<WA/>
2025

Authentication

For some, but not for all

Fulvio Corno
Luigi De Russis
Enrico Masala

Outline

- The need for authentication
- HTTP sessions
- Authentication in Express and React
- Authorization and tokens (hints)

https://flaviocopes.com/cookies/

Who are you?

AUTHENTICATION IN WEB APPLICATIONS

Authentication vs. Authorization

Authentication

- Verify you are who you say you are (identity)
- Typically done with credentials
  - e.g., username, password
- Allows a personalized user experience

Authorization

- Decide if you have permission to access a resource
- Granted authorization rights depends on the identity
  - as established during authentication

Often used in conjunction to protect access to a system

https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization

Authentication and Authorization

- Developing authentication and authorization mechanisms
  - is complicated
  - is time-consuming
  - is prone to errors
  - may require interacting with third-party systems (login with Google, Meta, …)
  - …
- Involve both client and server
  - and requires to understand several new concepts
- Better if you rely upon best practices and “standardized” processes

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

The many Layers of Authorization

Who	What	How	When
User	Login / Logout / Navigate pages		
React App	Is the user logged? Remember user information	State/Context variables	Set at login Destroyed at logout Queried during navigation
Browser	Remembers navigation session	Session Cookie (stores session ID)	Received at login, in HTTP Response Re-sent to server at every HTTP Request
Server	Remember session data	Session storage (creates session ID, remembers associated data: username, group, level, …)	Created at login Destroyed at logout Retrieved at every HTTP Request
Route (HTTP API) in the **server**	Check authorization Execute API	Verify session or token validity	At **every** HTTP Request for non-public API
Route (Login) in the **server**	Perform authentication	Check user/pass If ok, create session information	At Login time
Route (Logout) in **server**	Forget authentication	Destroy session information	At Logout request
Database (at Login)	Validates user information	Queries & password hashing	At Login time
Database (HTTP API)	Retrieves user information	Queries from session information	At every HTTP Request

Giving memory to HTTP

COOKIES AND SESSIONS

Sessions

- **HTTP is stateless**
  - each request is independent and must be self-contained
- A web application may need to keep some information between different interactions
- For example:
  - in an on-line shop, we put a book in a shopping cart
  - we do not want our book to disappear when we go to another page to buy something else!
  - we want our "state" to be remembered while we navigate through the website

Sessions

- A **session** is temporary and interactive data interchanged between two or more parties (e.g., devices)
- It involves one or more messages in each direction
- Often, one of the parties keeps the state of the application
- It is established at a certain point it time and ended at some later point

Session ID

- Basic mechanism to maintain session
- Upon authentication, the client receives from the server a session ID
- The session ID allows the server to recognize subsequent HTTP requests as part of an *authenticated session*
- The session ID
  - must be stored on the client side
  - must be sent by the client at every request which is part of the session
  - must not contain sensitive data: it is readable by the client!
  - must be protected from eavesdropping during communications
- Typically stored in and sent as a **cookie**

https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

https://en.wikipedia.org/wiki/HTTP_cookie
https://tools.ietf.org/html/rfc6265

Cookie

- A small portion of information inserted in HTTP headers
- Automatically handled by browsers
  - Automatically stored in the browser cookie storage
  - Automatically **sent** by the browser to servers when performing a request to the same **domain** and **path** that originally sent the cookie (note that port is not included)
    - options are available to send them in other cases
- **NEVER** store sensitive information in a cookie (e.g., password, secrets, etc.): they can be read by the client/browser/user!

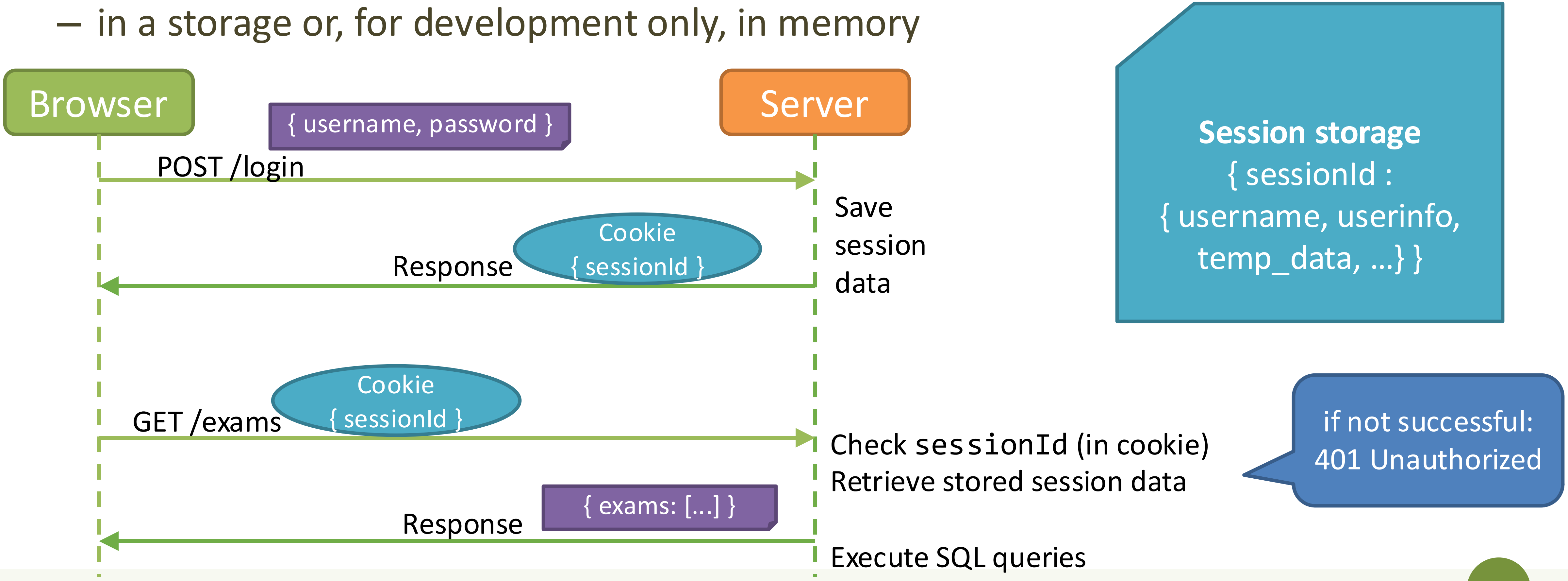
Cookie

- Some relevant attributes, typically set by the server:
  - **name**: the name of the cookie [mandatory]. Example: `SessionID`
  - **value**: the value contained in the cookie [mandatory]. Example: `94$KK763KCQ1!`
  - **secure**: *if set*, the cookie will be sent to the server only if using HTTPS, i.e., protected from eavesdropping
  - **httpOnly**: *if set*, the cookie will be inaccessible to JavaScript code running in the browser
  - **expiration date**

https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#cookies

Session-based Authentication

- The user state is stored on the server
  - in a storage or, for development only, in memory



Browser

Server

{ username, password }

POST /login

Save session data

Response

Cookie { sessionId }

Session storage
{ sessionId :
{ username, userinfo, temp_data, ...} }

GET /exams

Cookie { sessionId }

Check `sessionId` (in cookie)
Retrieve stored session data

if not successful:
401 Unauthorized

Response

{ exams: [...] }

Execute SQL queries

Security Tips

- **Always** use HTTPS and the “secure” option in cookies (at least in production)
  - Always use “httpOnly” option in cookies
- **Never**, never store sensitive information in cookies

NB: **Only for this course, only for configuration simplicity**, HTTPS will NOT be used

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

Security Advice

- Rely on **best practices** and avoid to *re-invent the wheel* for auth, because web applications can be exposed to several attacks, for instance
  - *XSS* (Cross-Site Scripting): attackers inject malicious JS code into web pages
    - **httpOnly** prevents cookie access by any JS code, regardless of the source (Note that, however, also the original JS app code cannot access such cookies)
  - *CSRF* (Cross-Site Request Forgery): a user is tricked by an attacker into submitting a request that they did not intend, with the browser also automatically sending authentication cookies!
- Proper usage of frameworks, best practices, and dedicated libraries help preventing many attacks

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

Authentication and authorization with Passport.js and React

AUTH IN PRACTICE

Basic Login Flow (part 1)

1. A user fills out a form in the client with a unique user identifier (“username”) and a password
2. Data is validated and, if ok, is sent to the server, with a **POST** API
3. The server receives the request and checks whether the user is already registered, and if the password match the stored one
   - Password comparison exploits cryptographic hashes *(more on this later)*
4. If not, it sends back a (generic!) response to the client, not to leak info!
   “Wrong username or password”

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html **#incorrect-and-correct-response-examples**

NO: "Login for User foo: invalid password."
NO: "Login failed, invalid user ID."
NO: "Login failed; account disabled."
NO: "Login failed; this user is not active."
OK: "Login failed; Invalid user ID or password."

Basic Login Flow (part 2)

5. If username and password are correct, the server generates a session ID
6. The server stores the session ID (together with some user info retrieved by the database) in its “server session storage”
7. The server replies to the login HTTP request by creating and sending a cookie
   - **name**: “SessionID”, **value**: the generated session ID, **httpOnly**: true, **secure**: true (secure: true only if it sent over HTTPS)
8. The browser receives the response with the cookie
   - the cookie is automatically stored by the browser
   - the response body is handled by the web application (e.g., to say "Welcome!” etc.)

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

Login Form: Use Standard Practice

- A React component with local state (“controlled” form components)

```
<LoginForm userLogin={userLoginCallback}/>

function LoginForm(props) => {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');

    doLogin = (event) => {
        event.preventDefault();
        if (… form valid …) {
            props.userLoginCallback(username, password);  // Make POST request to authentication server
        } else {
            // show invalid form fields, e.g. empty username or password
        }
    }
...
```

Authentication with Passport

- Suggested authentication middleware, to authenticate users in Express:
  - **Passport**, http://www.passportjs.org
  - install with: `npm install passport`
- Passport is flexible and modular
  - supporting 500+ different authentication strategies
  - for instance, username/password, login with Google, Meta (ex Facebook), etc.
  - able to adapt to different types of databases (SQL and noSQL)
  - adopting some best practices *under-the-hood*
    - e.g., httpOnly cookies for sessions

Passport: Configuration

An Express-based server app needs to be configured in three ways before using Passport for authentication:

1. Choose and set up which authentication strategy to adopt
2. Personalize (and install) additional middleware
3. Decide and configure which user info is linked with a specific session

1. LocalStrategy

- Strategies define how to authenticate users
- `LocalStrategy` supports authentication with username and password
  - install with: `npm i passport-local`
- `function verify (username, password, callback)`
  - <u>Goal</u>: to find the user and verify that it possesses the given credentials
- `callback()` supplies Passport with the user that was correctly authenticated
  - or `false` and an optional `message`

```
const passport = require('passport');
const LocalStrategy = require('passport-local');

passport.use(new LocalStrategy(
function verify (username, password, callback) {
  dao.getUser(username, password).then((user) =>
  {
    if (!user)
      return callback(null, false, { message:
        'Incorrect username and/or password.' });
    return callback(null, user);
  });
}));
```

The `verify` Function in LocalStrategy

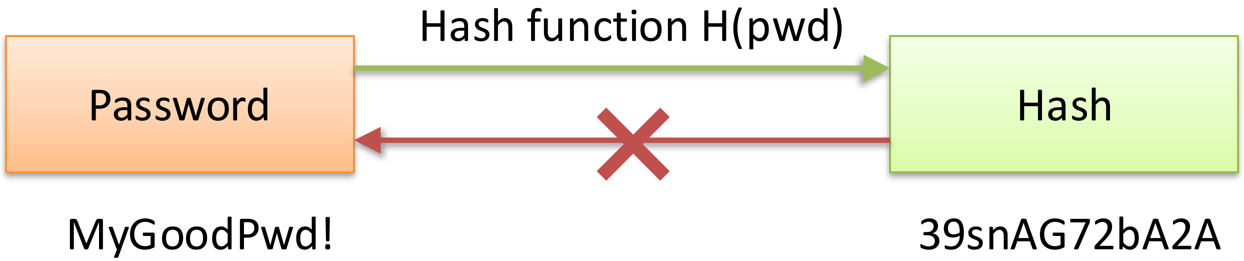
- `username`, `password`: automatically extracted from `req.body.username` and `req.body.password`
- Must check the validity of the credentials
- `callback()`: communicates the result
  - `callback(null,user)`
    → valid credentials
  - `callback(null,false)`
    → invalid credentials, login failed
  - `callback(null,false,{message:'error'})`
    → invalid credentials, login failed, with explanation
  - `callback({error: 'err msg'})`
    → application error (e.g., DB error)
- `user`: *any object* containing information about the currently validated user

```
const passport = require('passport');
const LocalStrategy = require('passport-local');

passport.use(new LocalStrategy( function verify
(username, password, callback) {
  dao.getUser(username, password).then((user) =>
    {
    if (!user)
      return callback(null, false, { message:
        'Incorrect username and/or password.' });
    return callback(null, user);
  });
}));
```

Storing Passwords in the Server

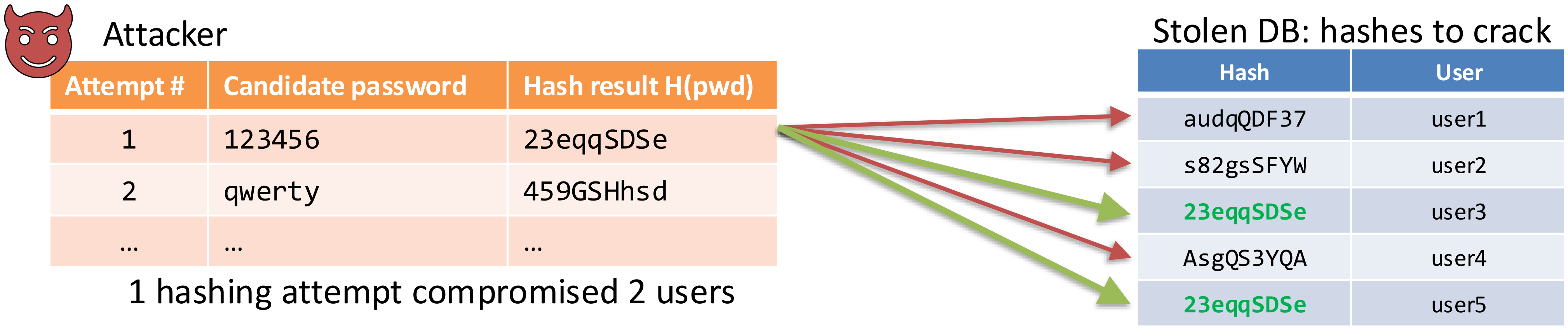
- **Never** store plain text passwords in the server (e.g., in the database)
- **Always** perform password hashing (NOT encryption)
  - so that nobody can read passwords even if it gets access to the database
  - hashing is a one-way function → password cannot be "decrypted"



https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Storing Passwords in the Server

- Good hashing is impossible to crack. But passwords can be guessed…
  - Guess/select possible password candidates (123456, qwerty, password1!, …)
  - Compute the hash (computationally expensive) and compare it with all the available hashes (cheap)
  - When the DB of big sites are compromised (i.e., leaked) many hashes are available: the chance of success drastically increases
    - Also, users with the same hash = same password (typically a weak one…)



Attacker

Attempt #	Candidate password	Hash result H(pwd)
1	123456	23eqqSDSe
2	qwerty	459GSHhsd
…	…	…

1 hashing attempt compromised 2 users

Stolen DB: hashes to crack

Hash	User
audqQDF37	user1
s82gsSFYW	user2
23eqqSDSe	user3
AsgQS3YQA	user4
23eqqSDSe	user5

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

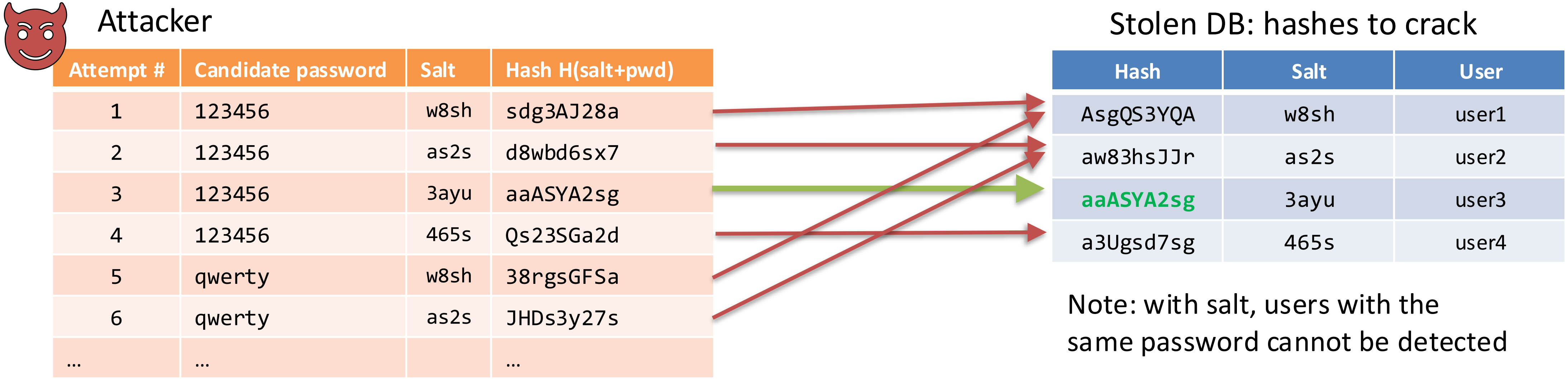
Password Salt

- A unique, randomly generated, string added to each password before the hashing process
- Salt is unique for every user, and stored in cleartext
  - Generated when creating a new username/password record in the DB
  - Force the attacker to run hashing for each password/salt pair
- <u>Always use salt</u> when hashing passwords
  - Most modern hashing algorithms automatically salt the password and store them as a part of the string representing the result
  - Using salt does not increase the complexity of the hashing process

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Salt Effect

- Salt makes cracking large number of hashes significantly more complex
  - The time required increases linearly with the number of hashes to crack
  - Without salt, a single hashing attempt could quickly check millions of hashes



Attacker

Attempt #	Candidate password	Salt	Hash H(salt+pwd)
1	123456	w8sh	sdg3AJ28a
2	123456	as2s	d8wbd6sx7
3	123456	3ayu	aaASYA2sg
4	123456	465s	Qs23SGa2d
5	qwerty	w8sh	38rgsGFSa
6	qwerty	as2s	JHDs3y27s
…	…		…

3 hashing attempts compromised 1 user

Stolen DB: hashes to crack

Hash	Salt	User
AsgQS3YQA	w8sh	user1
aw83hsJJr	as2s	user2
aaASYA2sg	3ayu	user3
a3Ugsd7sg	465s	user4

Note: with salt, users with the same password cannot be detected

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Storing Passwords in the Server

- `scrypt` is an easy to use (reasonably secure) *password hashing* function
  - it generates hashes in the form of a string, e.g., `d72c87d0f077c7766f2985dfab30e8955c373a13a1e93d315203939f542ff86e`
  - test it at https://www.browserling.com/tools/scrypt
- In Node, it is available in the already included `crypto` module
- The salt must be handled/stored separately from the hash

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

scrypt

- Two main functions, both async and returning Promises:

  1. Hash a password:

     ```
     crypto.scrypt(password, salt, keylen,
                   function(err, hashedPassword))
     ```

     `keylen` is the length of the hash to obtain (e.g., 32 or 64).

     The `salt` should be at least 16 bytes long. If needed, it can be generated as:

     ```
     const salt = crypto.randomBytes(16)
     ```

  2. Check if a given password matches with a stored hash:

     ```
     crypto.timingSafeEqual(storedPassword, hashedPassword)
     ```

     The given password <u>must</u> be hashed with the same salt of the stored password

Password Hash Check (within Passport)

```
exports.getUser = (email, password) => {
  return new Promise((resolve, reject) => {
    const sql = 'SELECT * FROM user WHERE email = ?';
    db.get(sql, [email], (err, row) => {
      if (err) { reject(err); }
      else if (row === undefined) { resolve(false); }
      else {
        const user = {id: row.id, username: row.email};

        const salt = row.salt;   // read in cleartext from the database
        crypto.scrypt(password, salt, 32, (err, hashedPassword) => {
          if (err) reject(err);
          if(!crypto.timingSafeEqual(Buffer.from(row.password, 'hex'), hashedPassword))
            resolve(false);
          else resolve(user);
        });
      }
    });
  });
};
```

2. Additional Middlewares

- Useful *additional middleware* to **enable sessions**: `express-session`
  - https://www.npmjs.com/package/express-session
  - install with: `npm install express-session`
- By default, it stores sessions in *memory*
  - which is highly inefficient and NOT recommended for production
- It also supports different session storages, from files to DB
  - Not discussed here

```
const session = require('express-session');

// enable sessions in Express
app.use(session({
  // set up here express-session
  secret: "a secret phrase of your choice",
  resave: false,
  saveUninitialized: false,
}));

// init Passport to use sessions
app.use(passport.authenticate('session'));
```

2. Session Options

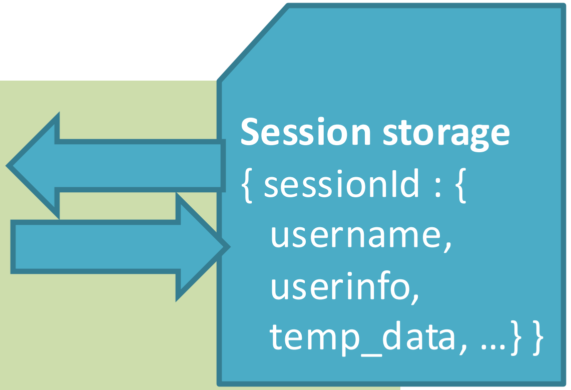
- The `express-session` middleware supports various parameters
- The most used ones are:
  - `secret`: used to sign the session ID cookie [**required**]
  - `store`: the session store instance, defaults to `MemoryStore` if not specified
  - `resave`: forces the session to be saved back to the session store, even if the session was never modified during the request. Default (deprecated) value is true, typically set to ***false***
  - `saveUninitialized`: forces a session that is new but not modified to be saved to the store. Choosing ***false*** is useful for implementing login sessions, reducing server storage usage, or complying with laws that require permission before setting a cookie. Default (deprecated) value is true.

3. Session Content

- After enabling sessions, you should decide the info to store in the session
  - The info is stored internally in req.session.passport (not to be accessed directly)
- The `serializeUser()` and `deserializeUser()` methods allow you to define callbacks to perform these operations
- The `user` object created by `deserializeUser()` will be available in every authenticated request in **`req.user`**

```
passport.serializeUser((user, cb) => {
  cb(null, {id: user.id, email: user.username, firstname: user.firstname});
});

passport.deserializeUser((user, cb) => {
  cb(null, user);
});
```



Session storage
{ sessionId : {
username,
userinfo,
temp_data, …} }

Minimal Content in a Session

- Just serialize a unique user ID (e.g., stored in user.id)
- Retrieve other info from DB during deserialization, if needed
  - Higher load on the DB: queried at every request to a protected API, but the advantage is that information is always updated (e.g., first name, last name, etc.)
- Later, use **req.user** in the API implementation to access info about user

```
passport.serializeUser((user, callback) => {
  callback(null, user.id);
});

passport.deserializeUser((id, callback) => {
  db.getUser(id)
      .then(user => callback(null, user))
      .catch(err => callback(err, null));
});
```

Login with Passport

- After setting everything up, log in a user with Passport
  - add an Express route able to receive the “login” requests
  - pass the `authenticate(<strategy>)` method as the first additional callback
    - `authenticate('local')` will look for `username` and `password` fields in `req.body`

```
app.post('/api/session', passport.authenticate('local'), (req,res) => {

  // This function is called if authentication is successful.
  // req.user contains the session info.

  res.json(req.user.username);  // Just an example

});
```

Storing User Information in React

- With the login response, some user information might be sent back to the browser
  - e.g., the username, the first name, the last name, etc.
- Typically, this information is stored for later usage in the client app
  - could be stored in a React State (or even a Context) and made available to the whole app
  - In any case it can be asked by the app whenever needed (e.g., with `API.getUserInfo()` in a `useEffect` at component mount time, …)
- More suggestions:
  - https://www.robinwieruch.de/react-router-authentication/

After the Login…

- Some routes in the server needs to be **protected**
  - i.e., they must provide a response ***only*** for authenticated users
- After the workflow shown before (session-based auth) is followed, the browser automatically sends the HTTP cookie header to any API belonging to the same domain and path
  - ***Beware***: cookies cannot be sent to other domains and paths

With CORS Enabled

- By default, cookies can only be sent to the same origin
  - CORS has a mechanism to overcome this limitation
- In the server, we need to define *both* the credentials and the origin options, when setting up the `cors` module:

```
const corsOptions = {
  origin: 'http://localhost:5173',
  credentials: true,
};
app.use(cors(corsOptions));
```

Note: if “credentials” is set, CORS does not allow a generic origin *

With CORS Enabled

- In the client, all the `fetch` requests to protected APIs must include the “*credentials: include*” option:

```
const response = await fetch(SERVER_BASEURL + '/api/exams', {
  …
  credentials: 'include',
});
```

- The login request **must** include such an option as well
  - even if it is not to a protected API
  - otherwise the cookie will not be available in subsequent requests even if sent with the ‘include’ option

Protecting Server APIs

- To *protect* server APIs, check if a request comes from an authenticated user by checking Passport's **`req.isAuthenticated()`**
  - returns true if the session id coming with the request is a valid one
- To be done at the beginning of **every** callback body in each API that needs protection
  - Same code to be inserted in different APIs → Create a custom middleware!

Protecting APIs with a Middleware

- *Create* an Express middleware that includes `req.isAuthenticated()`
- Use it at the route level (or even app level to protect everything if needed)
  - The middleware is especially useful to handle errors

```
const isLoggedIn = (req, res, next) => {
  if(req.isAuthenticated())
    return next();

  return res.status(400).json({message : "not authenticated"});
}

app.get('/api/exams', isLoggedIn, (req, res) => {
  ...
});
```

Authorization Check

- The API implementation must check if the user can do the operation
  - It checks the authorization (NB: not authentication).
    Only the application logic can determine which is the correct check to perform.
    It may apply to any operation (even just read, or create, update, delete)
- The server must retrieve authenticated **`user`** info in <u>a secure manner</u>
  - It cannot arrive from the client, <u>**even if**</u> it is in an authenticated session!
  - The code must always extract it from the session information which is safe since it never leaves the server (i.e., **req.user** content)
- Also, rely <u>only on trusted information</u>, e.g. info from DB

Major source of errors at exam, **big vulnerability if wrong!**

Example of Authorization Check

```
DELETE /reservations
HTTP request body: { resId: 1234 }
```

```
app.delete('/reservations', isLoggedIn, (req, res) => {
  if (reservationOwnerFromDb(req.body.resId) == req.user.id) {
    delete(…);
    res.end();
  }
});
```

Major source of errors at exam, **big vulnerability if wrong!**

- Any solution where the id of the user does not come from req.user is **WRONG**!
  - In particular, the user id cannot be part of an object coming from the client, even if the session exists and it is valid (i.e., `isLoggedIn() == true`)
  - Otherwise, any authenticated user could send the request and pass the isLoggedIn() check!

Logout

- The browser will send a "logout" request to the server
  - For example: DELETE /api/session
- The server will clear the session (and delete the stored session id)
  - extremely trivial with Passport!

```
app.delete('/api/session', (req, res) => {
  req.logout(() => {
    res.end();
  });
});
```

https://auth0.com/docs/secure/tokens

What are you allowed to you?

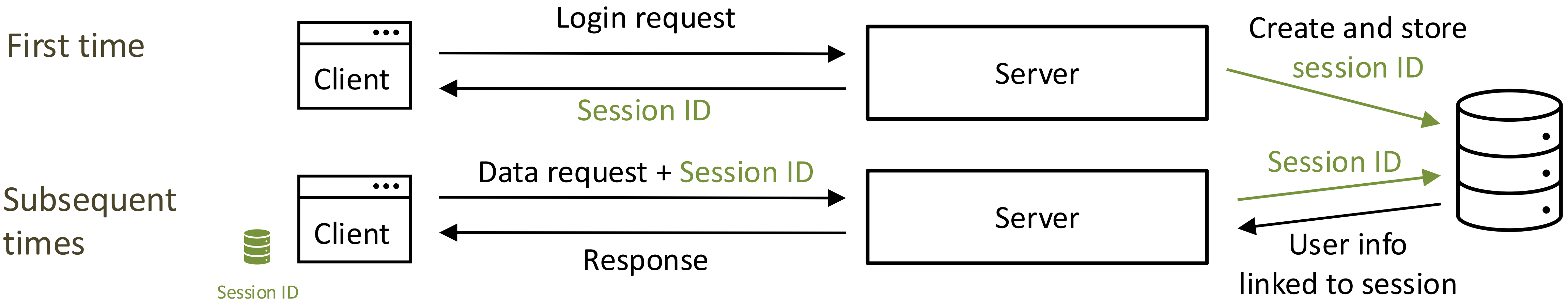
AUTHORIZATION IN WEB APPLICATIONS

Authorization after Authentication

- In general, there exists two approaches for web servers to handle authorization after authentication:
  - Stateful server
  - Stateless server

Stateful Server (described until now)

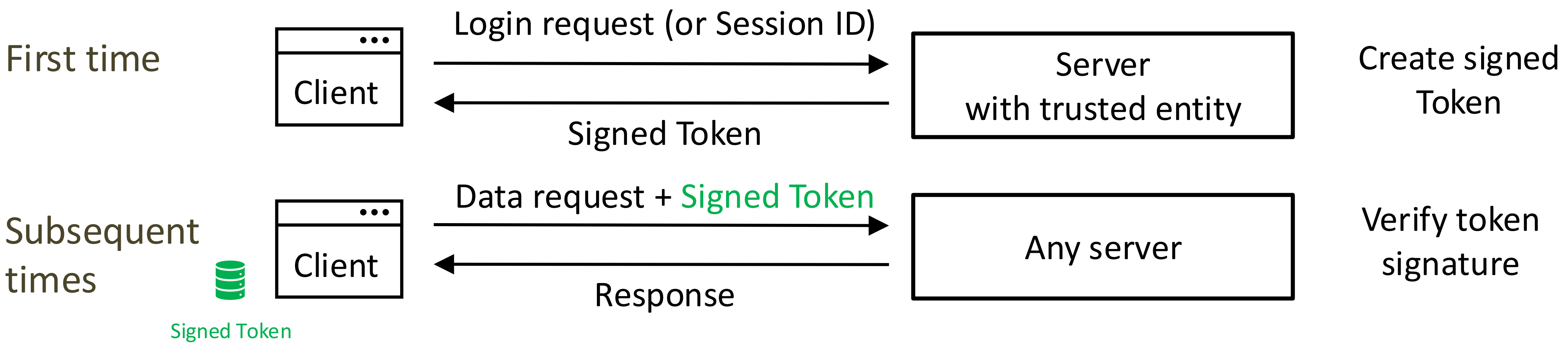
- The server **remembers** the valid session IDs and the associated user info (after login)
- Associated info cannot be maliciously altered: the trusted version is only in the **server**
- Each time a request arrives for a restricted resource, the server **retrieves** the info associated with the session and **decides if the associated user is authorized or not**
- Works best with a single server that manages everything



https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

Stateless Server

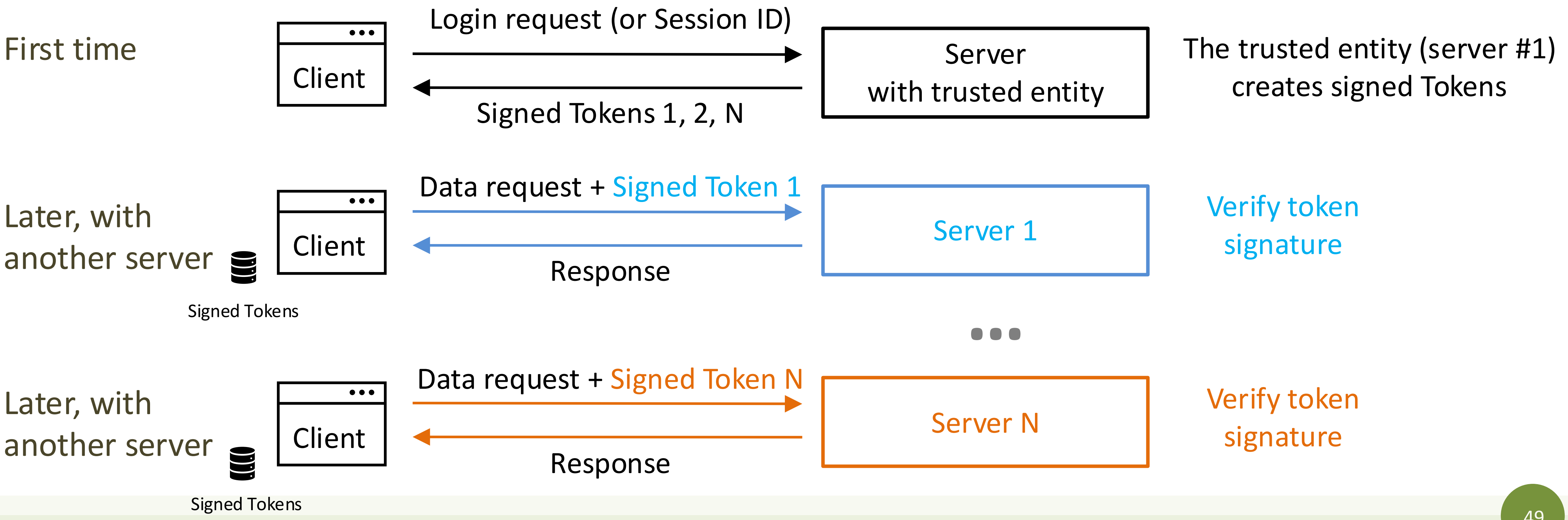
- A trusted entity (e.g., the authentication server) **signs a payload** which contains information about what can be accessed and how, the user info (if needed), and when the authorization expires
- The client gets the signed payload (**token**) and **stores** it
- Each time a request arrives for a restricted resource, the receiving **server verifies the signature**, extracts and uses the authorization information
  - such information is **trusted because it is signed**



https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

Stateless Server Scheme

- Works best where there are multiple servers which cannot easily share session information (but can easily verify a signature)
  - note: tokens can also be the same for all servers



First time

Client

Login request (or Session ID)

Signed Tokens 1, 2, N

Server with trusted entity

The trusted entity (server #1) creates signed Tokens

Later, with another server

Signed Tokens

Client

Data request + Signed Token 1

Response

Server 1

Verify token signature

...

Later, with another server

Signed Tokens

Client

Data request + Signed Token N

Response

Server N

Verify token signature

Note on Tokens and Stateless Servers

- Standard token formats exists, e.g., JWT (JSON Web Token), RFC 7519
- Many schemes exists to implement complex authorization flows, for different purposes (access permission, Single Sign On, etc.)
- Examples:
  - OAuth
  - SAML
  - OpenID

Out of the scope of this course

Tokens / authentication / authorization

- Note that tokens can also be used for identification purposes (i.e., after authentication, instead of a session ID)
  - NOT covered in this course
  - NOT to be used in this course (just for simplicity)
- In this course
  - Perform initial authentication, establish a session ID, then:
  - **Every** authorization must be checked on the server, in every API, through JS code, relying only on trusted information, i.e., the logged in user + info from DB

Important! Carefully checked at the exam

```
app.delete('/reservations', isLoggedIn(), (req, res) => {
  if (reservationOwnerFromDb(req.body.resId) == req.user.id) {
```

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/