication
- Authentication with passwords
- Biometric authentication
- **Authentication protocols**
  - **With asymmetric keys**

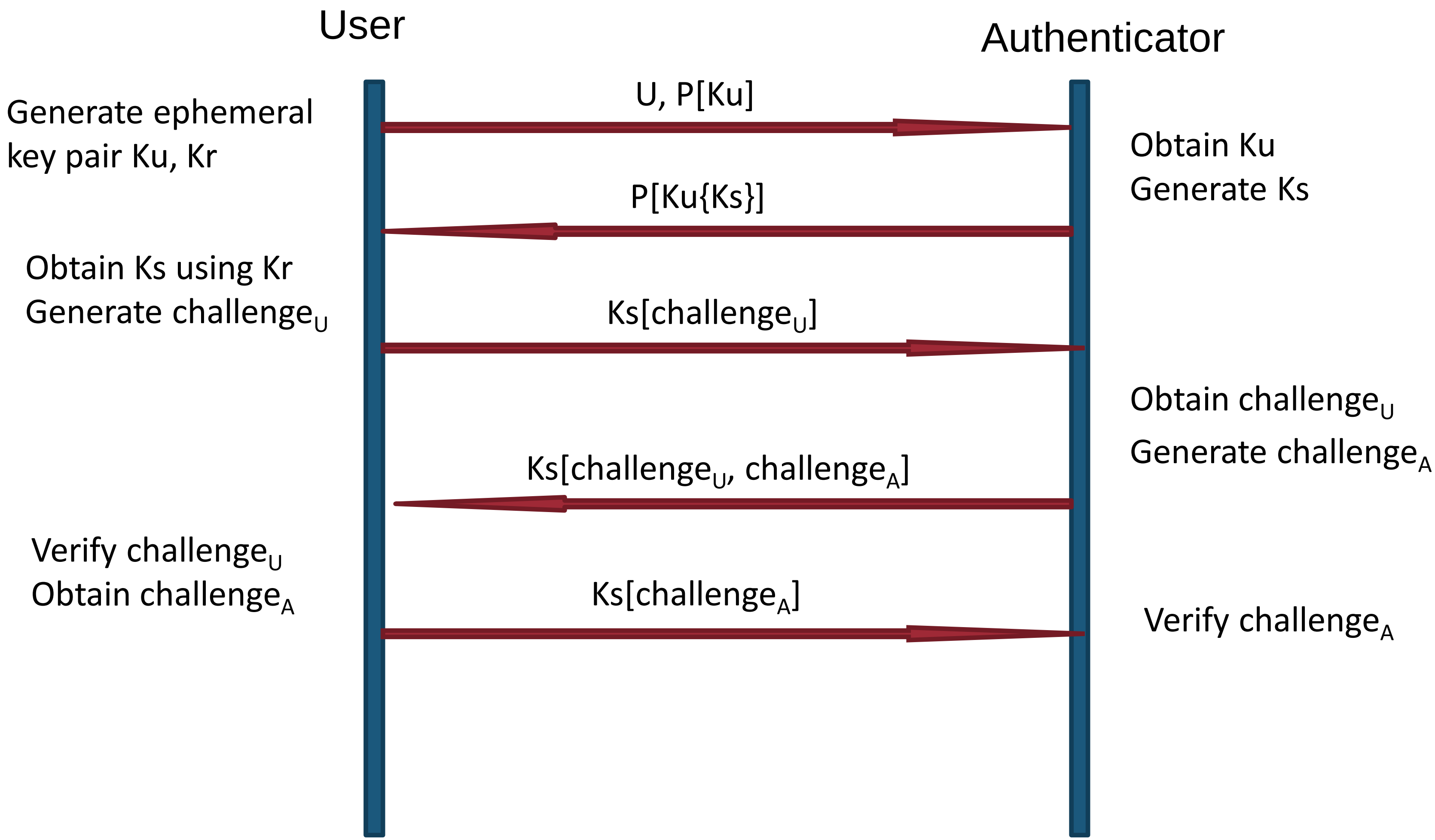
EKE (Encrypted Key Exchange)

- **Key agreement** –establishing a **session key (Ks)**
- **EKE** is a protocol that does **password-authenticated key agreement**
  - Uses combination of asymmetric and symmetric ciphers
  - It is one of the few key password-authenticated agreement protocols that is **resistant to dictionary attacks**
- Can be used with several asymmetric techniques:
  - RSA, ECC, Diffie-Hellman
- Uses **ephemeral keys** – keys used only once

EKE notation

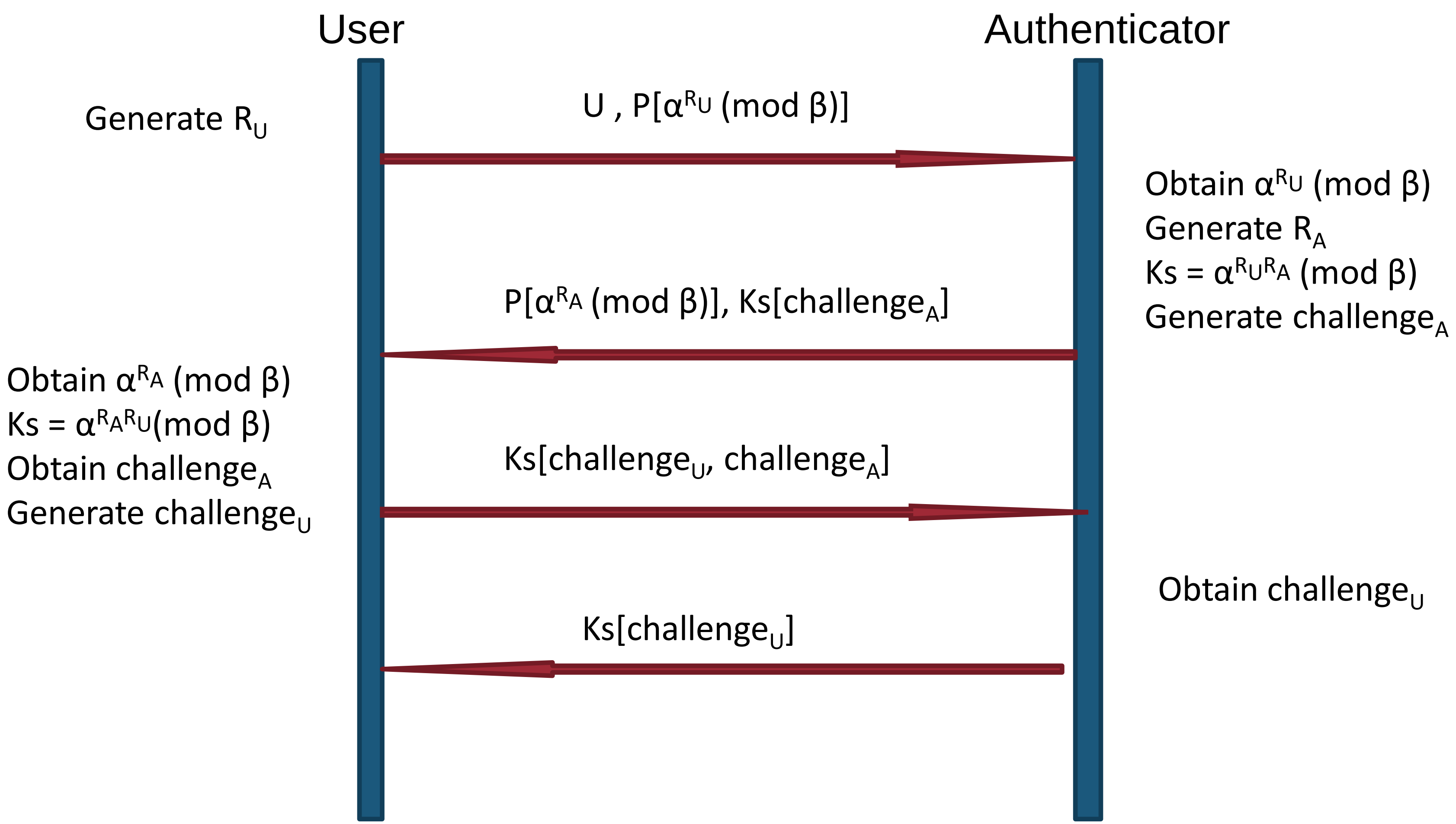
- General
  - U, A – User and Authenticator identifiers
  - P – Password, known by U and by A (!); used as a secret key
  - Ks – Session key established by the protocol; the result of the protocol
  - RU, RA, challengeU, challengeA – values generated by the protocol
- Symmetric cipher
  - K – some secret key
  - K[m] – Cipher m with the secret key K
  - K^{-1}[m] – Decipher m with the secret key K
- Asymmetric cipher
  - Ku, Kr – public and private keys, respectively
  - Ku{m} – Cipher m with the public key Ku
  - Kr{m} – Decipher m with the private key Kr

EKE with RSA


User
Authenticator
Generate ephemeral key pair Ku, Kr
U, P[Ku]
Obtain Ku
Generate Ks
P[Ku{Ks}]
Obtain Ks using Kr
Generate challenge$_U$
Ks[challenge$_U$]
Obtain challenge$_U$
Generate challenge$_A$
Ks[challenge$_U$, challenge$_A$]
Verify challenge$_U$
Obtain challenge$_A$
Ks[challenge$_A$]
Verify challenge$_A$

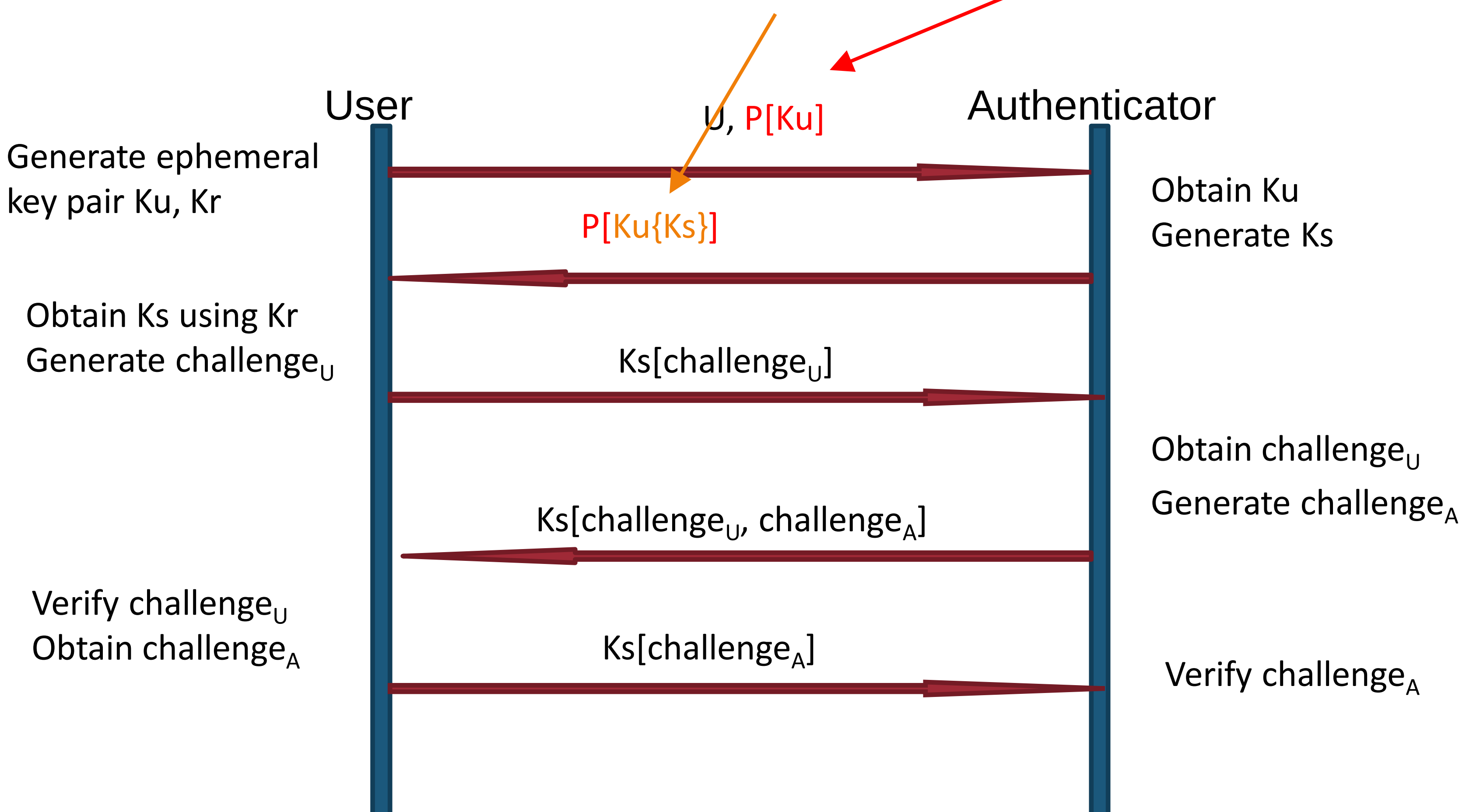
EKE with Diffie Hellman

DH-EKE is similar to EKE but with DH



EKE protection against dictionary attacks

- Even if successful decrypting the public key: $P'^{-1}[P[Ku]] = Ku'$
  - The attacker would also have to compute the private key Kr to obtain the session key, i.e., to decrypt Ku{Ks}

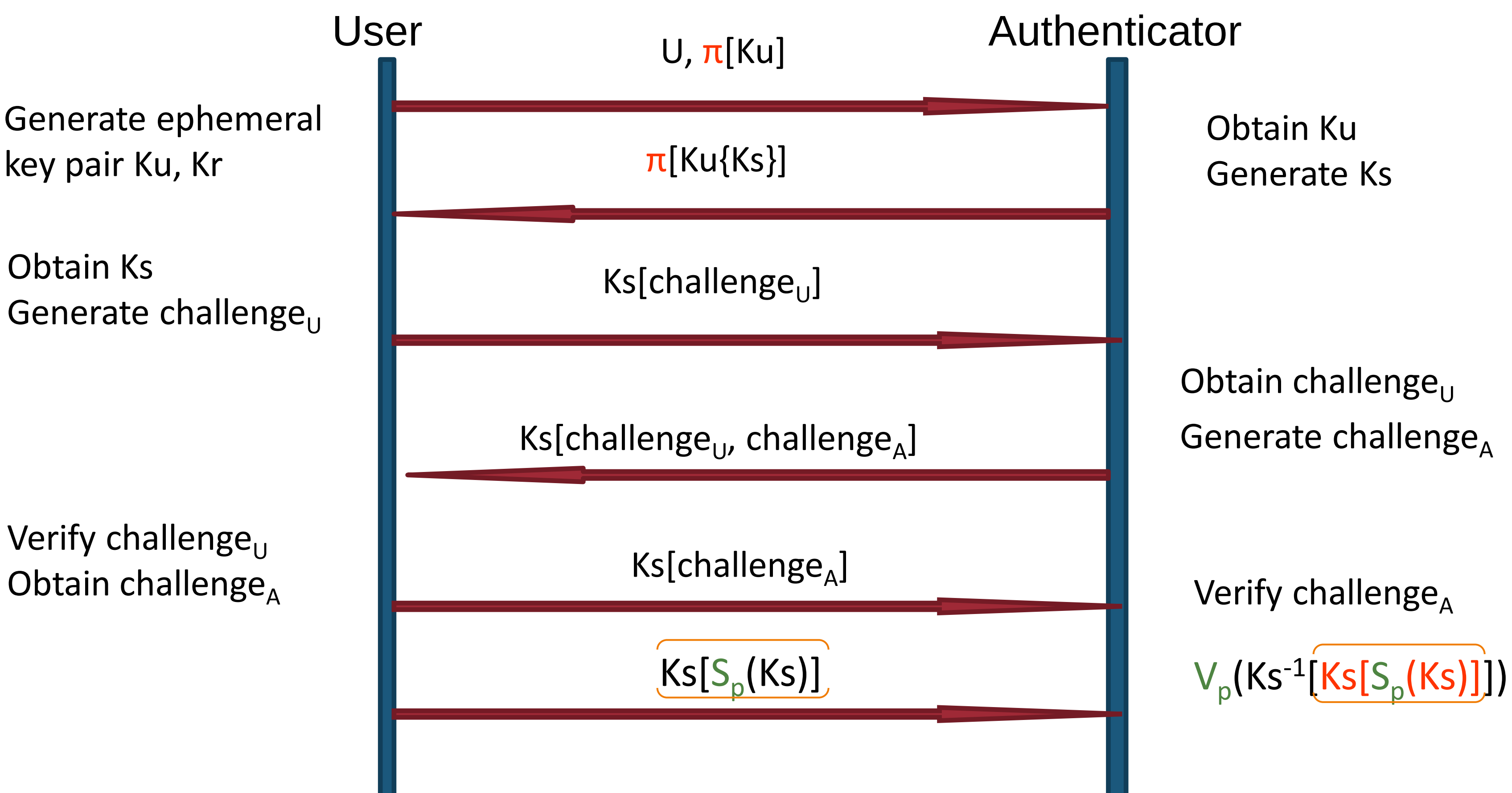


EKE security

- Man-in-the-middle protection
  - The session key Ks is only known by who generated it (Authenticator) and by who has the private key Kr (User)
- Replay protection
  - The last three messages validate the freshness of the key Ks
- Attack to the password database in the Authenticator would give access to P
  - Solution: **Augmented EKE** that only stores Hash(P) = π

Augmented EKE (A-EKE)

- Prevents impersonation of U with password P stolen from A
  - π = hash(P) – user knows P; authenticator has only this hash
  - $S_p()$: one-way function configured with P (e.g., MAC); verified with V_p



Roadmap

- Authentication
- Authentication with passwords
- Biometric authentication
- **Authentication protocols**
  - **Web authentication**

HTTP Basic/Digest Authentication

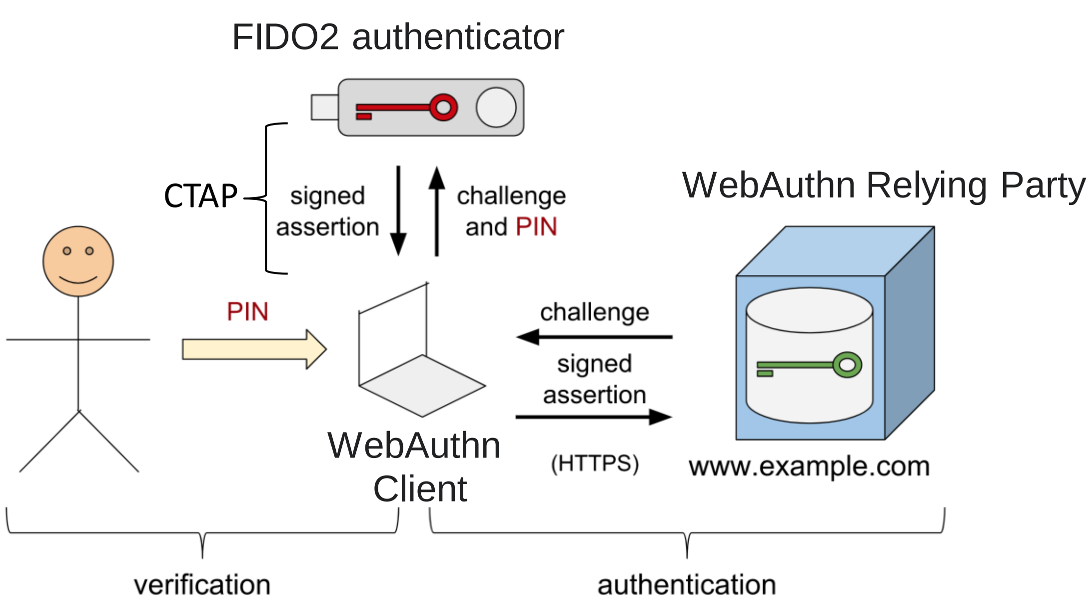
- note: never use on insecure connections, only over HTTPS
- Basic Authentication: when user asks for protected page:
  - server sends reply with status code 401 (unauthorized) and WWW-Authenticate header, e.g.:
  - *`WWW-Authenticate: Basic realm="Authorized Personnel Only"`*
  - browser asks user for username and password for that page and realm
  - browser sends username and password (**bad**) to the server encoded in base 64 (trivial to decode), e.g.:
  - *`Authorization: Basic YWJlbGFyZG86YmFzaWM=`*
- Digest Authentication:
  - server sends something similar but: *`Digest`*, a nonce, a hash algorithm
  - browser sends **hash** instead of credentials

Authentication in HTTPS

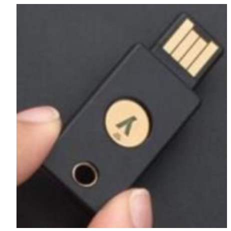
- **Server authentication**
  - Authentication provided by SSL/TLS (later)
  - Browser authenticates the server using the server's public key stored in a certificate
  - Browser has certificates with public keys of CAs
- **User/client authentication**
  - Typically, server does not authenticate the browser (**client**)
  - Instead, the **user** authenticates himself using, e.g., username/password
  - Ok if HTTPS connection already established (credentials are sent encrypted over the network)

FIDO2 user authentication

- **Strong user authentication for the web**
  - Based on user-controlled cryptographic authenticators: smartphone, hardware security key
- Two components:
  - W3C Web Authentication (WebAuthn) standard
  - FIDO Client to Authenticator Protocol (CTAP)



Yubico YubiKey 5 NFC



Yubico Yubikey 5C

CryptoTrust OnlyKey

Session-based authentication

- Server sends **cookie** with session identified to the client
  - `Set-cookie: string` in the header
- Whenever browser sends request to the server, cookie is inserted automatically
  - `cookie: string` in the header

- With HTTPS: cookies ciphered; OK
- Without HTTPS: cookies sent in plain text;
can be stolen and used by attacker

Roadmap

- Authentication
- Authentication with passwords
- Biometric authentication
- **Authentication protocols**
  - **Web single-sign on**

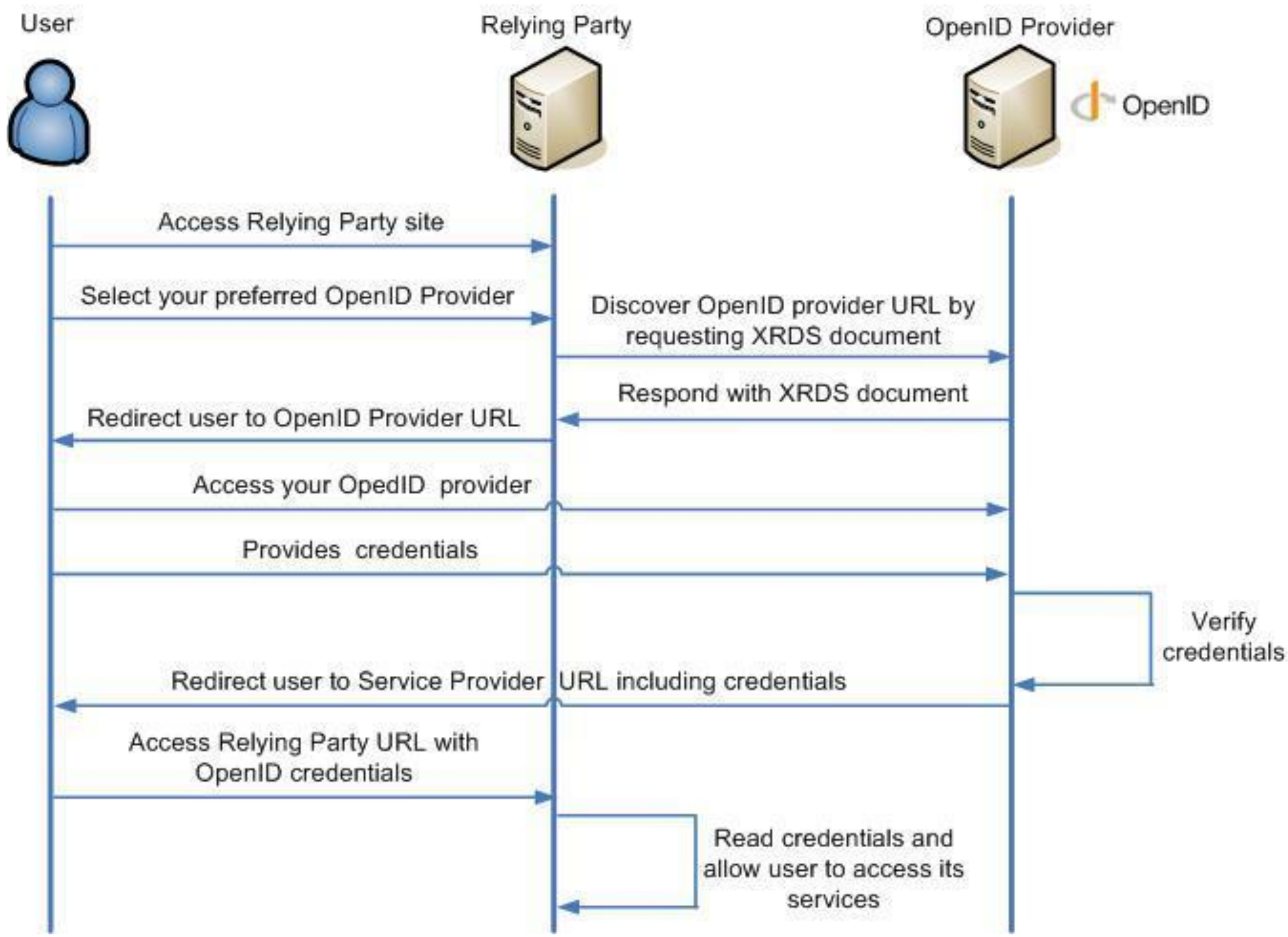
Single-Sign On

- Problem: users need to authenticate in many services, keep many passwords
  - Reusing passwords is bad, if one is stolen…
- Solution: **Single-Sign On**
  - Authenticate once, use several times, or
  - Have a single means of authentication, use several times
- For the web there are few SSO solutions:
  - **OpenID 2.0**
  - **OpenID Connect / OAuth**
  - Have a single means of authentication, use several time

OpenID 2.0

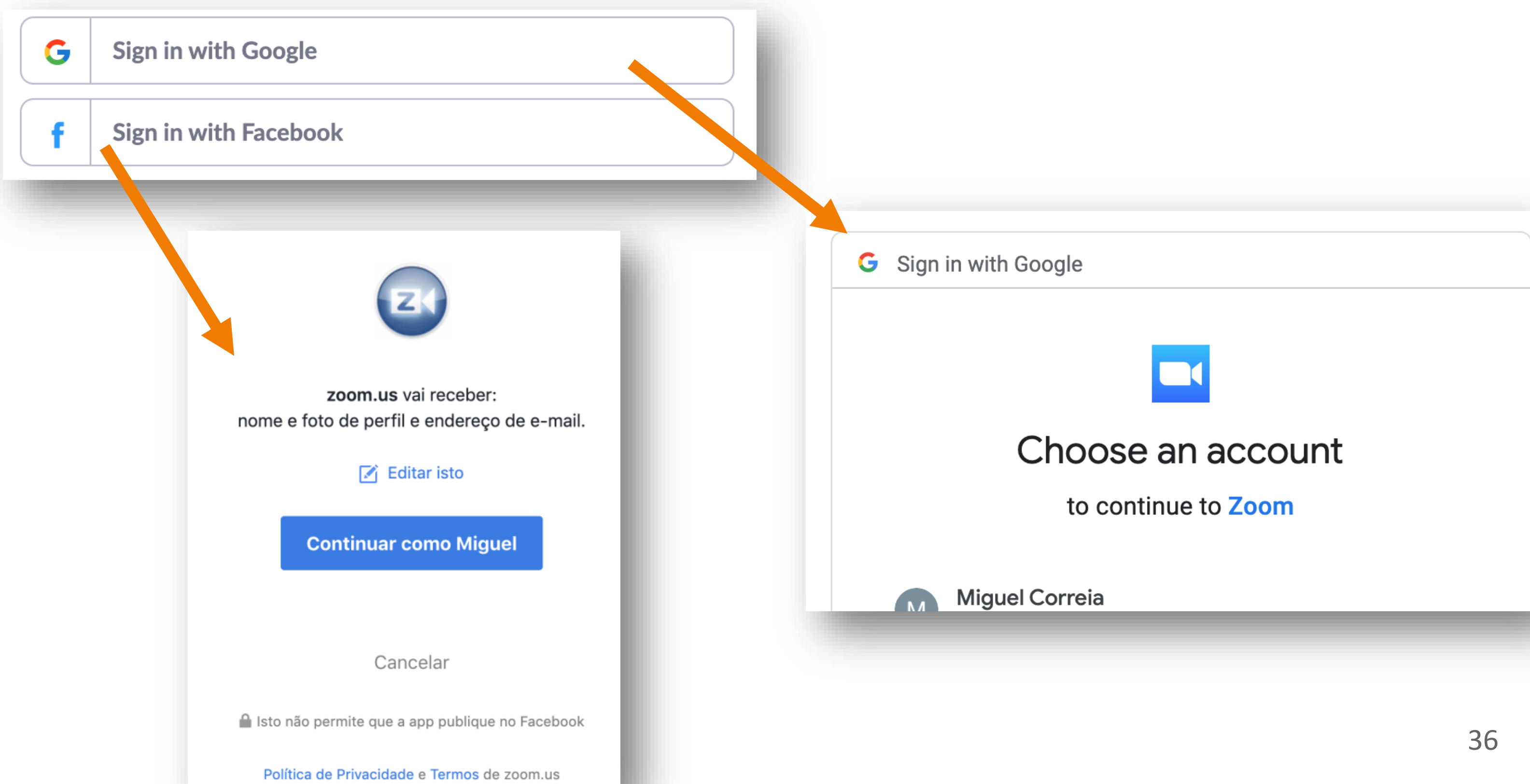
- Service provided by an **OpenID provider**
  - Google, Yahoo!, Wordpress...
- **User** wants to login in a **relying party**, i.e.,
  a web site that supports OpenID (displays the logo)
  - User provides his **OpenID**
  - Relying party transforms the OpenID in a URL and redirects the user's browser to the OpenID provider
  - OpenID provider typically asks for password and the user to allow the relying party to use the account
  - User is redirected to the relying party website; OpenID provider sends authentication data to the relying party
- **Important**: all the communication links use HTTPS
  - Secure channels with authenticated servers

OpenID 2.0

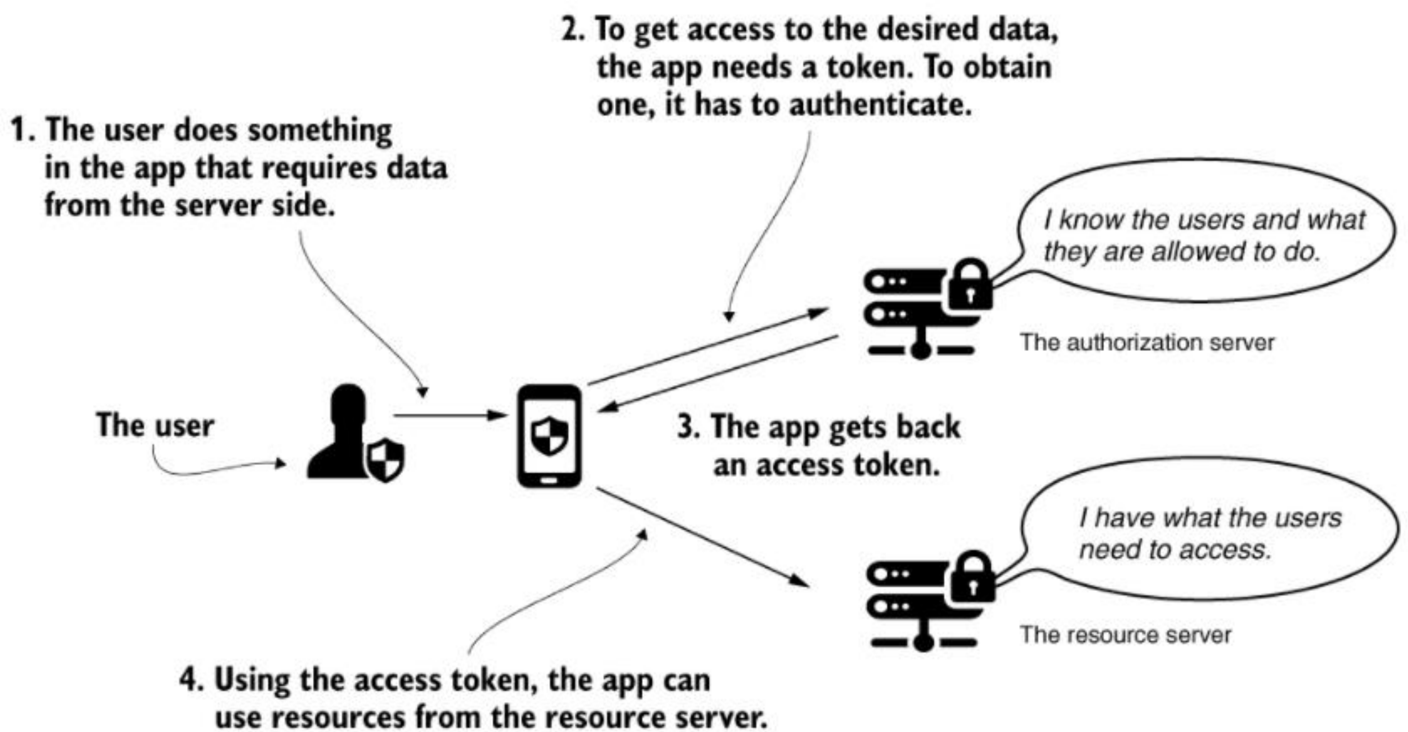

User
Relying Party
OpenID Provider
OpenID
Access Relying Party site
Select your preferred OpenID Provider
Discover OpenID provider URL by requesting XRDS document
Respond with XRDS document
Redirect user to OpenID Provider URL
Access your OpedID provider
Provides credentials
Verify credentials
Redirect user to Service Provider URL including credentials
Access Relying Party URL with OpenID credentials
Read credentials and allow user to access its services

OpenID Connect / OAuth

- **OpenID Connect** (OIDC) works similarly to OpenID
  - Works on top of **OAuth 2.0**, a distributed authorization framework
  - Example from Zoom's Help Center:



OAuth2 authorization flow



Summary

- Authentication
- Authentication with passwords
- Biometric authentication
- Authentication protocols