Lec 17: Web Authentication

CSED415: Computer Security

Spring 2026

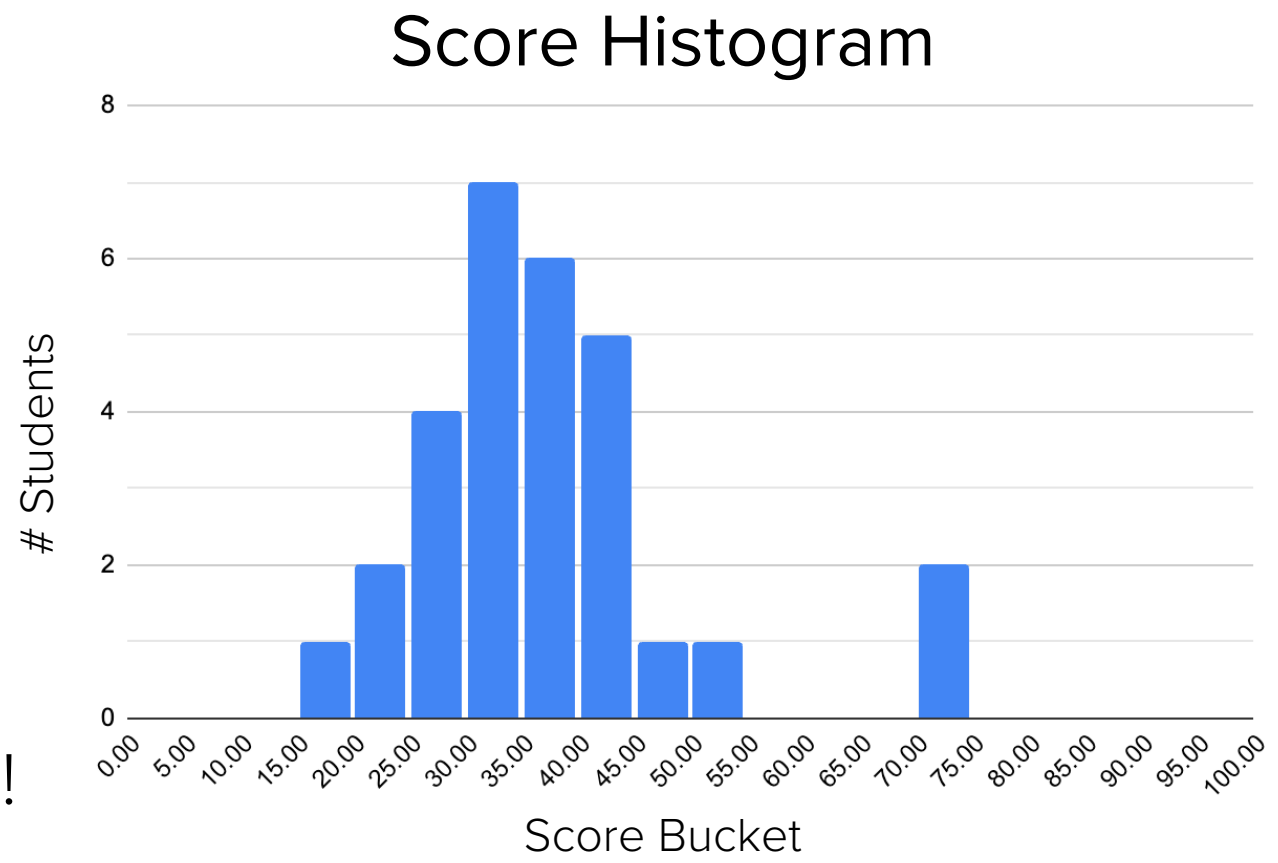
Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Midterm exam
  - Max: 72
  - Min: 16
  - Mean: 37
- Challenge
  - Please email TAs to schedule a challenge session this week!



Administrivia

- Lab 04 has been released
  - Due May 11
  - Topic: Token-based authentication (today's lecture)

Basics of the Web

Background: The World Wide Web

- World Wide Web (WWW): A collection of data and services
  - Web **servers** provide data and services
  - Web **browsers** access the data and services
- Elements of the web
  - URL (Uniform Resource Locator): Address of a unique resource (e.g., a web page) on the web
  - Web page: HTML + CSS + JavaScript
  - HTTP: Standard protocol for browser-server communication

Background: The World Wide Web

- Risks on the Web
  - Risk #1: Server security
    - Web servers should be protected from unauthorized access
    - An attacker should not be able to compromise a server and manipulate its services
  - Example:
    - An attacker modifies Google search results to deliver malicious content
  - Protection: Server-side security
    - Authenticate users before granting access
    - Secure the server against software vulnerabilities

Background: The World Wide Web

- Risks on the Web
  - Risk #2: User machine security
    - A malicious website should not be able to damage or access the user's computer
  - Example
    - When visiting `evil.com`, the site should not be able to read local files on your machine
  - Protection: Browser sandboxing (isolation)
    - Browser runs each website in a restricted execution environment
    - No direct access to OS resources (e.g., your files)

Background: The World Wide Web

- Risks on the Web
  - Risk #3: Protecting cross-site interactions
    - A malicious website should not be able to tamper with your interactions with other websites
  - Example:
    - When visiting `evil.com`, it should not be able to read your emails on `gmail.com` or buy things on your behalf on `amazon.com`
  - Protection: Same-origin policy
    - Each website is isolated by origin (protocol, domain, port)
    - A website cannot directly access the data belonging to other origins

Same-origin policy

- Same-origin policy
  - A rule the prevents one website from tampering with other websites
  - Enforced by the web browser
- Origin
  - The origin of each webpage is defined by its URL:
    - Protocol
    - Domain
    - Port

```
https://compsec.postech.ac.kr:443/assets/logo.png
```

Same-origin policy

POSTECH

- Same-origin policy
  - Two webpages have the same origin if and only if the protocol, domain, and port of the URL all match exactly
  - e.g., `https://compsec.postech.ac.kr` VS.

❌ `http://compsec.postech.ac.kr` → Protocol mismatch

❌ `https://postech.ac.kr` → Domain mismatch

❌ `https://compsec.postech.ac.kr:8080` → Port mismatch

☑ `https://compsec.postech.ac.kr/research` → Same origin!

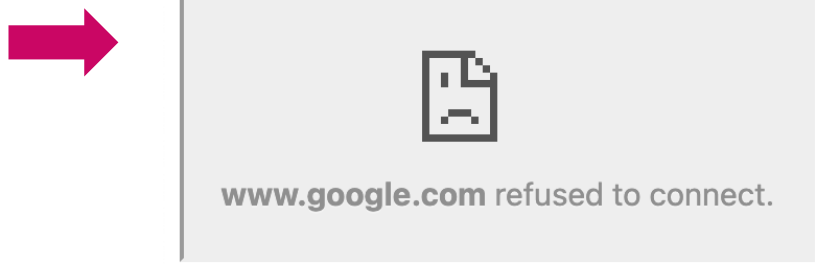
Same-origin policy

- Example

[Code]

```
<html>
<body>
  <h1>This is evil.com</h1>
  <div>
    <script>alert("Hello world");</script>
    <iframe src="https://google.com">
    </iframe>
  </div>
</body>
</html>
```

[Rendering Result]



[Browser Error Message]

```
Refused to display 'https://www.google.com/' in a frame because it set 'X-Frame-Options' to 'sameorigin'.   (index):1

Unsafe attempt to load URL file:///Users/seulbae/Downloads/evil.html from frame with URL file:///Users/seulbae/Downloads/evil.html. 'file:' URLs are treated as unique security origins.   evil.html:1

Unsafe attempt to load URL https://www.google.com/ from frame with URL chrome-error://chromewebdata/. Domains, protocols and ports must match.   (index):1
```

Security VS. Usability

- Security
  - Browser enforces isolation between websites
    - Sandboxing: No access to OS resources
    - Same-origin policy: No access to other sites' data
- Usability
  - Websites need to maintain user identity
    - Stay logged in across multiple requests
    - Share login across related services

We need a mechanism that allows a website to remember you and securely share your identity without directly accessing your machine's data

Cookies

Usability

- Examples
  - I have signed into `gmail.com` weeks ago. Today, I can access `gmail.com` and read my email without logging in again. How does `gmail.com` remember me?
  - I have signed into `gmail.com`. I can access `drive.google.com` on my browser without logging in. How?
  - When I log out of `gmail.com`, I also get signed out of `drive.google.com` immediately. How?

Cookie

- HTTP is a stateless request-response protocol
  - The web server processes each request independently of other requests
- What if we want our responses to be customized?
  - e.g., After enabling dark mode on a website, I want future responses from the website to be in dark mode
  - e.g., After logging into a website, I want future responses from the website to display my account information

→ Cookies are a way to add state to HTTP

Cookie

- Definition: A piece of data used to maintain state across multiple HTTP requests
- Creation
  - The server creates a cookie by including Set-Cookie header in resp
- Storage
  - Web browser stores cookies in a cookie jar
- Transmission
  - Browser automatically attaches cookies to every web request
  - The server uses received cookies to connect related requests and customize responses

Cookie

- Cookie structure
  - `Name` and `Value`: actual cookie data

Name	Theme
Value	Dark
Domain	compsec.postech.ac.kr
Path	/research
Secure	True
HttpOnly	False
Expires	27 Apr 2026, 15:30:00

Cookie

- Cookie structure
  - **`Name`** and **`Value`**: actual cookie data
  - **`Domain`** and **`Path`**: define which requests the browser should attach cookie to

Name	Theme
Value	Dark
Domain	compsec.postech.ac.kr
Path	/research
Secure	True
HttpOnly	False
Expires	27 Apr 2026, 15:30:00

Cookie

- Cookie structure
  - `Name` and `Value`: actual cookie data
  - `Domain` and `Path`: define which requests the browser should attach cookie to
  - `Secure`: browser sends cookie if the request is made over HTTPS
  - `HttpOnly`: if set, JavaScript is not allowed to access the cookie

Name	Theme
Value	Dark
Domain	compsec.postech.ac.kr
Path	/research
Secure	True
HttpOnly	False
Expires	27 Apr 2026, 15:30:00

Cookie

- Cookie structure
  - `Name` and `Value`: actual cookie data
  - `Domain` and `Path`: define which requests the browser should attach cookie to
  - `Secure`: browser sends cookie if the request is made over HTTPS
  - `HttpOnly`: if set, JavaScript is not allowed to access the cookie
  - `Expires`: defines when the cookie is no longer valid. Once invalidated, the browser deletes the cookie

Name	Theme
Value	Dark
Domain	compsec.postech.ac.kr
Path	/research
Secure	True
HttpOnly	False
Expires	27 Apr 2026, 15:30:00

Cookie security issues

- Recall:
  - The server can create a cookie by including a `Set-Cookie` header in its response
  - The browser automatically attaches relevant cookies in every request
- Security issues:
  - A server should not be able to set cookies for unrelated websites
    - `evil.com` should not be able to set a cookie that gets sent to `google.com`
  - Cookies should not be sent to the wrong websites
    - Authentication cookie of a user to `google.com` should not be sent to `evil.com`
    - (We will see how cookies are used for logins later)

Cookie policy

- Cookie policy: A set of rules enforced by the browser
- Cookie policy determines
  - 1) When should the browser accept the cookie sent by a server
  - 2) When should the browser attach the cookie when making a request to a server

Cookie policy

- Cookie policy 1: Setting cookies
  - Server with domain X can set a cookie whose `Domain` attribute is Y if
    - (1) Y is a domain suffix of X, i.e.,
      - X ends in Y or
      - X is below or equal to Y in the hierarchy or
      - X is more specific or equal to Y or
    - and (2) Y is not a top-level domain (TLD)
  - Examples:
    - Server `mail.google.com` can set cookies with `Domain=mail.google.com` or `Domain=google.com`, but not `Domain=com` (`com` is a TLD)

[Cookie Structure]

`Name`	`Theme`
`Value`	`Dark`
`Domain`	`compsec.postech.ac.kr`
`Path`	`/research`

Cookie policy

- Cookie policy 2: Sending cookies
  - The browser sends the cookie if both are true:
    - The `Domain` attribute is a domain suffix of the server's domain
    - The `Path` attribute is a prefix of the server's path
  - Examples:
    - Server URL: `https://compsec.postech.ac.kr/teaching/csed415/2025sp`

Domain	compsec.postech.ac.kr
Path	/research

✗ Path is not a prefix of server's path

Domain	compsec.postech.ac.kr
Path	/teaching

✓ Domain is a suffix, Path is a prefix

Domain	postech.ac.kr
Path	/teaching/misc

✗ Path is not a prefix of server's path

Domain	postech.ac.kr
Path	/teaching

✓ Domain is a suffix, Path is a prefix

Domain	postech.edu
Path	/teaching

✗ Domain is not a suffix of server's domain

Domain	compsec.postech.edu
Path	/teaching

✗ Domain is not a suffix of server's domain

Session-Based Authentication

Session authentication

- Session: A sequence of requests and responses associated with the same authenticated user
  - You log into `gmail.com`
  - You read multiple emails, delete some, and send a few emails
  - Each of these activities are web requests
  - Gmail server needs a way to know all these requests are from you
- Naïve solution: Type your username and password before each request
- Better solution: Make browser automatically send some information in a request, i.e., Cookies!

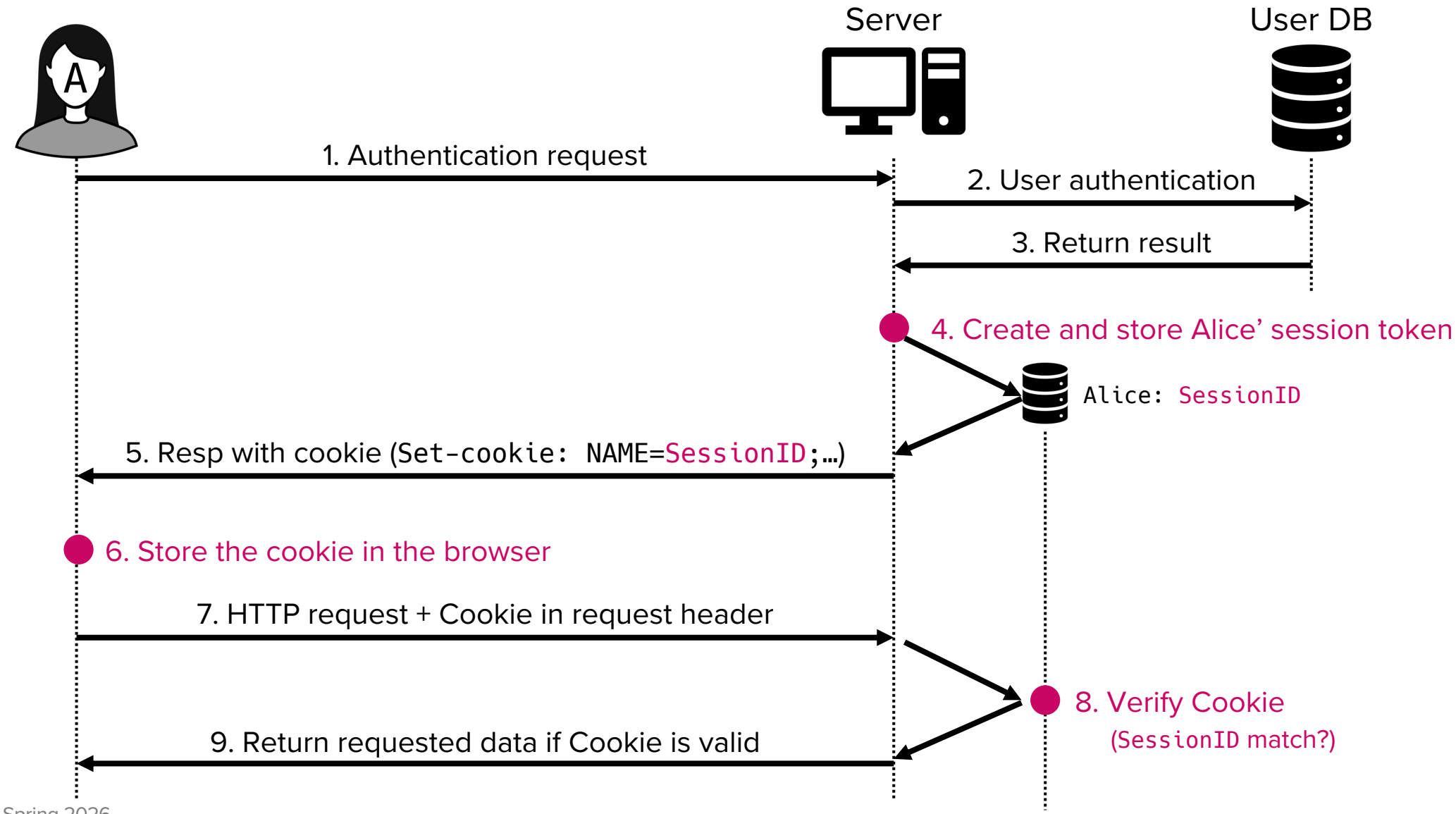
Session tokens

- Session token: A secret value used to associate requests with an authenticated user
  - The first time a user visits the website:
    - User presents username and password
    - Server authenticates the user and sends a **session token**
  - When the user makes further requests:
    - User's browser attaches the session token in the requests
    - The server checks the session token for authentication
    - Authentication without re-entering username and password

Cookies as session tokens

- Session tokens can be implemented with cookies
  - Cookies can be used to save any state across requests
    - Session tokens are just one way to use cookies
  - The first time a user visits the website:
    - User presents username and password
    - Server authenticates the user and sends a cookie with `Value=session_token`
  - When the user makes further requests:
    - User's browser attaches the session token cookie in the requests
    - The server checks the session token in the cookie for authentication
    - Authentication without re-entering username and password
  - Browser and server delete the session token when the user logs out

Cookies as session tokens



Security of session tokens

- If an attacker steals your session token, they can log in as you!
  - By attaching your session token to requests
  - The server will think the attacker's requests are coming from you
- For security,
  - Servers need to generate session tokens randomly and securely
  - HTTPS must be used to encrypt request and response (and tokens)
  - Browsers need to make sure malicious websites cannot steal session tokens
    - Cookie policy + Same-origin policy

Limitations of session-based authentication

- Problems with session token cookies:
  - Server-side state management is expensive
    - Server must maintain a session token-to-user mapping
    - Distributed systems (e.g., service running on multiple servers) need synchronization for cookies
  - Limited portability across domains
    - Cookies are bound by domain policies
    - Difficult to use across independent services

→ Can we avoid server-side session storage? Yes, Using JWT!

→ Can we enable cross-domain authentication? Yes, Using SSO!

JSON Web Token (JWT)

JSON Web Token (JWT)

- JWT:
  - A self-contained authentication token
  - Encodes user identity + metadata
  - Digitally signed for integrity
- Key idea of JWT:
  - Server does not store session state
  - Client stores the signed JWT token and sends it with requests

JWT structure

- A JWT always consists of three parts:
  - Header
  - Payload
  - Signature
- A JWT typically has the following format:

```
HEADER.PAYLOAD.SIGNATURE
```

JWT header

- Header is a JSON object that usually contains two fields
  - alg: Signing algorithm (e.g., HS256, RS256)
  - typ: Type of the token (JWT)
- Header example:

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

→ This JSON is Base64Url-encoded to form the first part of the JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

JWT payload

- Payload is a json object containing “claims”
  - Claims are optional data about the token
  - nbf (not before), exp (expiration time), sub (subject), aud (audience), iss (issuer), iat (issued at), and other data (username, role, …)
- Header example:

```
{
  "username": "David",
  "admin": true
}
```

→ This JSON is Base64Url-encoded to form the second part of the JWT:

```
eyJ1c2VybmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
```

JWT signature

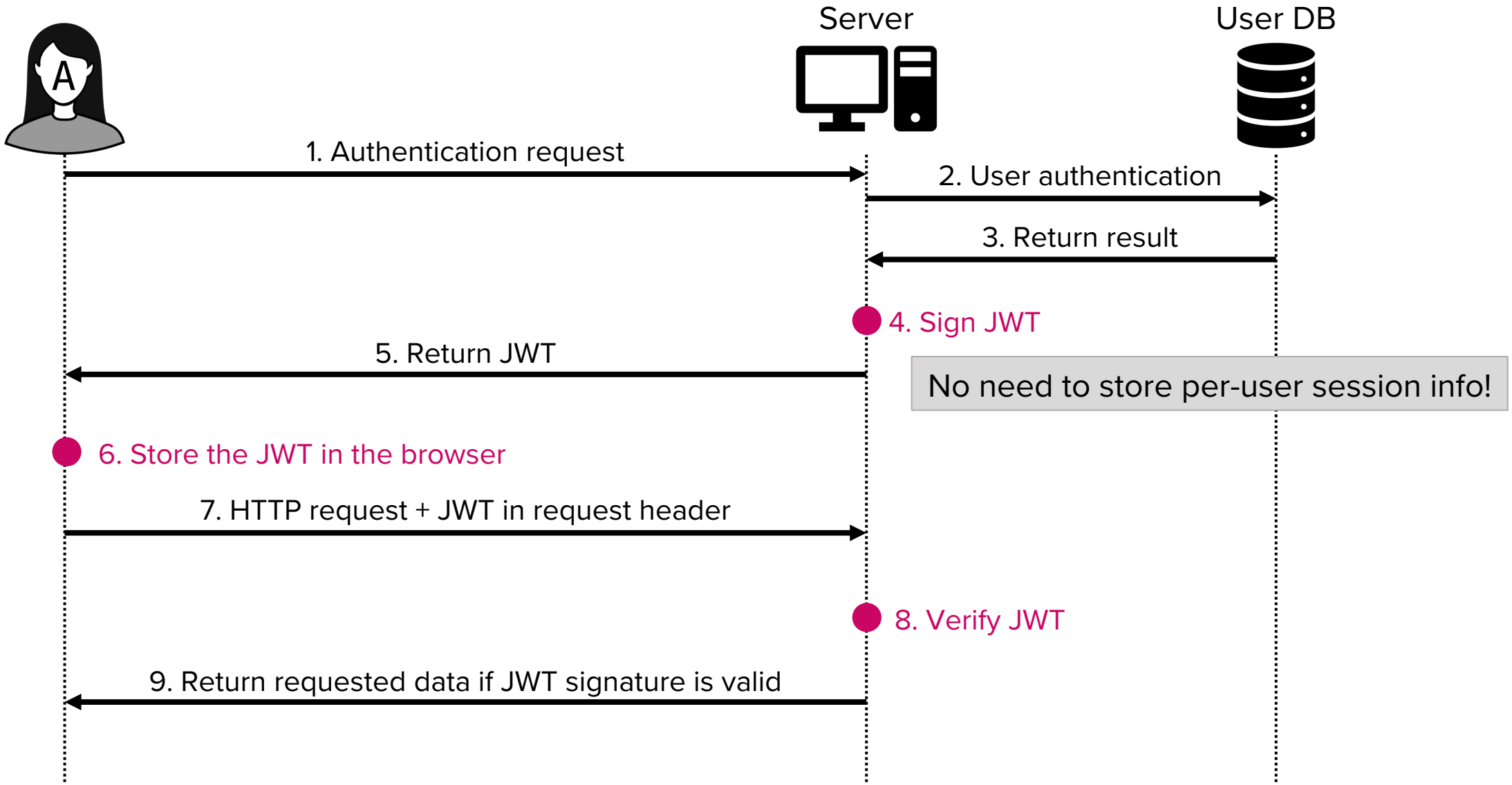
- Signature: `sign(key, header || payload)`
  - Using a key and the signing algorithm specified in the header, sign the encoded header and encoded payload
  - Popular signing algorithms:
    - HS256 (HMAC + SHA256)
      - Both server and user sign and verify JWT using the same secret key
    - RS256 (RSA + SHA256)
      - Server signs using the server's private key
      - Both server and user verify JWT using the server's public key

JWT in practice

- In Python 3 using PyJWT library:

```python
import jwt

private_key = open("private.pem").read()
public_key = open("public.pem").read()

payload = {
  "username": "alice",
  "admin": True
}

# --- SIGN ---
token = jwt.encode(payload, private_key, algorithm="RS256")

# --- VERIFY ---
decoded = jwt.decode(token, public_key, algorithms=["RS256"])
```

JWT authorization



Lab 04

- Breaking JWT-based user authentication
- Tips
  - Get familiar with Chrome Developer Tools
  - Try https://jwt.io

Single Sign-On (SSO)

Single Sign-On (SSO)

- SSO:
  - Authentication scheme that allows a user to log in with a single ID to multiple related, yet independent, services (domains)
  - Examples:
    - Google: Gmail, Google Drive, YouTube
    - Microsoft: Outlook, OneDrive, Teams
    - POSTECH: podium, povis, email

Single Sign-On (SSO)

- Benefits of SSO:
  - Better user experience:
    - Fewer authentication prompts
  - Enhanced security:
    - The user does not have to remember different usernames and passwords, reducing the risk of breaches and phishing
    - Services can monitor and audit all login attempts through an identity server

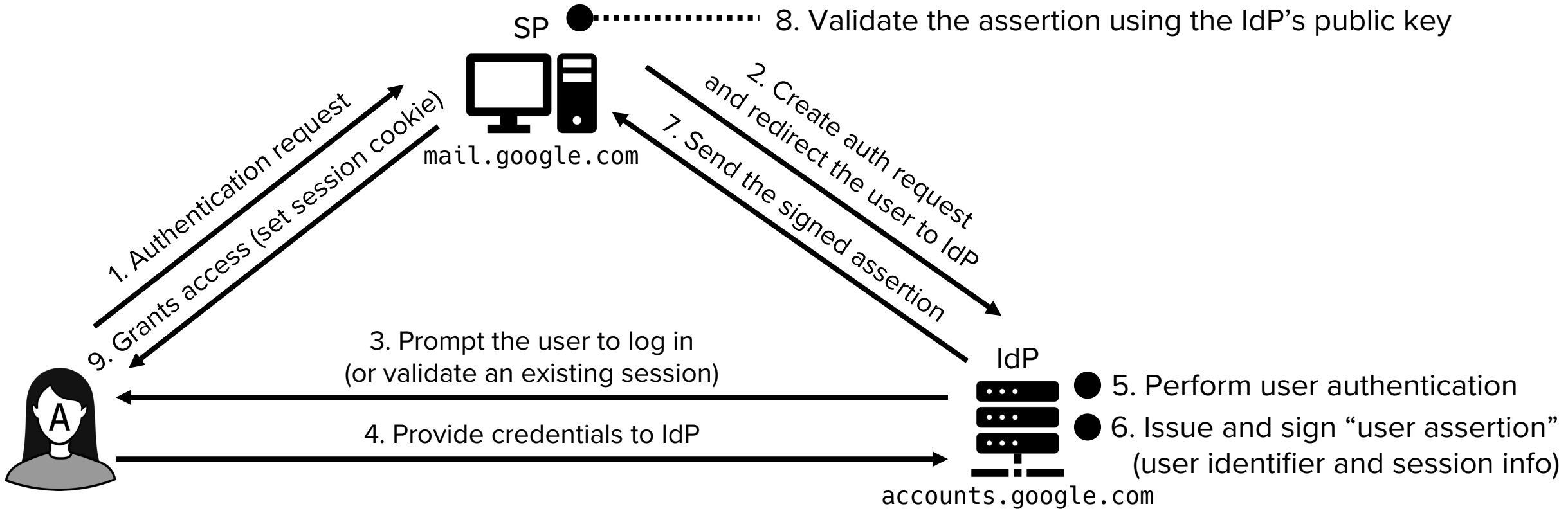
How does SSO work?

Single Sign-On (SSO)

- SSO Entities
  - User (browser)
  - Service Provider (SP)
    - e.g., Gmail, Google Drive
  - Identity Provider (IdP)
    - e.g., accounts.google.com
- Key idea
  - Services do not authenticate users themselves
  - They delegate authentication to a central IdP

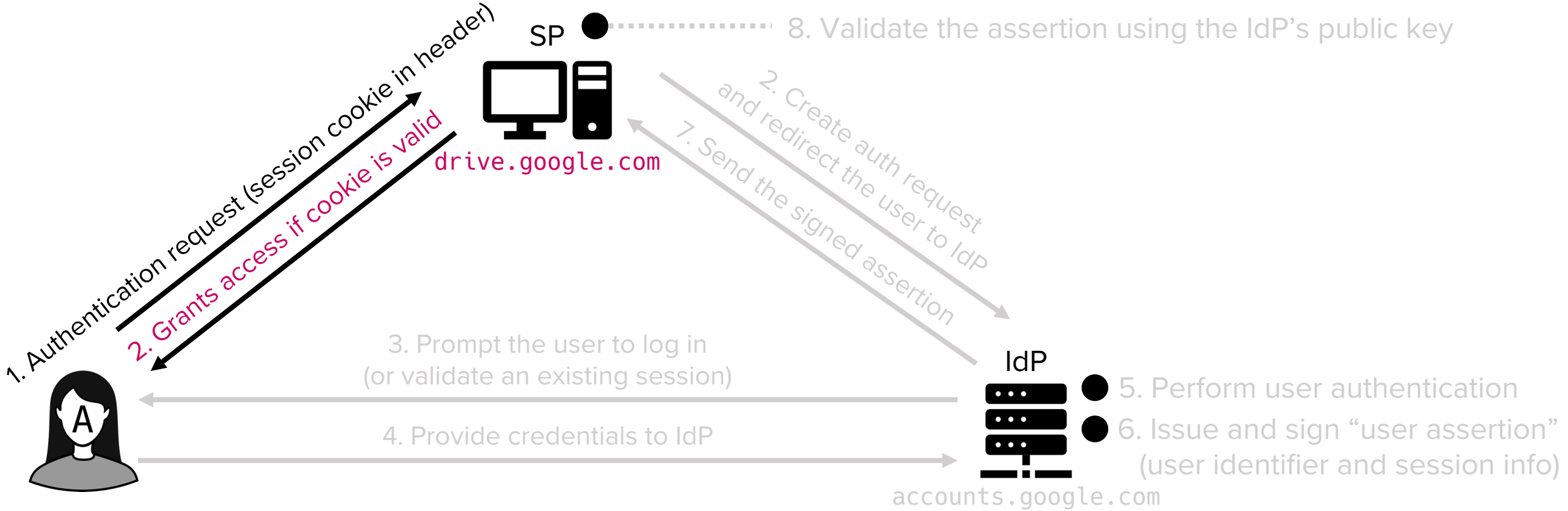
Single Sign-On (SSO)

- SSO workflow example
  - Initial log in



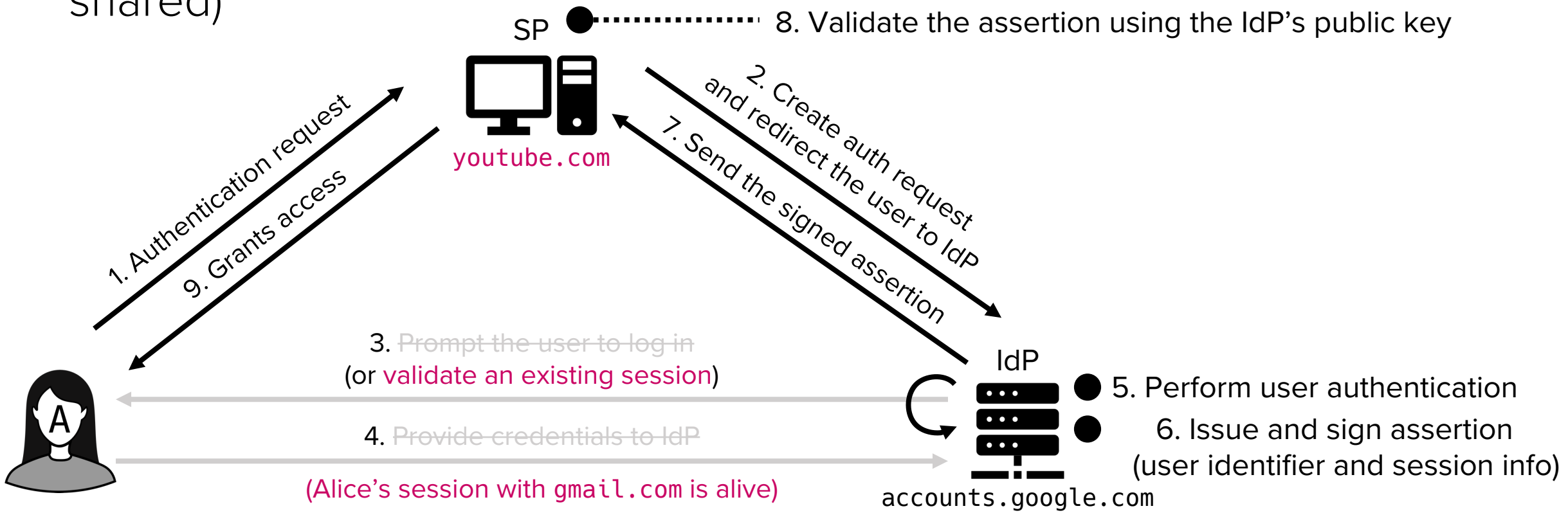
Single Sign-On (SSO)

- SSO workflow example
  - Same domain login without re-authentication using session cookie



Single Sign-On (SSO)

- SSO workflow example
  - Cross-domain login without re-authentication (Note: cookie cannot be shared)



Summary

- From stateless web to shared identity
  - HTTP does not remember users
    - Cookies enable states
  - Session-based authentication using cookies is simple but hard to scale and share
  - JWT: Client carries signed identity
    → No server-side session state needed
  - SSO: Authentication is delegated to an IdP
    → One login across multiple services

Coming up next: Attacks on web authentication

Q & A