香港中文大學
The Chinese University of Hong Kong

MORE ON WEB SECURITY

ESTR2106 2022–23 Term 1

Building Web Applications

Dr. Chuck-jee Chau
chuckjee@cse.cuhk.edu.hk

OUTLINE

- Handling user passwords
- HTTP authentication
- JSON Web Tokens (JWT)
- More on security risks
- CORS

AUTHENTICATION FOR WEB APPS

- Membership is one important feature in apps and services, but how to check the ***identity of users***?

HTTP Authentication	Session/token based	Delegating/Decentralizing
• HTTP Basic/Bearer/Digest authentication • User/password pairs to be checked • Stateless: resending all data in every request	• Authenticated with user/password pairs • Stateful: user info stored on server or client	• OpenID Connect / OAuth 2.0 • User identity being checked by a ***third party***, e.g., "Sign in with Google" • More robust if set up properly
Well supported, not preferred	***Currently most preferred***	Outsourcing – is it good?

See: https://testdriven.io/blog/web-authentication-methods/

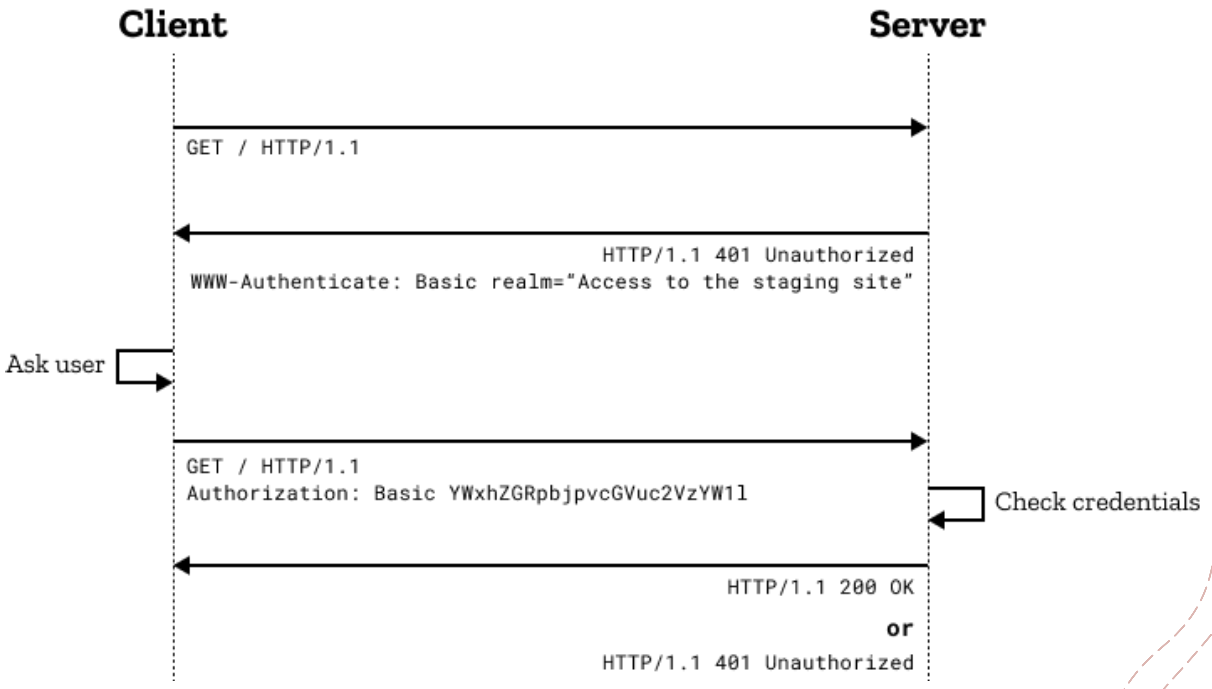
HANDLING USER PASSWORDS

- Passwords should NEVER be stored in plain text
  - ***Hashing***: a one-way function transforms the password into hashed text, which is good enough for validation without storing the actual password
  - ***Encryption***: two-way function which makes the actual password retrievable, not preferred
- Improving password storage
  - ***Salting***: unique random string appended to password before hashing
  - ***Peppering***: an extra encryption with the pepper key, stored separately
- *See: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html*
- *See: https://www.vaadata.com/blog/how-to-securely-store-passwords-in-database/*

HTTP AUTHENTICATION

1. If needed, server can send response with header **`WWW-Authenticate`** with a challenge
2. Usually, a browser will show a dialog for user to enter credentials
3. Client includes **`Authorization`** header in next request, with credentials

See: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication



```
GET / HTTP/1.1

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Access to the staging site"

GET / HTTP/1.1
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l

HTTP/1.1 200 OK
or
HTTP/1.1 401 Unauthorized
```

HTTP AUTHORIZATION SCHEMES

- In the Authorization request header, different schemes are allowed, e.g.:
  - *Basic*
    - The *username:password* string is base64 encoded
  - *Bearer*
    - A bearer token is provided as an encrypted string for server to process
    - Examples: JWT, OAuth 2.0
  - *Digest*
    - Username and password are MD5 hashed before sending
    - Note: MD5 is now considered cryptographically broken, but still useful as a checksum
  - *HOBA*
    - HTTP Origin-Bound Authentication: based on digital signatures at client

JSON WEB TOKENS (JWT)

- JWT gets popularity as a way to generate authentication tokens
- JWT always has 3 parts with a dot in between
  - Header (algorithm and token type)
  - Payload (the actual contents)
  - Signature (the header+payload encoded, signed with server's private key)
- The 3 parts are encoded separately in Base64, e.g.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dnZWRJbkFzIjoiYWRtaW4iLC
JpYXQiOjE0MjI3Nzk2Mzh9.gzSraSYS8EXBxLN_oWnFSRgCzcmJmMjLiuyu5CSpyHI
```

  - Try the debugger at https://jwt.io
- Server creates JWT for the user and send it to the client

→ then the token is sent in the HTTP **`Authorization`** header in subsequent requests

JWT VS. SESSION

- Data stored in JWT and session ***cannot be tempered*** with at the client side
- With JWT token, data are kept on *client side* whereas session data are kept on *server side*
  - Should not keep sensitive data in JWT
  - With JWT, the storage requirement on the server side is less
  - Easier to scale with JWT (e.g., easier to share data such as login status across multiple servers)
- Can JWT replaces session for implementing *login/logout*?
  - For best practice with JWT security, see: *https://curity.io/resources/learn/jwt-best-practices/*

TOP 10 OF OWASP

- ***Open Web Application Security Project***
- Top 10 Web Application Security Risks (2021)
    1. Broken access control
    2. Cryptographic failures
    3. Injection
    4. Insecure design
    5. Security misconfiguration
    6. Vulnerable and outdated components
    7. Identification and authentication failures
    8. Software and data integrity failures
    9. Security logging and monitoring failures
    10. Server-side request forgery
- *See more: https://owasp.org/Top10/*

INJECTION

- Untrusted data is sent to an interpreter as part of a command or query
  - *e.g.*, SQL database command formed in a server script, with string concatenated from user input (“SQL injection”)
    - For example, there is such a line in an application

      ```
      String query = "SELECT * FROM accounts WHERE custID='" +
      request.getParameter("id") + "'";
      ```

    - What if the attacker sends such a request?

      ```
      http://example.com/app/accountView?id=' or '1'='1
      ```

- All user input must be ***validated*** and ***sanitized*** before using!
- *See: https://owasp.org/Top10/A03_2021-Injection/*

CROSS-SITE THREATS

- Cross-Site Scripting (XSS)
  - ***Stored XSS***: If a script snippet is inserted into an input box in application, showing the received input to the user ➔ executing the script
  - ***Reflected XSS***: A script snippet is inserted into the query string of a URL as a link to trick users to click (and execute the script)
    - e.g., `http://example.com/search?q=<script>...</script>`
  - ***DOM based XSS***: Instead of using query string, the HTML fragment # is used so that the script is not sent to the server
    - e.g., `http://example.com/search#q=<script>...</script>`
- *See: https://owasp.org/www-community/attacks/xss*

CROSS-SITE THREATS

- Cross-Site Request Forgery (CSRF)
  - If ***request origin*** of an action is not checked, a request could be made on a malicious site other than the expected site, e.g.,
    - *Site A* allows an action with **`GET`** using query strings. An attacker could lure a Site A user to visit *Site B*, where a hidden **`GET`** request is sent to *Site A*, and obtain the user's information.
- Clickjacking: overlaying a transparent `<iframe>` of another site, and trick the users into click on that
  - *See: https://javascript.info/clickjacking*
- ***CORS configuration must be considered carefully!***

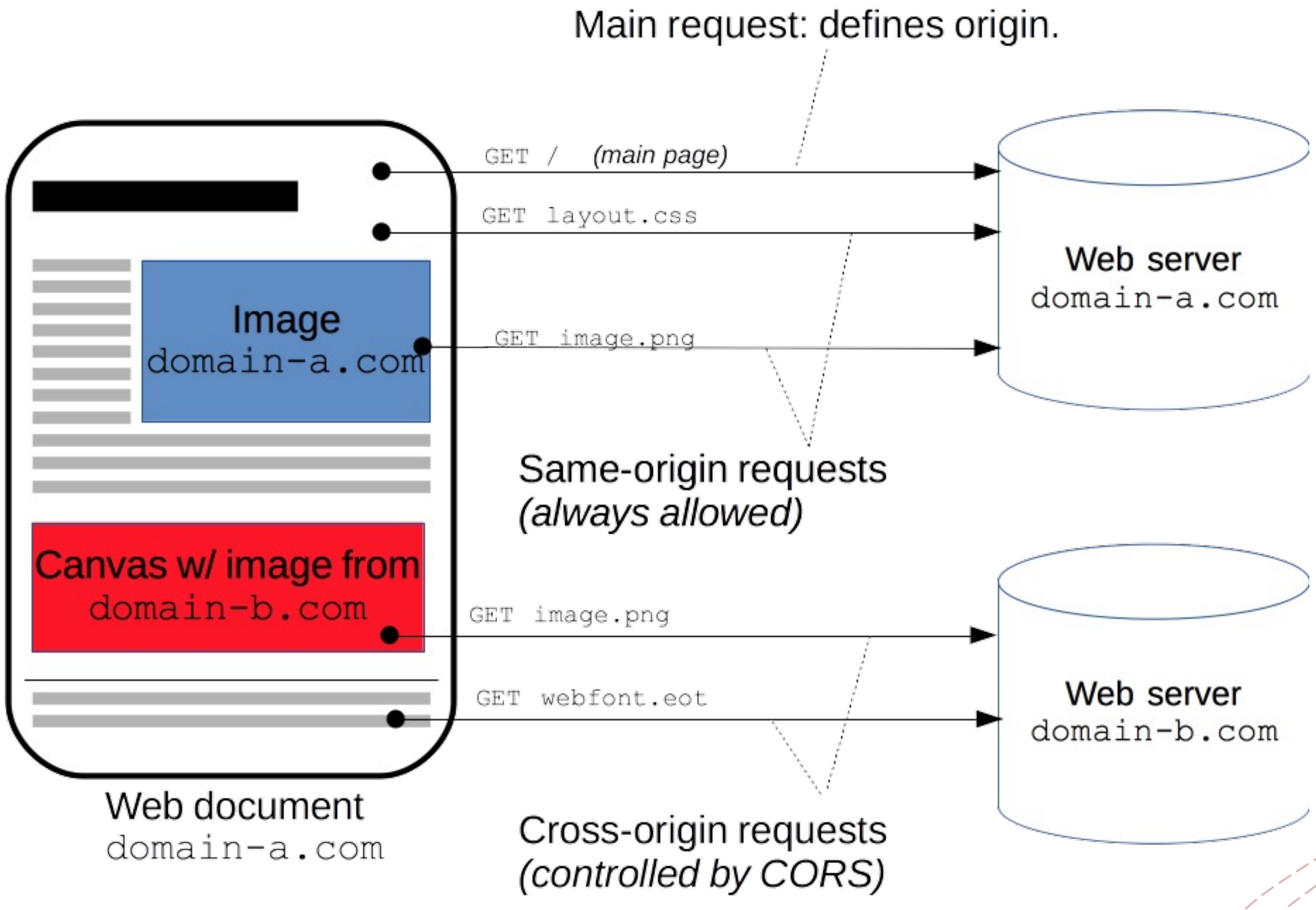
CROSS-ORIGIN RESOURCE SHARING (CORS)

- Same origin ➔ Same protocol, host, and port
  - e.g., Only resources 1) and 2) have same origin

```
1)  http://www.example.com/dir/page1.html
2)  http://www.example.com/dir2/abc.jpg
3)  http://foo.example.com/dir/page2.html   (different host)
4)  http://www.example.com:8000/index.html  (different port)
5)  https://www.example.com/dir/page1.html  (different protocol)
```

- A resource makes a cross-origin HTTP request when it requests a resource from a different origin
  - e.g., An HTML page served from origin A embeds an image from origin B or sends a Fetch request to origin B
- For security reasons, browsers by default restrict cross-origin HTTP requests initiated from within scripts (e.g., Fetch, Ajax)

CROSS-ORIGIN RESOURCE SHARING (CORS)



See: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

CROSS-ORIGIN RESOURCE SHARING (CORS)

- CORS standard: how a client and a server should interact to make CORS request possible ***without compromising security***
- ➔ New HTTP headers that allow servers to inform clients
  - Which origins are permitted to send CORS request
  - What headers are permitted in the request
  - What HTTP request methods are permitted
  - Whether credentials (including Cookies and HTTP Authentication data) should be sent with requests
- ➔ Describing how clients should ***preflight*** certain requests before sending the actual request

READ FURTHER…

OWASP Cheatsheets

https://cheatsheetseries.owasp.org