

# Authentication

---



# Outline

1. Password-based authentication
2. 2-Factor Authentication (2FA)
3. 2FA in Practice
4. Single Sign-On (SSO) using Token-based Security
5. Delegated Authentication

# Application Security Aspects

- **Authentication (Identity verification):**
  - Verify the identity of the user given the credentials received
  - Making sure the user is who he claims to be
- **Authorization:**
  - Determine if the user should be granted access to a particular resource.
- **Confidentiality:**
  - Encrypt sensitive data to prevent unauthorized access in transit or in storage
- **Data Integrity:**
  - Sign sensitive data to prevent the content from being tampered (e.g., changed in transit)

# Password-based Authentication



# Passwords

- Passwords are the most commonly used authenticator in computer systems
  - A combination of letters, numbers, and special characters that a user supplies in order to prove their identity
  - Frequently passwords are chosen by the user
- But security problems associated with passwords:

81%

Data breaches in 2016  
that involved weak,  
**default, or stolen**  
passwords (VDBR)

1 IN 14

Phishing attacks were  
successful in 2016  
(VDBR)

1,579

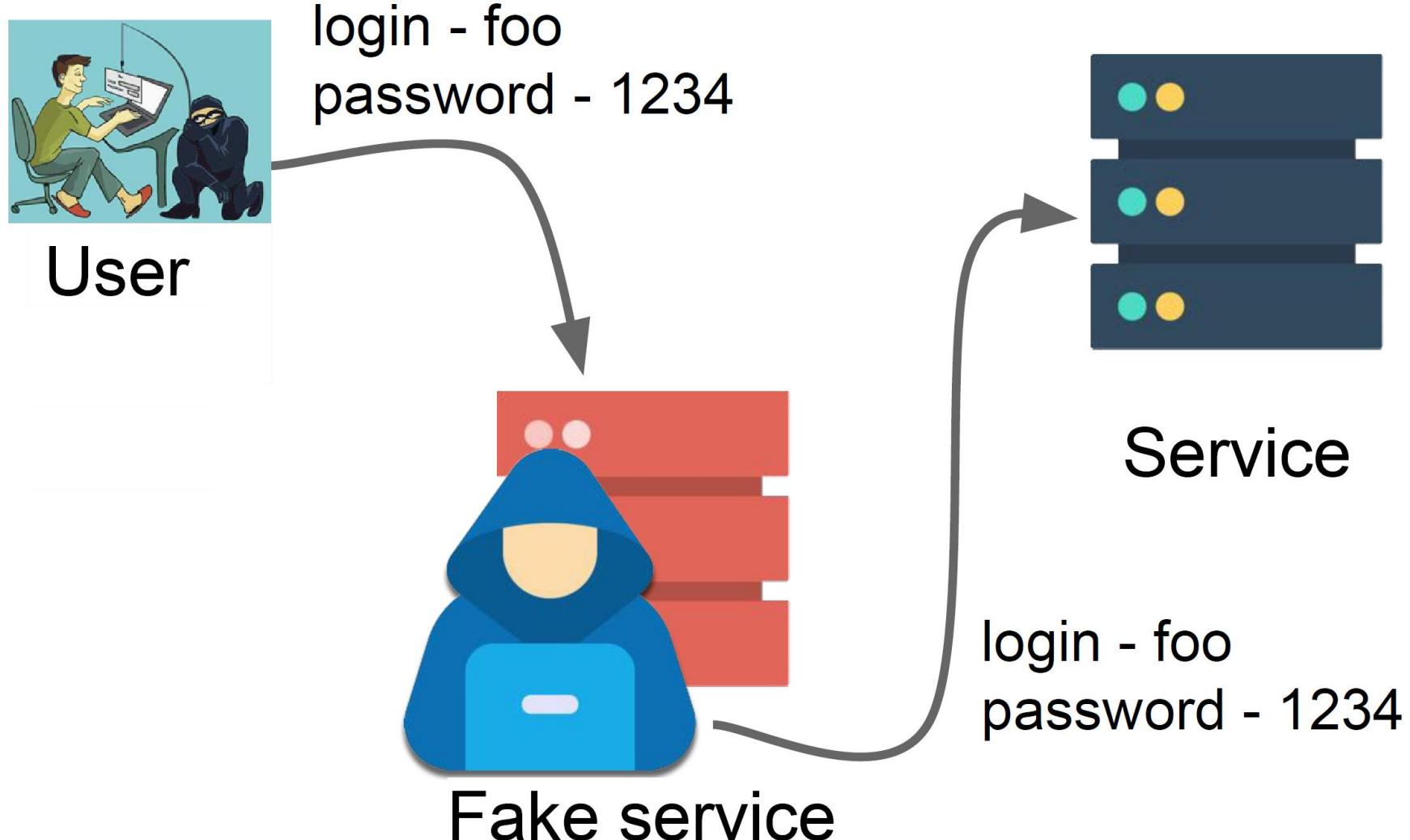
Breaches in 2017, a 45%  
**increase over 2016**  
(ITRC)

# Password Vulnerabilities

- Actually not very secure
  - Weak password: Users tend to pick simple passwords and easy to guess passwords
    - 123456 and *password* are the most common passwords!!!
    - Good (random) passwords are almost impossible to remember
  - Password Reuse: re-use passwords between services
    - If one service is hacked, then someone knows your password for all other services
  - Vulnerable to key loggers
  - Vulnerable to phishing attacks



# Phishing



# How Much Password Complexity is Needed?

- In general, a password can be thought of as good if brute-forcing it would require the same amount of work as a 64-bit encryption key
- How do we measure brute-force complexity of a password?

# Brute force Complexity for Passwords

- How many different 4 character, all lowercase passwords are there?

$$\begin{matrix} ? & ? & ? & ? \\ 26 * 26 * 26 * 26 \\ = 26^4 \end{matrix}$$

## Translating into Key Complexity

$$\begin{aligned} 26^4 &= 2^x \\ x &= \log_2(26^4) \\ x &= 18.8 \text{ bits} \\ 26^4 &\approx 2^{19} \end{aligned}$$

So, a 4 character password of all lowercase letters has the same brute force complexity as a ~19 bit encryption key

# Let's make it better!

- 8 characters, all lowercase
  - $26^8 \rightarrow 37.6$  bits
- 8 characters, lower and uppercase
  - $52^8 \rightarrow 45.6$  bits
- 8 characters, lower, upper and numbers
  - $62^8 \rightarrow 47.6$  bits
- 8 character, lower, upper, numbers, special
  - $94^8 \rightarrow 52.4$  bits

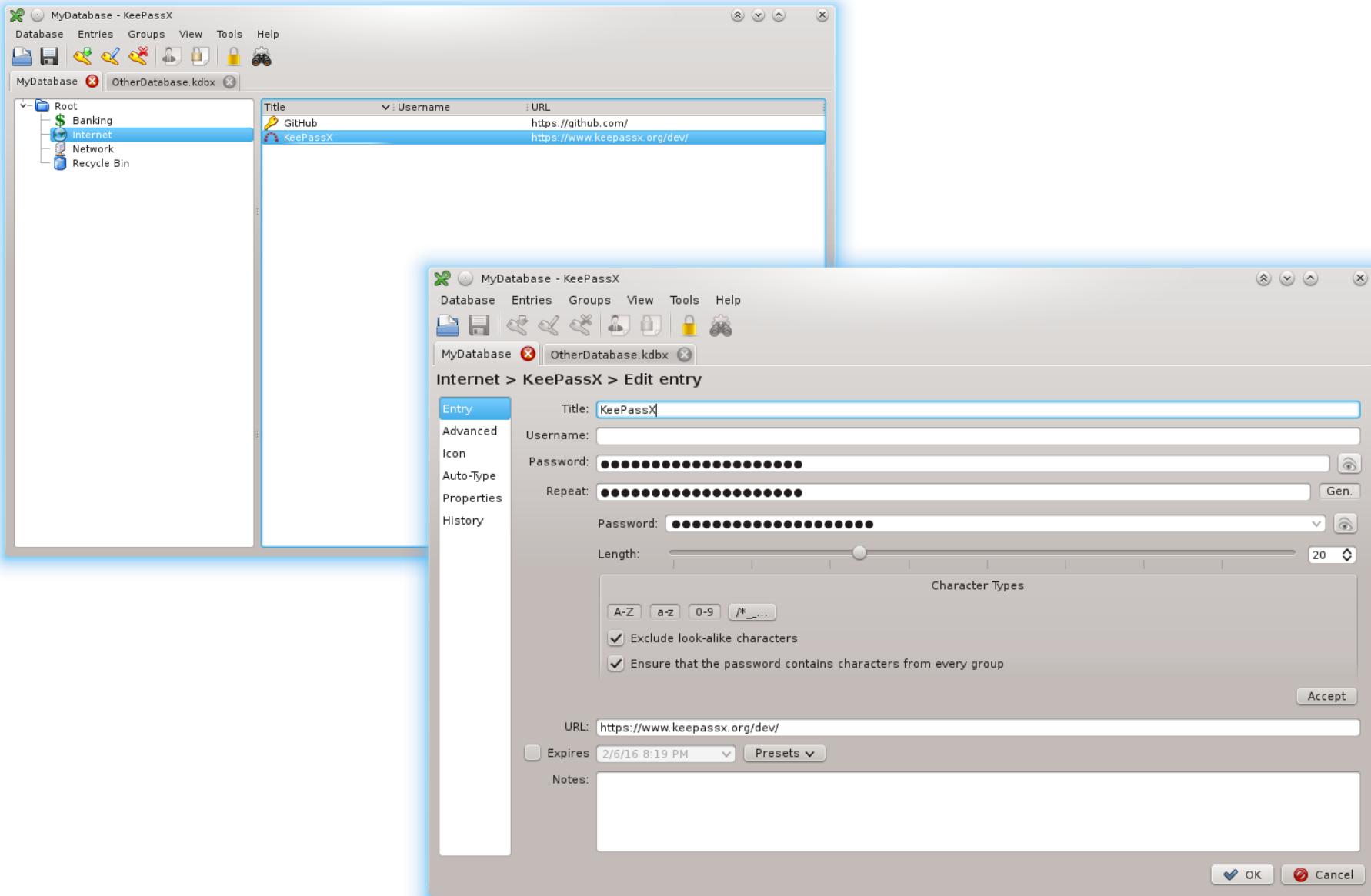
=> A randomly generated 8 character password with lowercase, uppercase, numbers, and special characters *is still not secure enough*

- How long would the password need to be for 64-bits of security?
  - $94^{10} \rightarrow 65.54$  bits

# Improvements - End-User Password Hygiene

- Use a different password for every account
- Make your passwords complex or randomly generated
- Use a password safe to generate and store passwords so that you don't need to remember them
  - All your passwords can be truly random!
- Open source password manager
  - <http://www.keepassx.org/>
- Commercial password managers
  - <https://www.pcmag.com/article2/0,2817,2407168,00.asp>

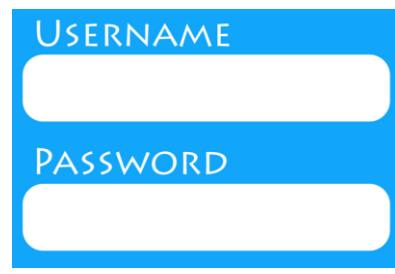
# Password Safe



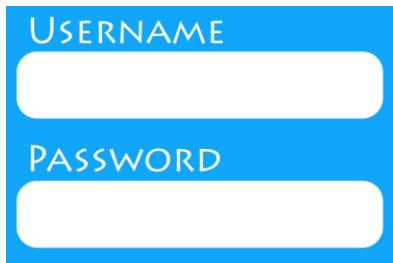
# Improvements - Service Provider

- Lock the account after 3 unsuccessful attempts
- Use 2-Factor Authentication (2FA) as Single Factor Authentication is insufficiency

# 2-Factor Authentication (2FA)



# Authentication Factors



**Something the user knows  
(PIN, password)**



**Something the user has  
(mobile phone, device)**



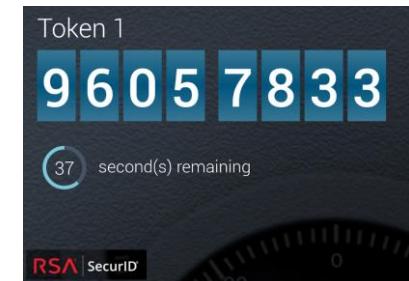
**Something the user is  
(biometric, retina, fingerprint)**

# 1. Something the user knows

- Information that would only be known by the user
- Examples:
  - Password (most common)
  - PIN number
  - Security questions (usually biographical information)

## 2. Something the user has

- A physical item the user has
- Examples
  - Smart phone
  - Smart card
  - Security token
- Security Token is Device or App that displays a number that changes ~every 30 seconds
  - Commonly used for corporate VPNs, generating one-time passwords (OTPs), etc.
- USB security key



### 3. Something the user is

- *Biometrics*
  - Based on physical characteristics of the user
- Examples
  - Fingerprints and Retina/Iris scan used in Qatar for e-government and e-gate systems
- Not revokable or transferable...
- But the use of biometric information is less common since fingerprint or retina recognition software is expensive and difficult to implement.



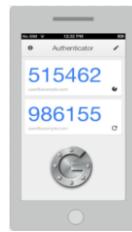
# Multi-Factor Authentication

- Using multiple factors of authentication increases security
  - Must include *unique* factors
  - In practice 2-Factor Authentication (2FA) is used
- Example:
  - Requiring a password and a security question is still only one-factor (what the user knows)
  - Requiring a password and the code from an SMS sent your phone is two-factor (what the user knows and what the user has)
  - Requiring a password and fingerprint: *two-factor* authentication (what the user knows and what the user is)

# 2FA in Practice



SMS



Time-based One-time  
Passwords (TOTP)  
e.g. Google Authenticator



Approve

Push Notifications  
e.g. Google Prompt

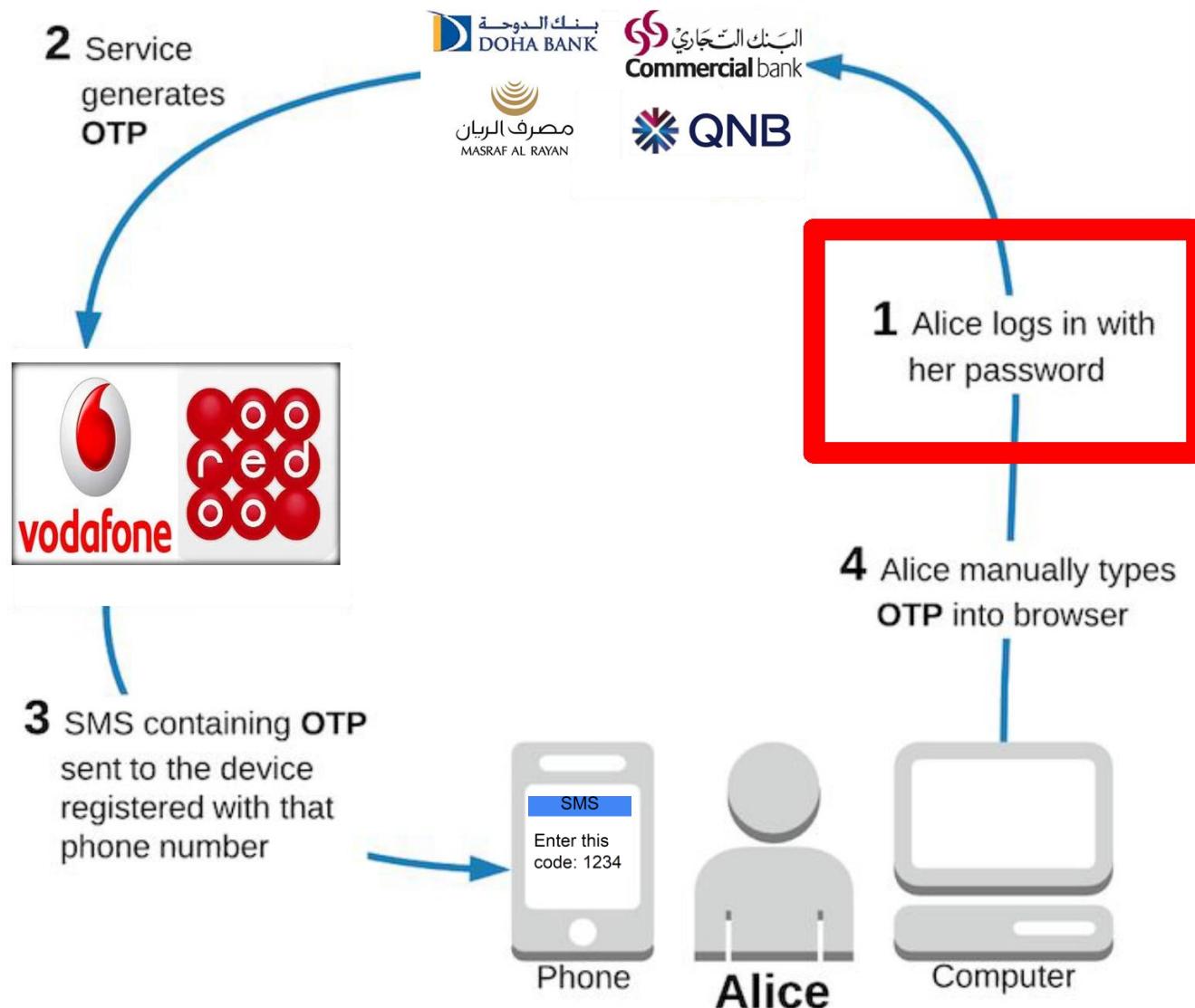


Universal 2<sup>nd</sup> Factor (U2F)  
e.g. Yubico Security Key  
(Public-Key Cryptography Device)



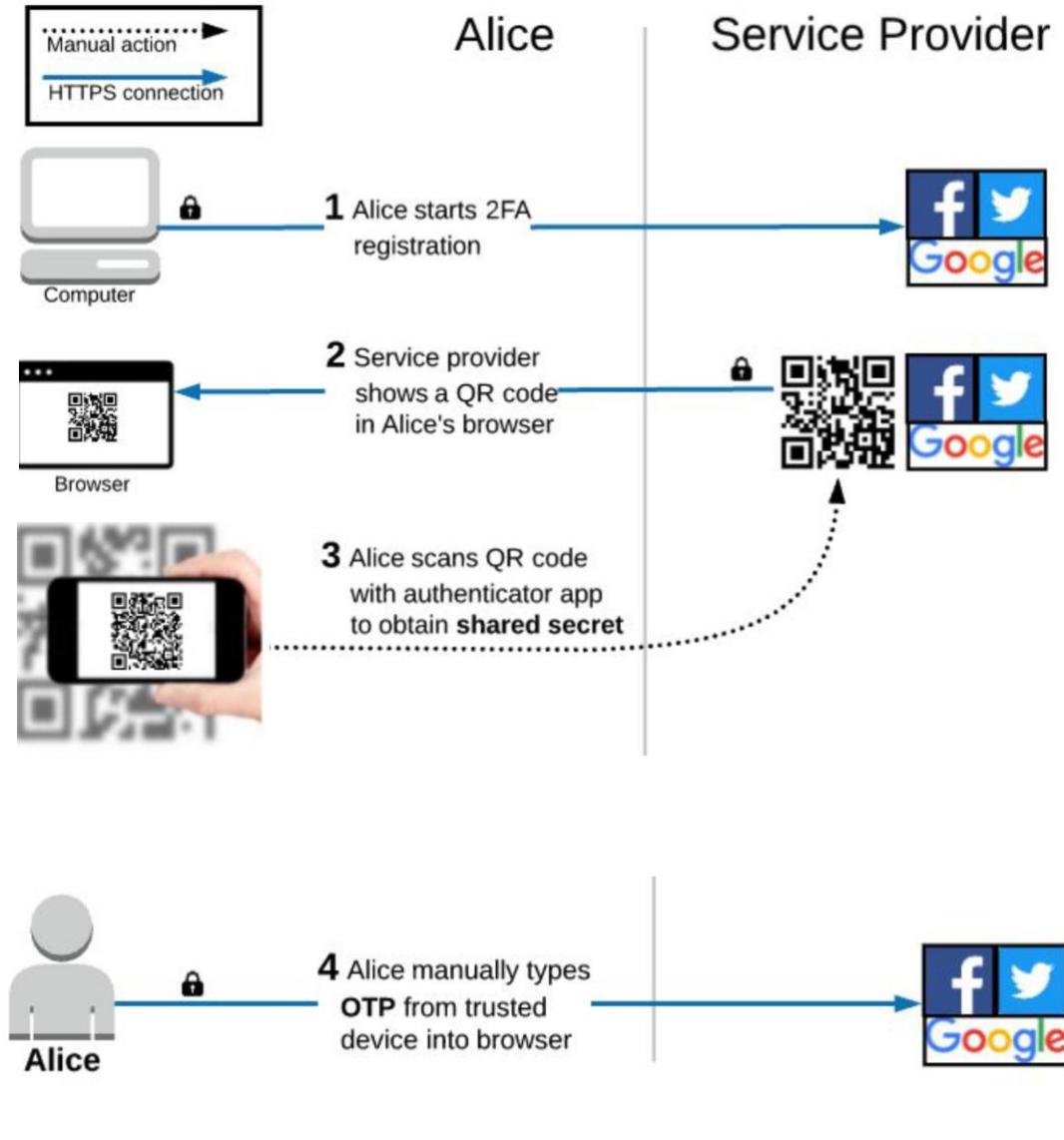
Web Authentication API  
WebAuthn

# SMS – 2FA Authentication Flow

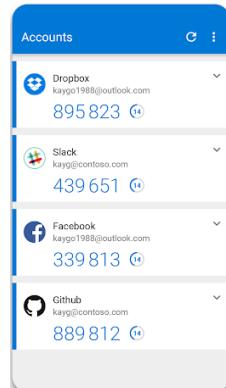


# Time-based One-Time Password (TOTP)

## Registration Flow

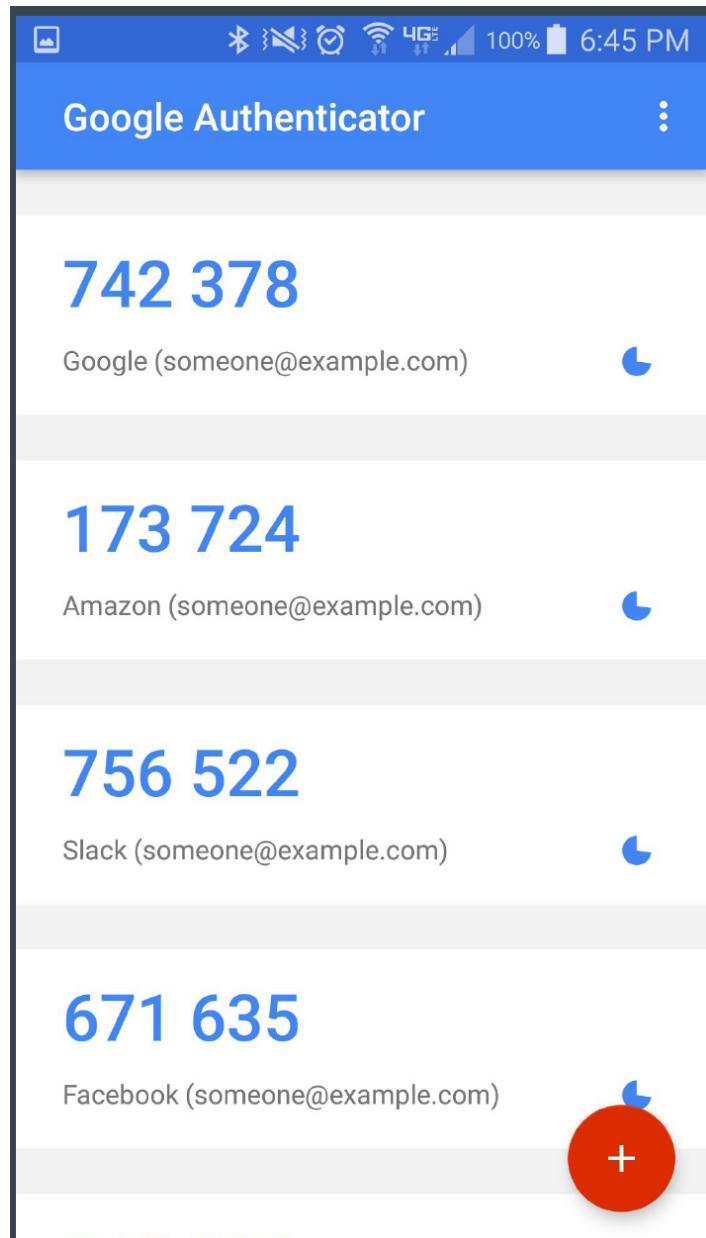


## Authentication Flow

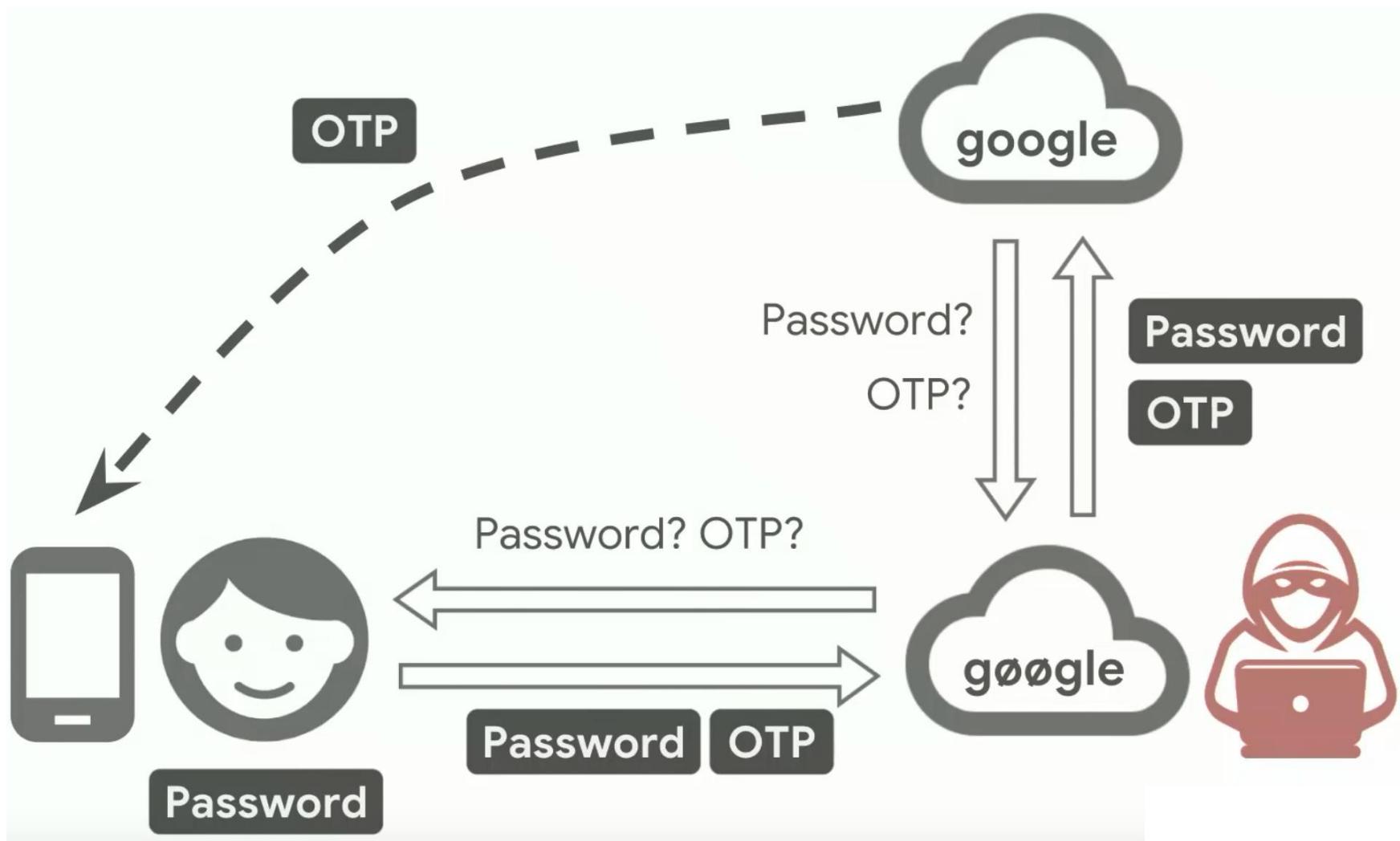


**TOTP = HMAC-SHA-1 (shared secret + time)**

# TOTP: Example Authenticator Apps



# SMS and OTP are still vulnerable to Phishing Attacks



# Push: Authentication prompt



Google

Trying to sign in from  
another computer?



mrscreencast@allthingsauth.com

**Device**

Intel Mac OS X 10\_12\_6

**Near**

Summit, NJ, USA

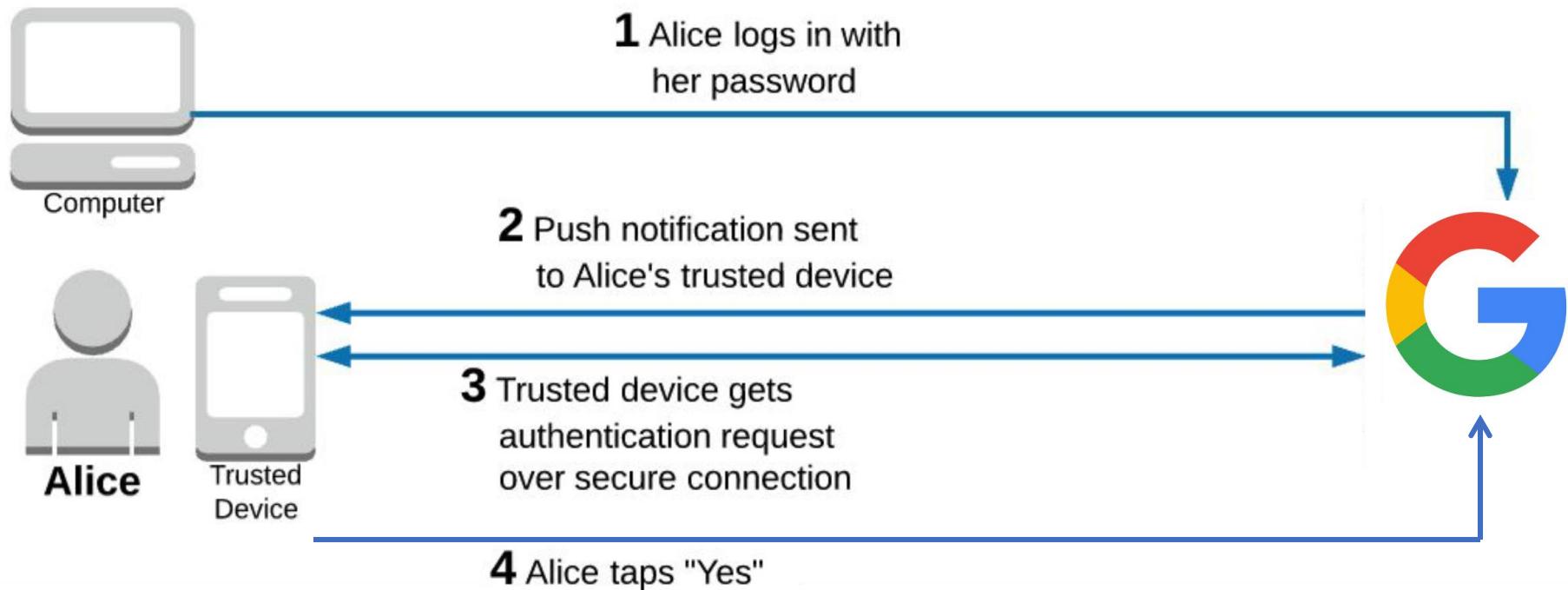
**Time**

Just now

NO

YES

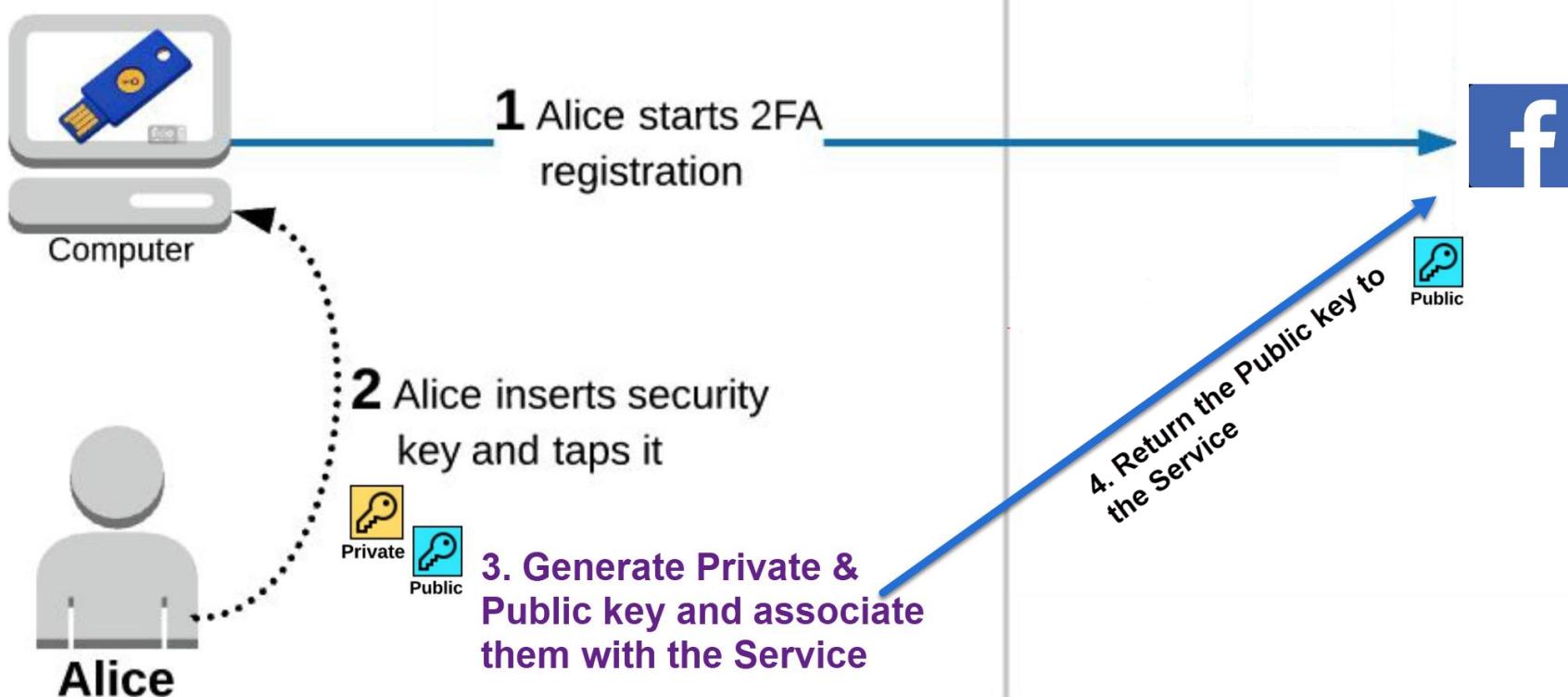
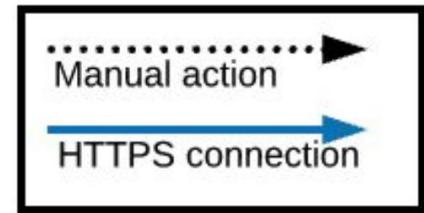
# Push: Authentication flow



- Secure communication over HTTPS using public key cryptography
- Other solutions: Duo, Authy

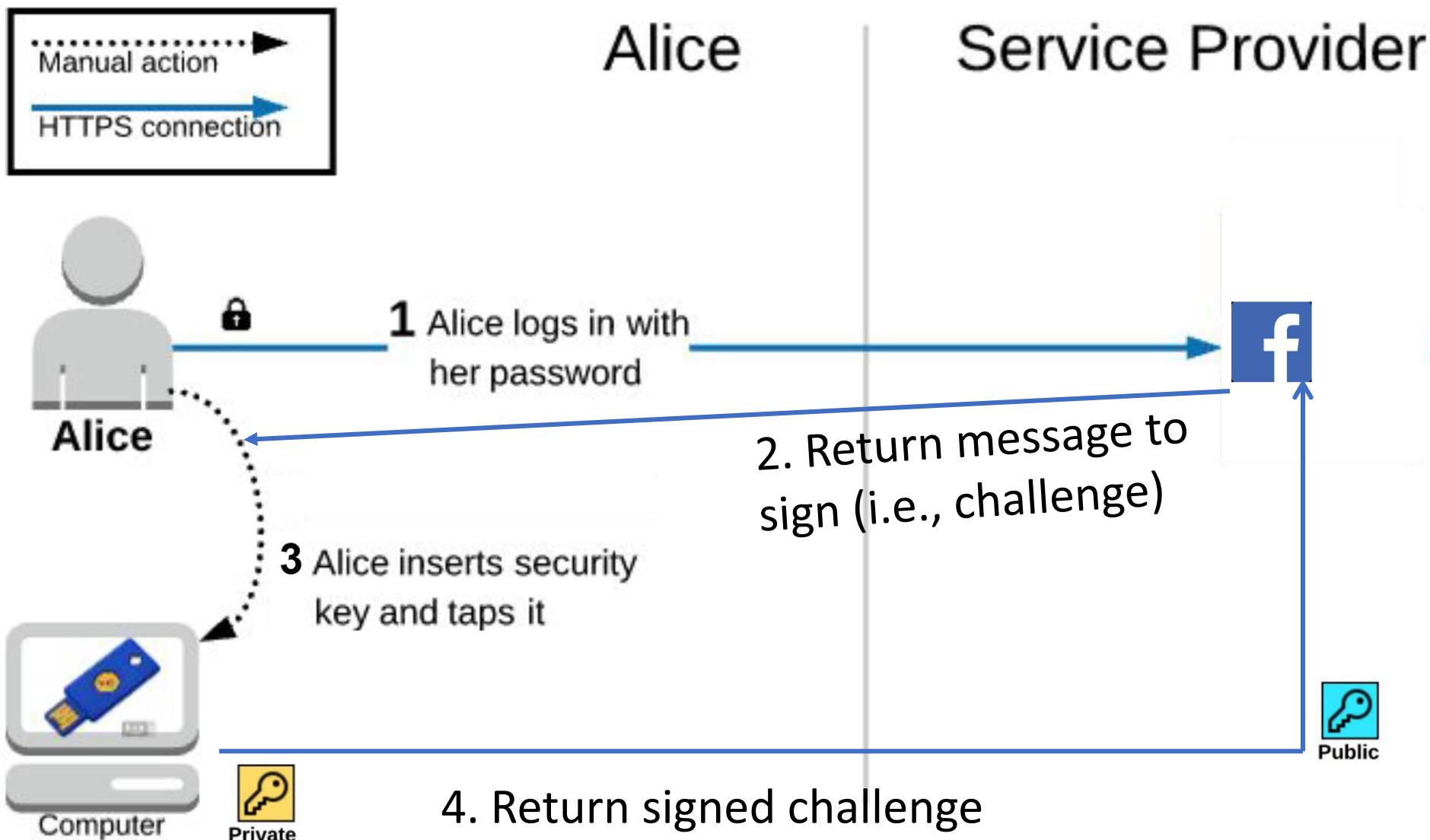


# U2F: Registration flow



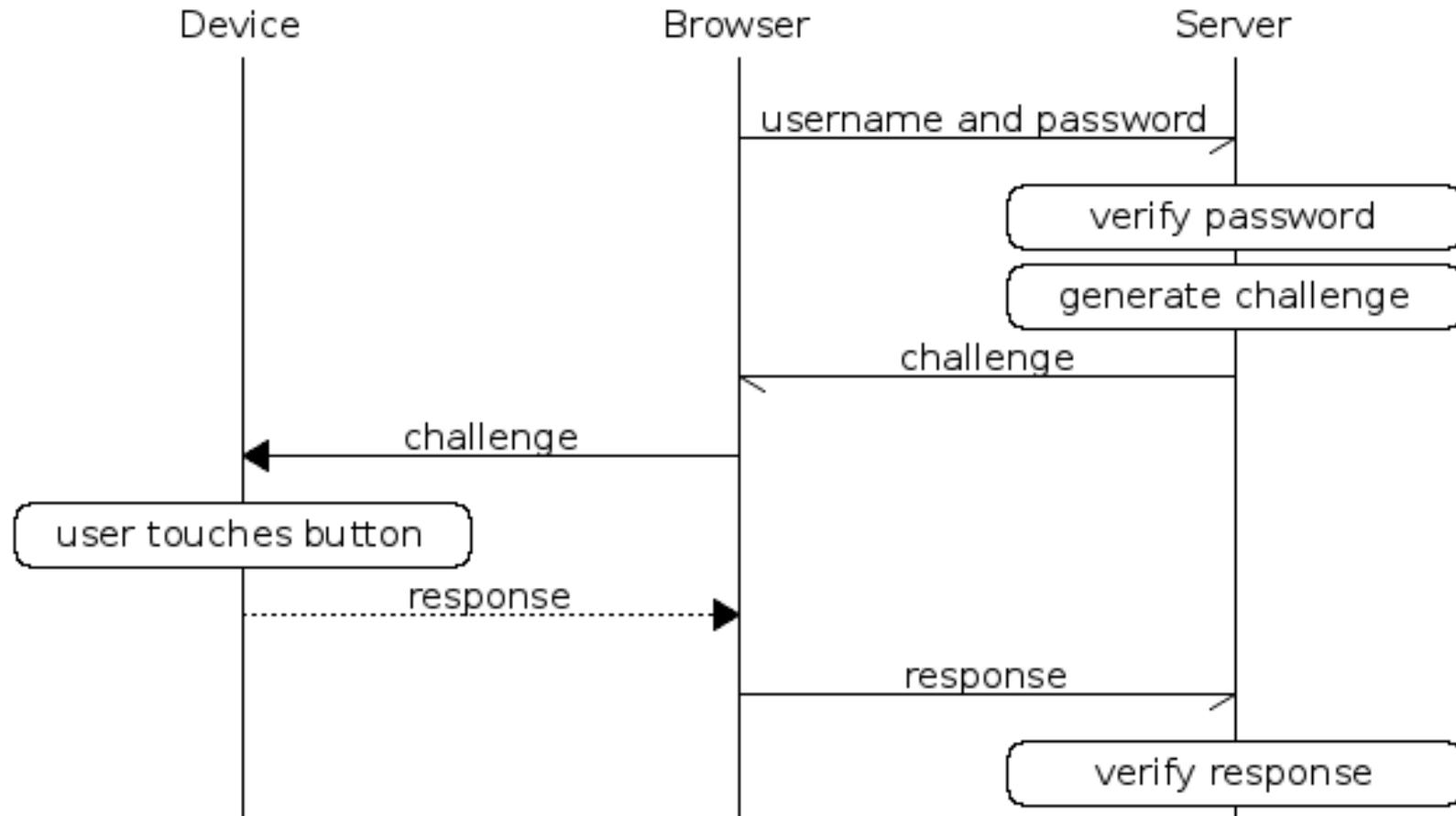


# U2F: Authentication flow





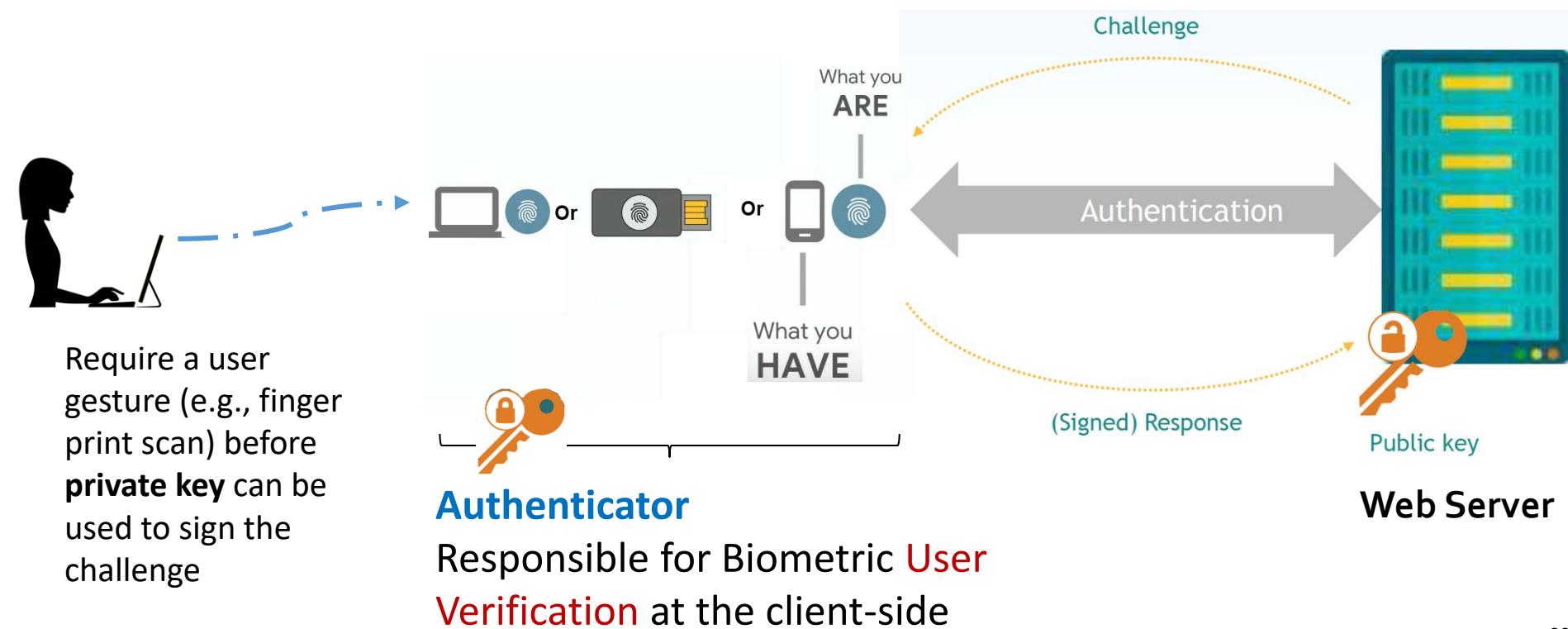
# U2F: Authentication flow (simplified)



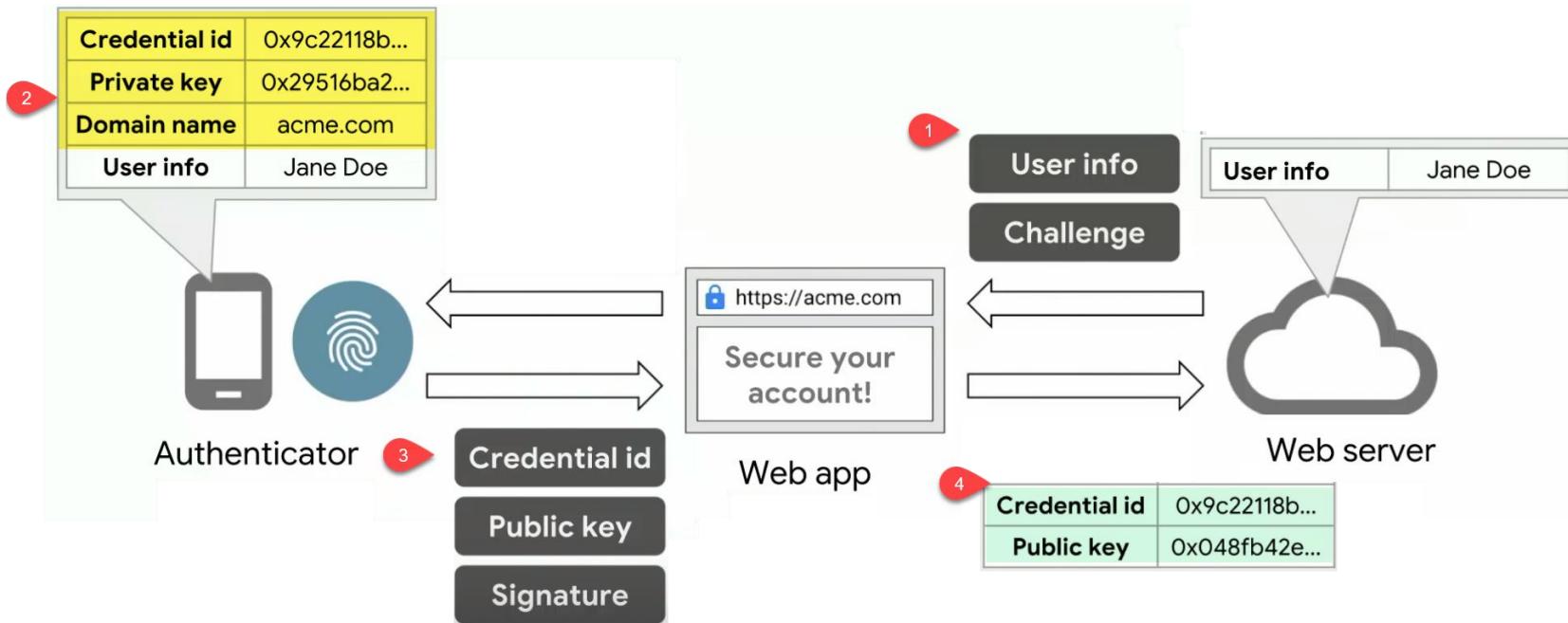
# WebAuthn - Web Authentication API

WebAuthn uses Public key cryptography for **passwordless authentication**

- Unbreakable security & Great user experience 
- Requires a one time setup flow to register an **Authenticator** with the user account
- Every time the user wants to authenticate they **have to prove to the server that they possess the private key**. This is done through a challenge-response based protocol.

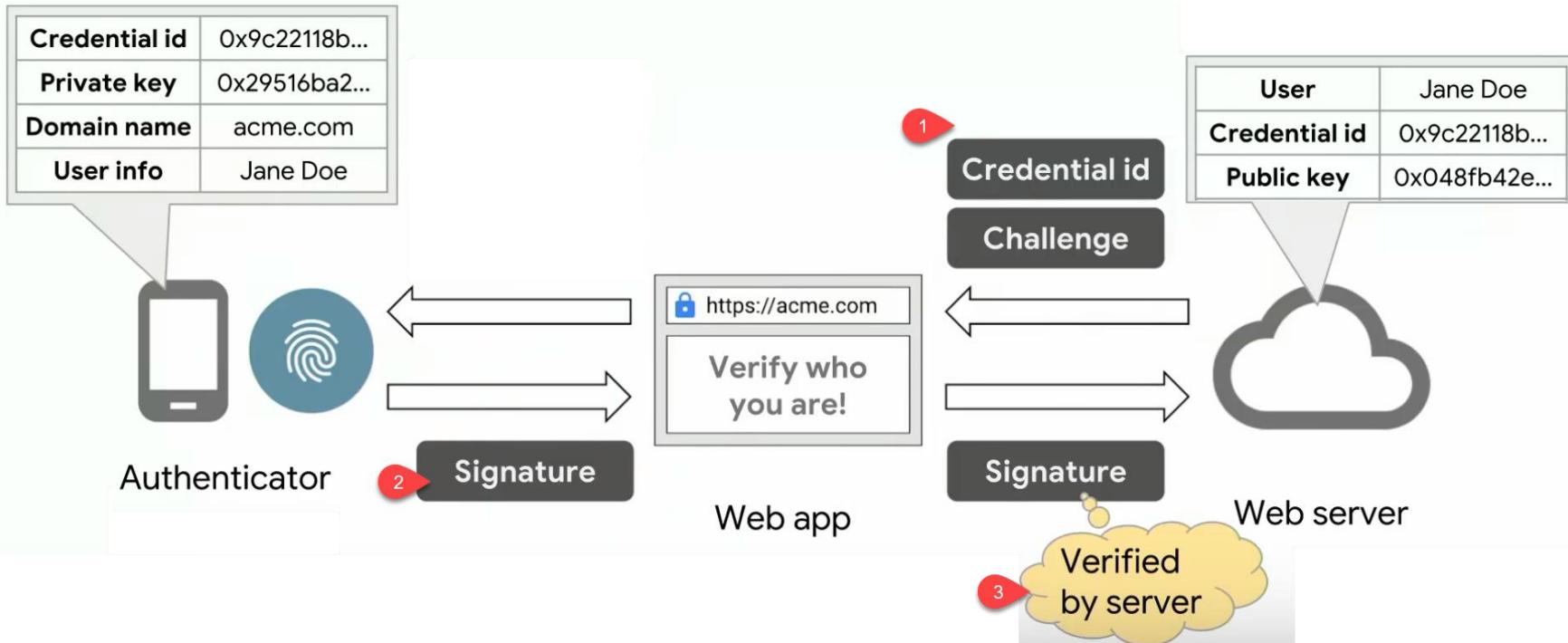


# WebAuthn - Registration Flow



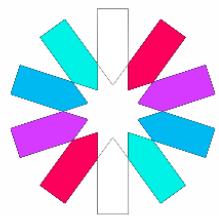
1. The Web server first generates a challenge (i.e., a random number) and sends it along with the User info to the Web app running in the browser.
2. The Authenticator asks for the **user consent** (e.g., figure print scan) then generates a **Public Key and Private Key pairs**. It store the private key locally along with the Credential id, User info and the Domain name.
3. The Authenticator **signs** the Credential id, the **Public key** and the Challenge and transmits them to the Web server.
4. The server validates the signature then stores the Public key and the Credential id with the user account.

# WebAuthn - Authentication Flow



- Every time the user wants to authenticate they have to prove to the server that they possess the private key. This is done through a challenge-response based protocol:
  1. The Web server first generates a challenge (i.e., a random number) and sends it along with the Credential id to the Web app running in the browser.
  2. The Authenticator asks for the **user to authenticate** (e.g., figure print scan). Then it uses the private key to sign the challenge and send the signature to the server.  
The finger print never leaves the device, it is only used locally to verify the user.
  3. The server validates the signature using the user's public key.

# **Single Sign-On (SSO) using Token-Based Security**



**J W U T**

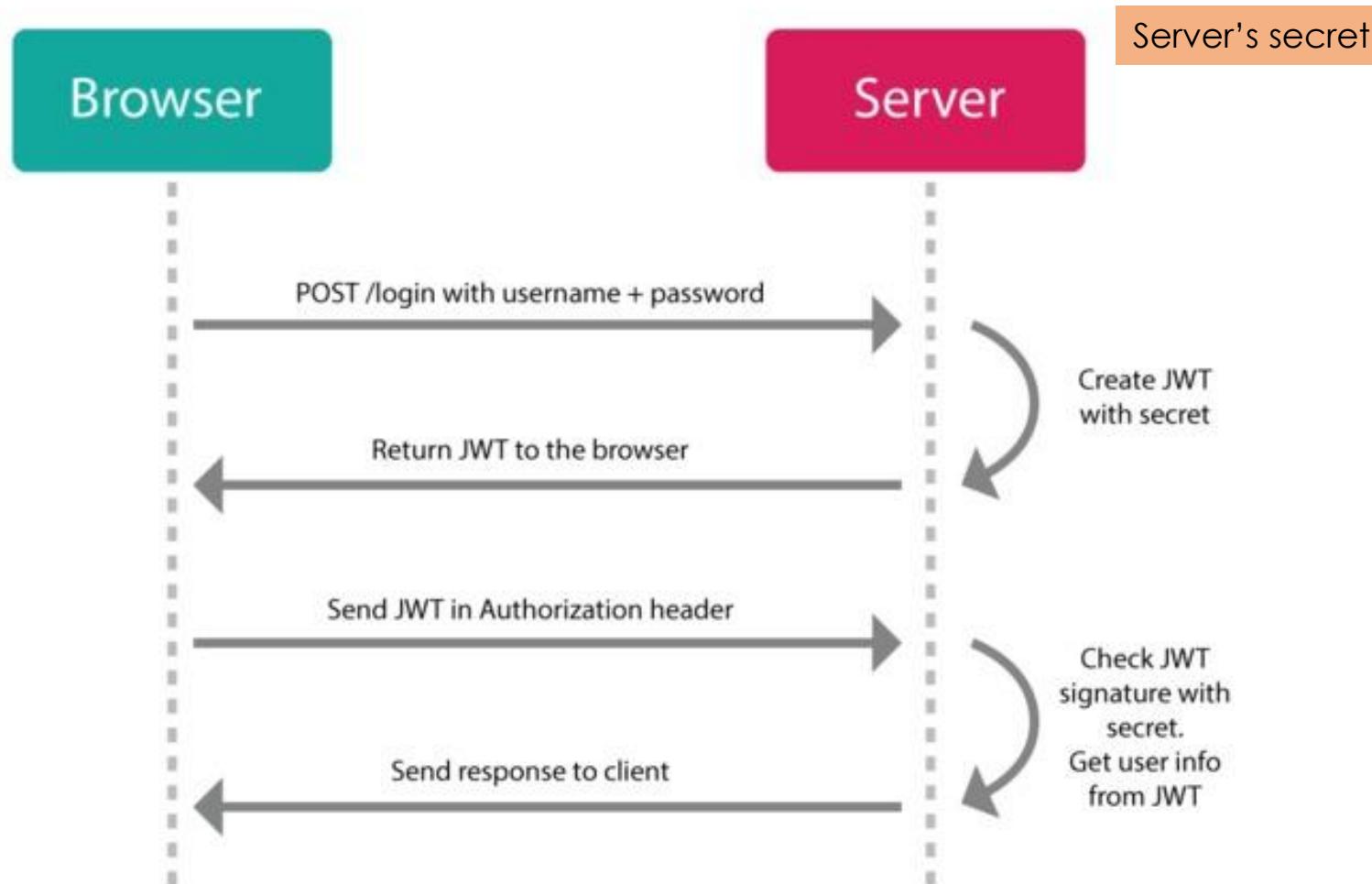
# Token-based Single Sign-On (SSO)

- After a successful authentication a **token** is issued by the server and communicated to the client
- JSON Web Token (JWT) is a widely used token format. It is a **signed json object**
  - Contains claims (i.e., information about the issuer and the user, e.g., user details and their role)
  - Signed (tamper proof & authenticity)
  - Typically contain an expiration time
- JWT is added to the HTTP header of subsequent requests to other Web resources
  - The resource validates the token to authenticate the user (without asking for username and password again)

# Token-based Single Sign-On (SSO)

- JWT enables SSO by sharing the JWT between different applications
  - Web App gets a request that includes a JWT token
  - It checks that the JWT token is valid
  - It uses info in the token to make an access control decision

# JSON Web Token (JWT)



# JWT Structure

## Header

```
{  
  "typ": "JWT",  
  "alg": "HS256"  
}
```

## Claims

```
{  
  role: "Admin",  
  given_name: "Abdelkarim",  
  family_name: "Erradi",  
  name: "erradi",  
  email: "erradi@jwt.org",  
  iat: 1526597430,  
  exp: 1526604630  
}
```

eyJhbGciOiJub251In0.eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzM.D.4MTkz0DAsDQogImh0dHA6Ly91eGft



Header

Claims

Signature

# JWT Issued after Successful Login

- Example sign in <http://localhost:3040/auth/login>

The screenshot shows a Postman request configuration and its resulting response.

**Request Configuration:**

- Method: POST
- URL: <http://localhost:3040/auth/login>
- Body tab selected, type: raw, format: JSON (application/json)
- Body content:

```
1 {
2   "name": "errradi",
3   "password": "secret"
4 }
```

**Response Body:**

- Body tab selected
- Pretty, Raw, Preview, JSON (selected), and a copy icon are shown
- Response content:

```
1 {
2   "idToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJvaWRQcm92aWRlcjI6ImxvY2FsaWZlcnJhZG1Aand0Lm9yZyIsIl9fdiI6MCwiawF0IjoxNTI2NTk"
3 }
```

# Use JWT to Access Protected Resource

- Get users <http://localhost:3040/auth/users>

The screenshot shows a Postman request configuration for a GET request to `http://localhost:3040/auth/users`. The **Headers** tab is selected, containing one entry: **Authorization** with the value `Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJvaWRQcm92aWRlcil6ImxvY2Fsiwi...`. A blue arrow points from this entry to a callout box.

**Add the JWT token to standard **Authorization** header of HTTP requests to allow the Web API to verify it and allow access to resources**

```
1 [ 
2   { 
3     "oidProvider": "local",
4     "role": "Visitor",
5     "defaultUrl": "/",
6     "_id": "5afe02cbac0f4c4260622e46",
7     "given_name": "Abdelkarim",
8     "family_name": "Erradi",
9     "name": "erradi",
10    "email": "erradi@jwt.org",
11    "password": "$2b$10$I/DbUjl0eja.dMnekCLZTOqnBi4xyI9zPCsE3e48xn5Dwpcs45NWe"
12  }
13 ]
```



<https://jwtinspector.io/>

**JWT Inspector** is a chrome extension that lets you **decode** and **inspect** JWT in requests, and local storage

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJv  
aWRQcm92aWRlcii6ImxvY2FsIiwicm9sZSI6IlZpc  
2l0b3IiLCJkZWhdWx0VXJsIjoiLyIsIl9pZCI6Ij  
VhZmUwMmNiYW MwZjRjNDI2MDYyMmU0NiIsImdpdmV  
uX25hbWUiOijBYmRlbGthcmIiwiZmFtaWx5X25h  
bWUiOijFcnjhZGkiLCJuYW1IjoiZXJyYWRpIiwiZ  
W1haWwiOijlcnJhZG1Aand0Lm9yZyIsIl9fdiI6MC  
wiaWF0IjoxNTI2NTk2NDQ1LCJleHAI0jE1MjY2MDM  
2NDV9.fwG\_o7zbvdEIRnNifQ5Bj8sZ5Q4VxtaC5c6  
VPbxADLc

▼ Header

```
{  
  alg: "HS256",  
  typ: "JWT"  
}
```

▼ Payload

```
{  
  oidProvider: "local",  
  role: "Visitor",  
  defaultUrl: "/",  
  _id: "5afe02cbac0f4c4260622e46",  
  given_name: "Abdelkarim",  
  family_name: "Erradi",  
  name: "erradi",  
  email: "erradi@jwt.org",  
  _v: 0,  
  iat: 1526596445,  
  exp: 1526603645  
}
```

▼ Signature

```
fwG_o7zbvdEIRnNifQ5Bj8sZ5Q4VxtaC5c6VPbxADLc
```

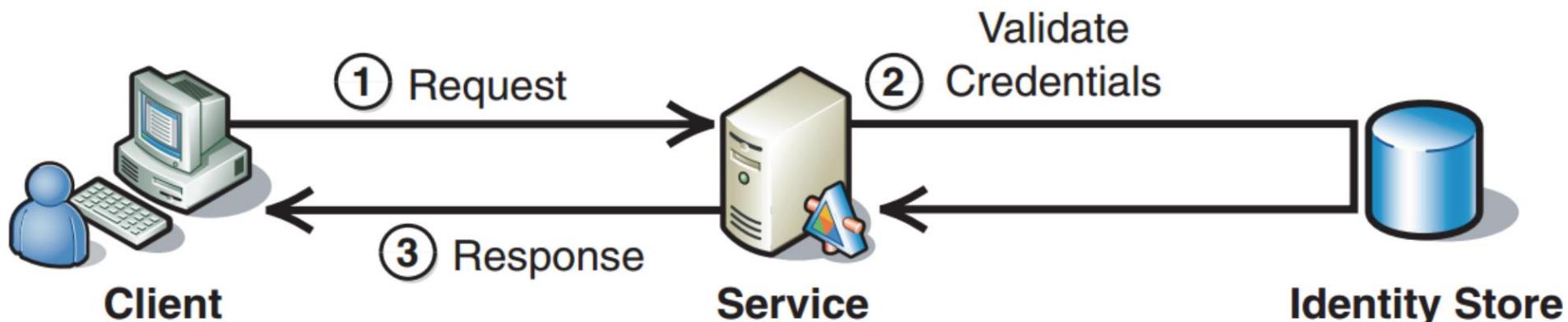
# Delegated Authentication





# Direct Authentication

- The App implements its own authentication and the **server maintains the user identity**.



Authentication Required  
http://localhost:5000

Username

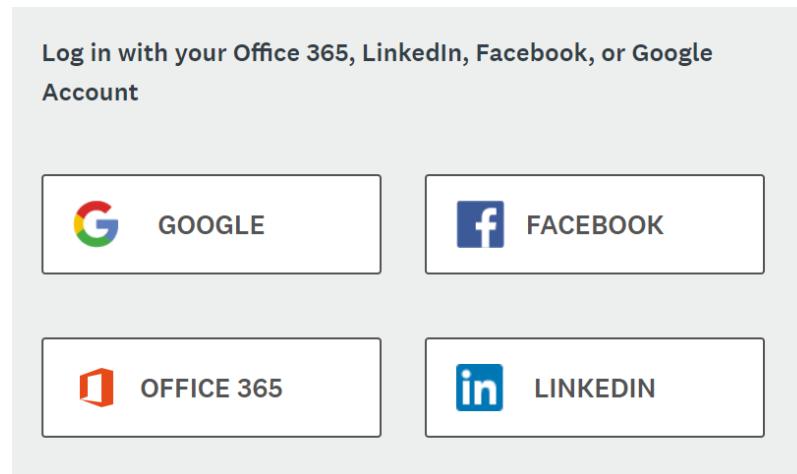
Password

# Authentication is hard

- Trying to write your own login system is difficult:
  - Need to save passwords securely
  - Provide recovery of forgotten passwords
  - Make sure users set a good password
  - Detect logins from suspicious locations or new devices
  - etc.
- **Luckily, you don't have to build your own authentication!**
- You can use **OpenID Connect** to delegate login to an **Identity Provider** and get the user's profile

# OpenID Connect

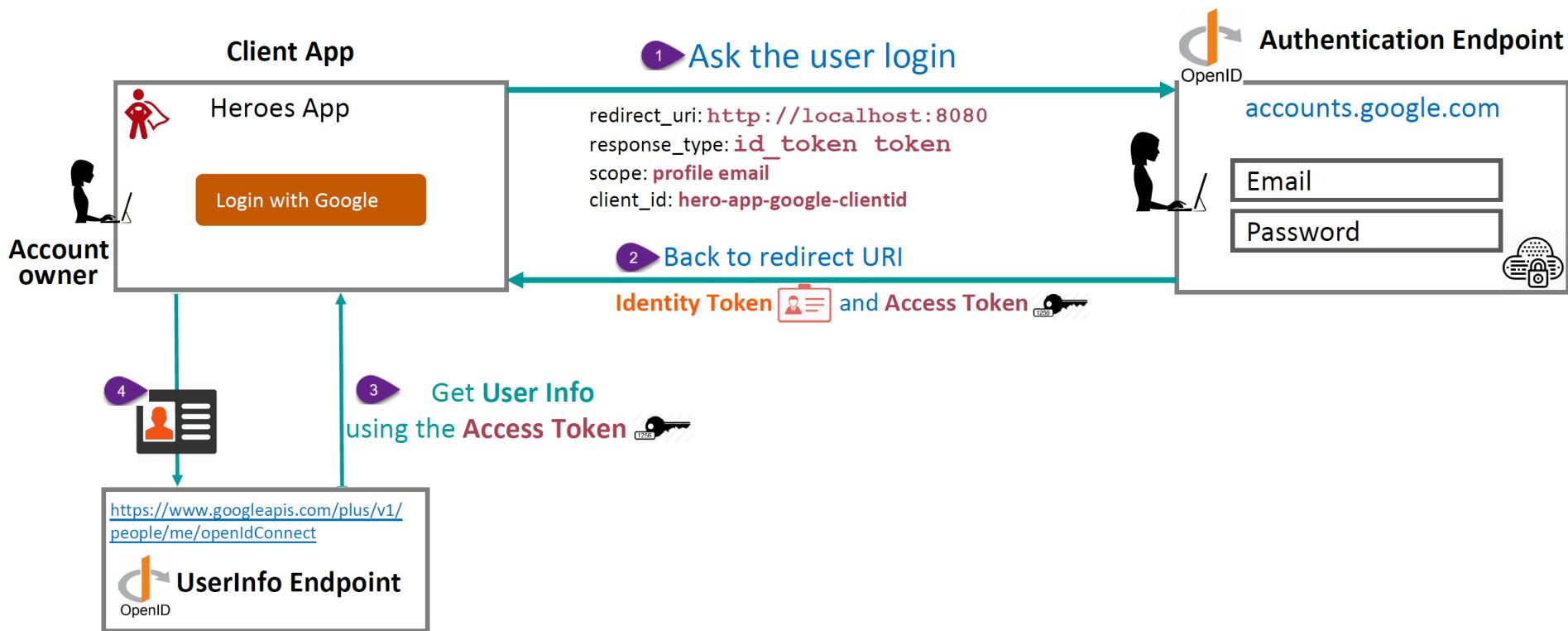
- **OpenID Connect** is a standard for user authentication
  - For users:
    - It allows a user to log into a website like AirBnB via some other service, like Google or Facebook
  - For developers:
    - It lets you authenticate a user without having to implement log in
  - Examples: "Log in with Facebook"



# OpenID Connect APIs

- Companies like Google, Facebook, Twitter, and GitHub offer OpenID Connect APIs:
  - [Google Sign-in API](#)
  - [Facebook Login API](#)
  - [Twitter Login API](#)
  - [GitHub Apps/Integrations](#)
  - OpenID Connect is standardized, but the API that these services provide are slightly different
  - You must read the documentation to understand how to connect via their API
- After the user logins, you will get the user profile such name, email, etc

# OpenID Connect Authentication Flow



# Authenticating via a SPA App

- **User** starts the flow by visiting a Web App
- **Client** sends authentication request with *profile* scope via browser redirect to the **Authorization endpoint**
- **User** authenticates and consents to **Client** to access user's identity
- **ID Token** and **Access Token** is returned to **Client** via browser redirect
- **Client** optionally fetches additional user info with the **Access Token** from **UserInfo endpoint**

# Authorization Request

- Ask the user to login via browser redirect to the Authentication Endpoint

<https://accounts.google.com/o/oauth2/auth>

The image shows a screenshot of a Google sign-in page. At the top, there's a "Sign in with Google" button with the Google logo. Below it, the text "Hi Abdelkarim" and the email "drerradi@gmail.com" are displayed. A password input field is present with the placeholder "Enter your password". Below the input field is a "Forgot password?" link. To the right of the input field is a blue "NEXT" button.

- This will return an **Access Token** to the client to allow it to request the user's profile from the UserInfo Endpoint

# Authentication Parameters

GET ▾

[https://accounts.google.com/o/oauth2/auth?scope=profile%20email%20phone&client\\_id=866457396346-piq09ek9kiofq9uspsnjuvqfj4nnpn&response\\_type=id\\_token%20token&redirect\\_uri=http://localhost:8080](https://accounts.google.com/o/oauth2/auth?scope=profile%20email%20phone&client_id=866457396346-piq09ek9kiofq9uspsnjuvqfj4nnpn&response_type=id_token%20token&redirect_uri=http://localhost:8080)

Params

Send ▾

Scope = what user info the client needs access to?

scope

Value

profile%20email%20phone

client\_id

866457396346-piq09ek9kiofq9uspsnjuvqfj4nnpn

Need to register and get **client\_id** from <https://console.developers.google.com/apis/credentials>

response\_type

id\_token%20token

redirect\_uri

http://localhost:8080

Redirect\_urrl = callback address google will use to deliver to access\_token and id\_token

What is the desired response type to get from the Authentication EndPoint?

- **id\_token**: jwt of the authentication user
- **token**: access-token to be able to access the UserInfo endpoint

Body

Cookies (2)

Headers (15)

Test Results

Status: 200 OK

Time: 461 ms

Size: 72 KB



## One account. All of Google.

Sign in with your Google Account

Email or phone

Next

# ID Token

- JWT representing logged-in user

Example **ID Token** from Google

```
{  
  iss: "accounts.google.com",  
  aud: "lv1muk.apps.googleusercontent.com",  
  sub: "111893194175723488203",  
  email: "karimerradi@gmail.com",  
  email_verified: true,  
  exp: 1526656174,  
  iat: 1526652574  
}
```

- Claims:

iss - Issuer

sub - User Identifier

aud - Audience for ID Token

exp - Expiration time

iat - Time token was issued

# Calling the UserInfo Endpoint

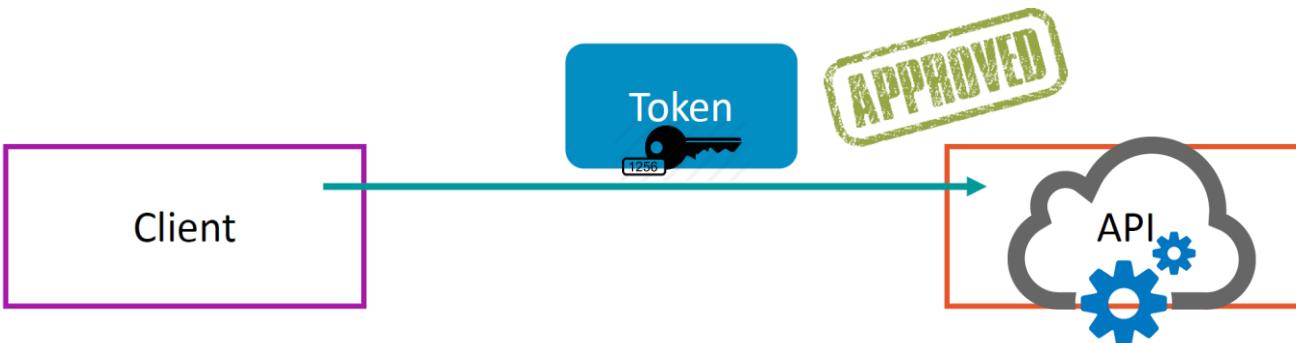
- Get the user's profile from the UserInfo Endpoint

The screenshot shows a Postman request configuration for a GET request to `https://www.googleapis.com/plus/v1/people/me/openIdConnect`. The **Headers** tab is selected, containing a single entry for the **Authorization** header with the value `Bearer ya29.Gly_BcvwEQdaZgHKJlk2nB7N2g3falZmpCxp3NXM7UjoWxou_1Jp4v...`. The **Body** tab displays the JSON response in **Pretty** format:

```
1 {  
2   "kind": "plus#personOpenIdConnect",  
3   "gender": "male",  
4   "sub": "111893194175723488203",  
5   "name": "Erradi",  
6   "given_name": "Erradi",  
7   "family_name": "",  
8   "profile": "https://plus.google.com/111893194175723488203",  
9   "picture": "https://lh6.googleusercontent.com/-iuZD8qYF0xQ/AAAAAAAAGIM/l135MtiUkJ8/photo.jpg?sz=50",  
10  "email": "karimerradi@gmail.com",  
11  "email_verified": "true",  
12  "locale": "en"  
13 }
```

A green callout bubble points to the **Authorization** header in the Headers table, containing the text: "Send the **access token** received after the authentication. Add it to the **Authorization** header."

# Use the Access Token to access Web Resources



- Validate token
- Grant access to the resource

Screenshot of Postman showing a successful API request to Google People API:

- Method: GET
- URL: <https://people.googleapis.com/v1/people/me/connections?personFields=names,emailAddresses,phoneNumbers,addresses,pho...>
- Params: None
- Send button: Clicked
- Headers (1):
  - Key: personFields Value: names,emailAddresses,phoneNumbers,addresses,pho...
- Authorization:
  - Key: Authorization Value: Bearer ya29.GlvABUS5jPiHTOywJTpKr6cCwUEsACshu3...
- Body: None
- Cookies (1): None
- Headers (13): None
- Test Results: Status: 200 OK Time: 253 ms
- Pretty, Raw, Preview, JSON: View options

Sample JSON Response (Pretty Print):

```
1 { "connections": [ 2 { "resourceName": "people/c3945308633452445077", "etag": "%Egg8AgMJCxA3LhoMAQIDBAUGBwgJGgsMIGxuWmJUQU5BT3FKOD0=", "names": [ 3 { "metadata": { "primary": true, "source": { "type": "CONTACT", "id": "36c08d4c8a36fd95" } }, "displayName": "Qatar University", "familyName": "University", "givenName": "Qatar", "displayNameLastFirst": "University, Qatar" 4 ] } 5 ] } 6 }
```

# Summary

- Three types of authentication factors
  1. What the user knows
  2. What the user has
  3. What the user is
- 2 Factors Authentication (2FA) is better

 Web Authentication API (WebAuthn) uses Public key cryptography for passwordless authentication

- JWT is easy to create, transmit and validate to protect Web Apps in a scalable way
- Use **OpenID Connect** for **Delegated Authentication**

# Resources

- NIST Digital Identity Guidelines

<https://pages.nist.gov/800-63-3/>

- Web Authentication API (WebAuthn)

<https://en.wikipedia.org/wiki/WebAuthn>

- JWT Handbook

<https://auth0.com/resources/ebooks/jwt-handbook>

- Authentication Survival Guide

<https://auth0.com/resources/ebooks/authentication-survival-guide>

- What the Heck is OpenID Connect?

<https://www.youtube.com/watch?v=6ypYXxRPKgk>