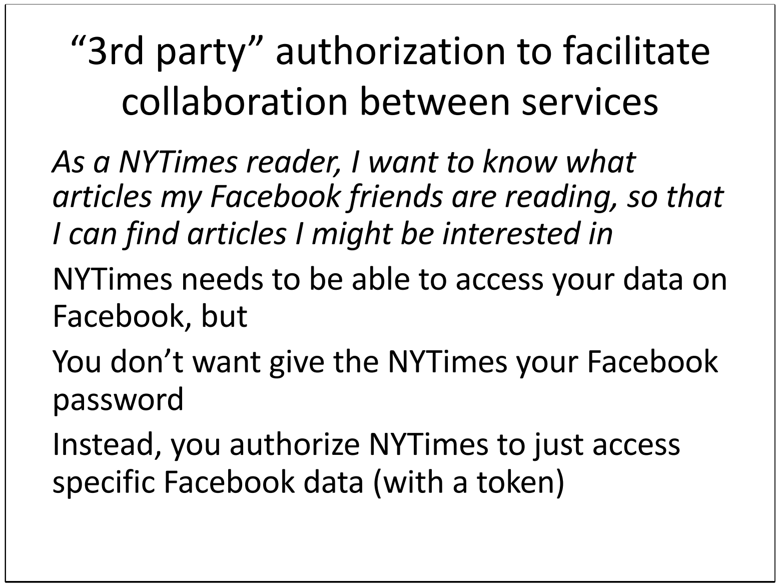


“3rd party” authorization to facilitate collaboration between services

As a NYTimes reader, I want to know what articles my Facebook friends are reading, so that I can find articles I might be interested in

NYTimes needs to be able to access your data on Facebook, but

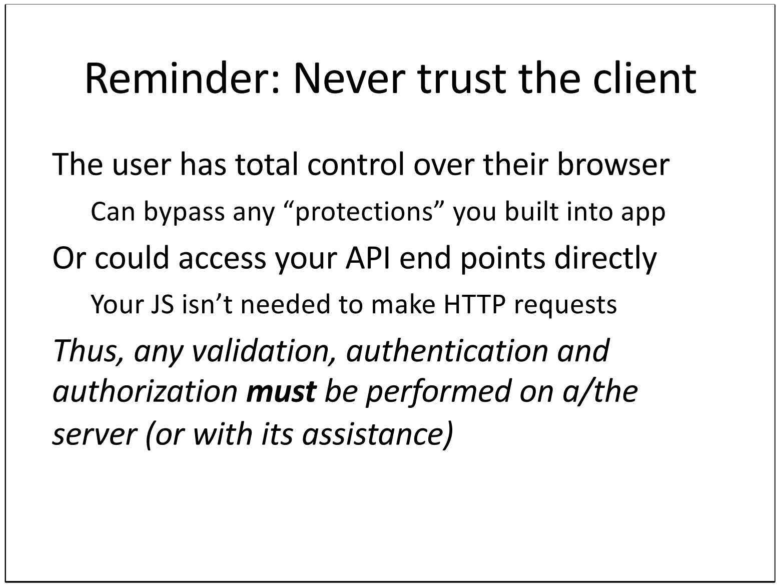
You don’t want give the NYTimes your Facebook password

Instead, you authorize NYTimes to just access specific Facebook data (with a token)

In a web 1.0 world few sites had an API, so implementing this functionality was very difficult. Often you would need to effectively “log in” as the user (which required the user to share their password). This precluded collaboration between services. Increasingly services like Facebook and others provide APIs that facilitate this integration. In this model, which I suspect you have used extensively, when an application wants to access data held by a 3rd party, e.g., Facebook, it redirects the user to that 3rd party to login directly (authenticate) and authorize the application (e.g., NYTimes) to access the user’s data. If the user agrees, Facebook, or other 3rd party, sends back a token that the application (NYTimes) can use to retrieve just the data its authorized to access.

Note the distinction between authentication and authorization, the user authenticates to their identity and separately authorizes the third part access to their data..

Single sign-on is a variant of this approach (without the data sharing, or only minimal data sharing), where we rely on authentication provided by a third party, e.g., Microsoft or Google, to verify the person is the same person who initially setup the account… Unless there is a compelling reason for an alternate approach, any projects that need authentication should use a 3rd party (specifically Google). This is an aspect where we want to avoid DIY if we can… it is just too easy to get it wrong.



Reminder: Never trust the client

The user has total control over their browser

- Can bypass any “protections” you built into app

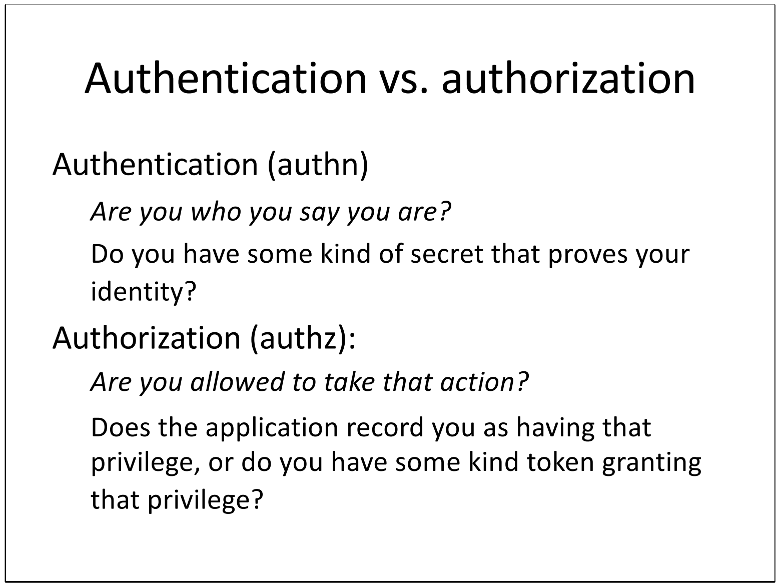
Or could access your API end points directly

- Your JS isn’t needed to make HTTP requests

*Thus, any validation, authentication and authorization **must** be performed on a/the server (or with its assistance)*

For example, in Simplepedia we disabled the Save button, but that is just an HTML attribute. The user could easily ”re-enable” that button in their browser.

When we have tested our servers, we used the console to make requests directly to the API. Our front-end application was not involved. Someone with malicious intent could similarly make requests to your server without involving your front-end application in any way. So, you can’t rely on any “protections” built into that application.



Authentication vs. authorization

Authentication (authn)

Are you who you say you are?

Do you have some kind of secret that proves your identity?

Authorization (authz):

Are you allowed to take that action?

Does the application record you as having that privilege, or do you have some kind token granting that privilege?

We want to distinguish between authentication, proving your identity, and authorization, proving that you are allowed to perform some action. This is a familiar notion to us, that different users may have different roles, or privileges, and that just because I have an account on a service, just because I am authenticated, doesn't mean I can access all data, routes, etc.

In many cases authentication is a prerequisite to authorization, that is first I login, then I can access certain data/features. However, in widely used 3rd party workflows, i.e. where an application is accessing data held by a third party, that application uses a cryptographic token, issued by the 3rd party, to prove the user has authorized it to access that data, etc.

In our practical we will implement both aspects, i.e. authn and authz, albeit with simple authorization – some actions are restricted to logged in users (but we don't distinguish between different users). In many of your applications you will need more sophisticated approaches to protect sensitive data.

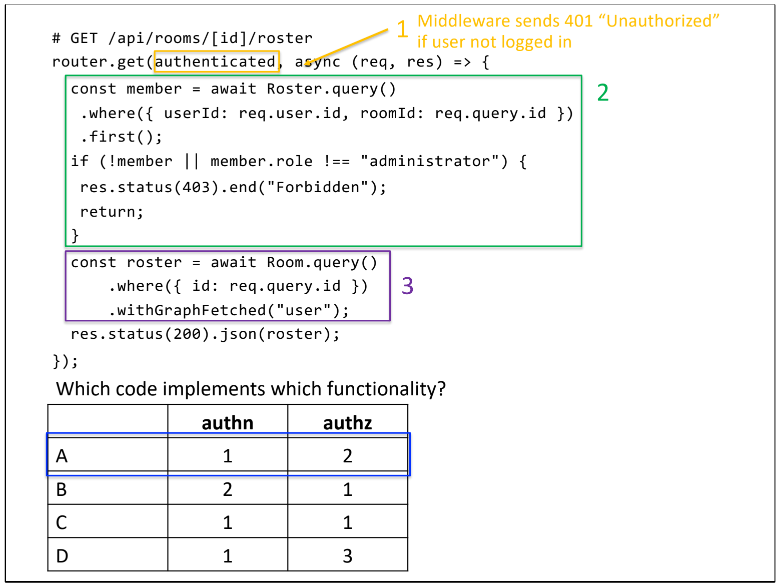
Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

How would you best implement the following item in the Class Interactor's backlog, “Users should only be able to list rooms they are administrators of”? Assume you have required user to authenticate to view the relevant React component.

A. Filter the list of rooms in the React component with current user as an administrator
B. Require authentication in '/api/rooms' route
C. Answers A and B
D. Answer B and filter rooms database query by administrator status
E. Answers A and D

Answer: D

Answers without D, potentially send data to the client that they are not authorized to see. Recall we can't trust the client, filtering data out in client, doesn’t prevent them from viewing it in the network tab for example. If are we effectively enforcing authorization on the server, then A (and thus E) is not necessary and is just unneeded code/work.



1 Middleware sends 401 “Unauthorized” if user not logged in

```
# GET /api/rooms/[id]/roster
router.get(authenticated, async (req, res) => {
  const member = await Roster.query()
   .where({ userId: req.user.id, roomId: req.query.id })
   .first();
  if (!member || member.role !== "administrator") {
   res.status(403).end("Forbidden");
   return;
  }
  const roster = await Room.query()
      .where({ id: req.query.id })
      .withGraphFetched("user");
  res.status(200).json(roster);
});
```

2

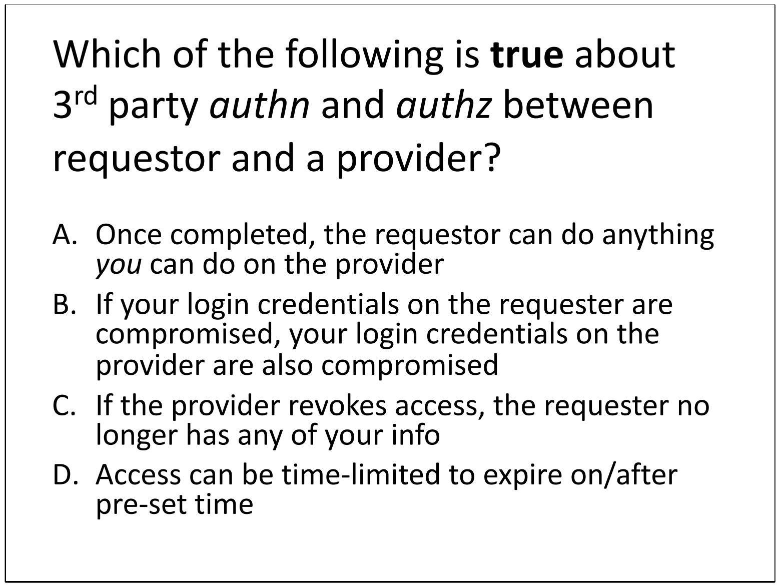
3

Which code implements which functionality?

	authn	**authz**
A	1	2
B	2	1
C	1	1
D	1	3

Answer: A

As its name suggests, the authenticated middleware implements authn, but only authn. Block 2 is required for authz, that is restricting access to this endpoint to administrator. Block 3 doesn’t perform any authentication or authorization.



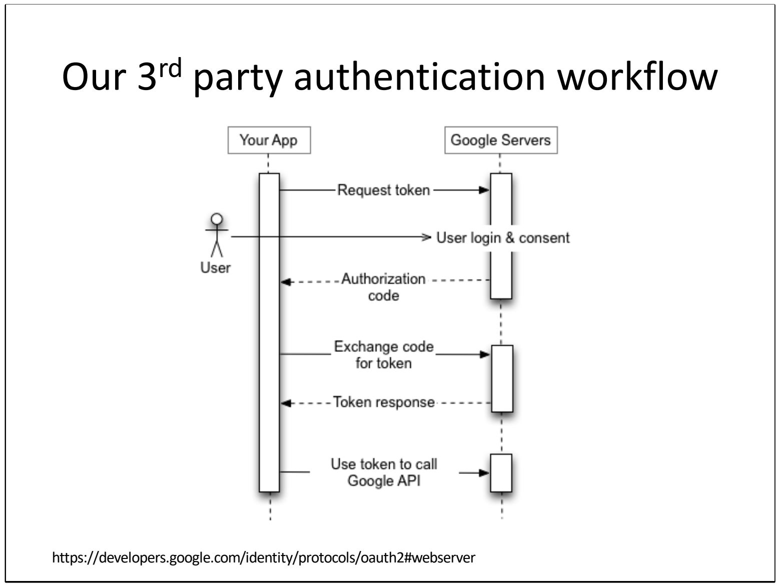
Which of the following is **true** about 3rd party *authn* and *authz* between requestor and a provider?

A. Once completed, the requestor can do anything *you* can do on the provider
B. If your login credentials on the requester are compromised, your login credentials on the provider are also compromised
C. If the provider revokes access, the requester no longer has any of your info
D. Access can be time-limited to expire on/after pre-set time

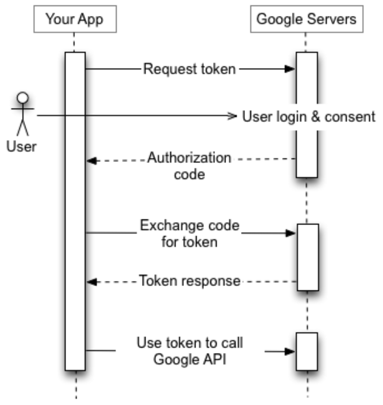
Answer: D

We can approach this via elimination, that is A-C are all false. The requestor is allowed to do those operations you have authorized, not everything you can do. The requestor doesn't have your login credentials (just a token) and so a breach at the requestor doesn't compromise your login credentials at the provider. And while you can revoke future access, you can't "revoke" access to data the requestor has already obtained. We can however can typically set access to only last for a certain amount of time before it expires and needs to be renewed (or is terminated).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

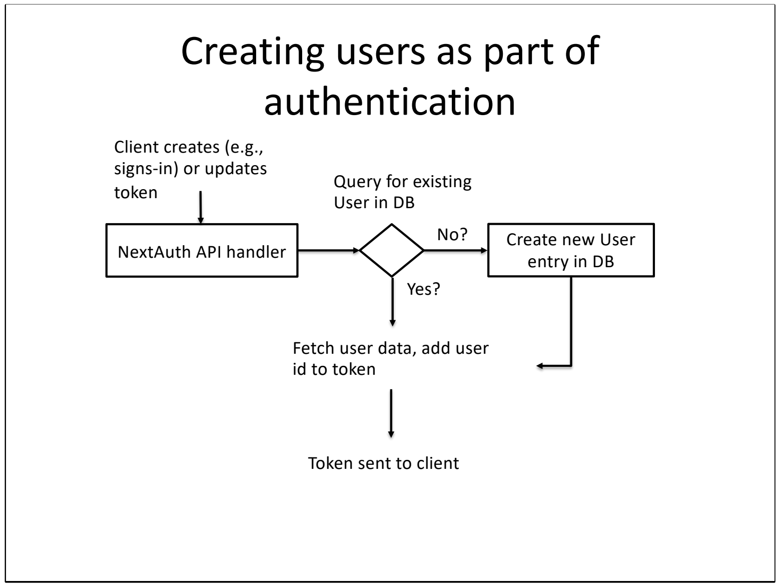


Our 3rd party authentication workflow



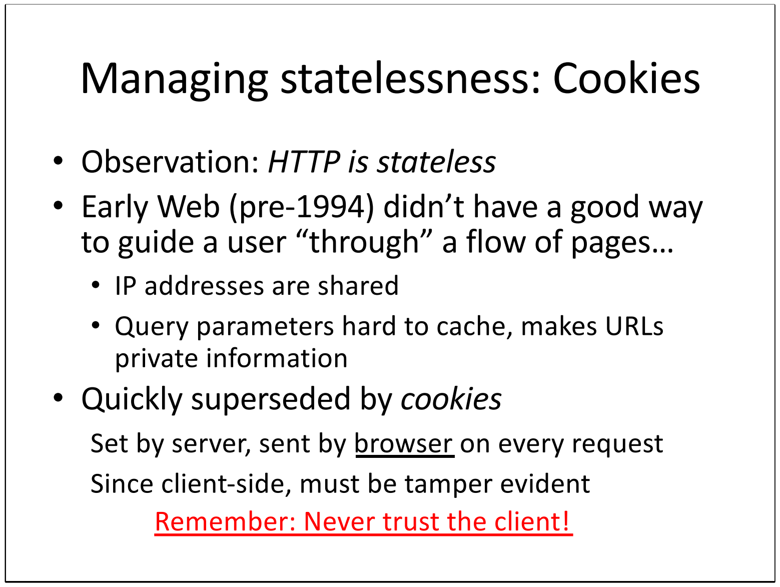
https://developers.google.com/identity/protocols/oauth2#webserver

That is a lot going on! Fortunately, most of this is transparent to us (it is handled by NextAuth). At the end of this process, NextAuth is creating an encrypted token that is stored on client and sent with each request. We use that token to verify users.



Creating users as part of authentication

Client creates (e.g., signs-in) or updates token
NextAuth API handler
Query for existing User in DB
No?
Create new User entry in DB
Yes?
Fetch user data, add user id to token
Token sent to client



Managing statelessness: Cookies

- Observation: *HTTP is stateless*
- Early Web (pre-1994) didn't have a good way to guide a user "through" a flow of pages…
  - IP addresses are shared
  - Query parameters hard to cache, makes URLs private information
- Quickly superseded by *cookies*

  Set by server, sent by browser on every request

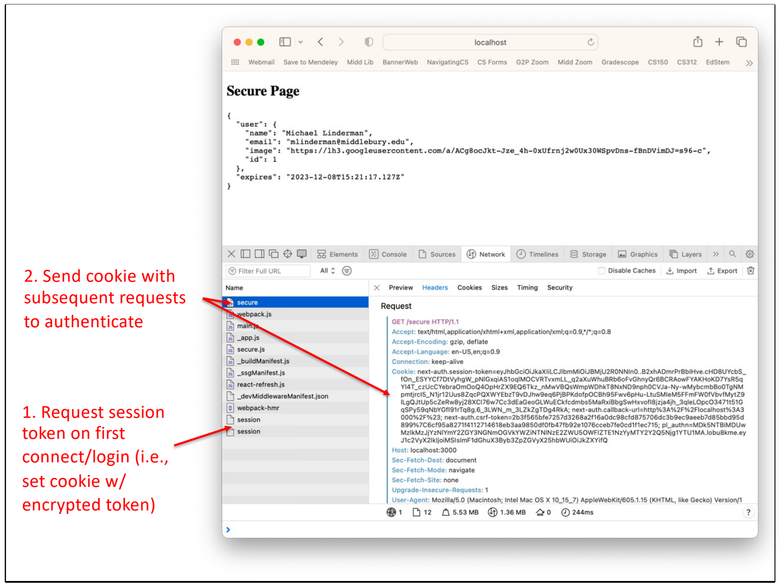
  Since client-side, must be tamper evident

  Remember: Never trust the client!

What is the value of statelessness? Treats requests independently. No need to maintain client's previous interactions, and thus different servers can handle different requests.

We use similar cryptographic approaches to what we described previously for ensuring that cookies are tamper evident.

Note that we switch to using cookies because we are authenticating with our own server. Cookies are only sent to the server that set them, so when interacting with 3rd party APIs we will continue to use tokens. We could also use tokens with our own API (may be needed if not clients are not a browser), but I think we will find cookies easier, since they are transparently handled by the browser and automatically sent by fetch (in more recent versions of the specification). While tokens must be explicitly sent with each request.



Secure Page

```
{
  "user": {
    "name": "Michael Linderman",
    "email": "mlinderman@middlebury.edu",
    "image": "https://lh3.googleusercontent.com/a/ACg8ocJkt-Jze_4h-0xUfrnj2w0Ux30WSpvDns-fBnDVimDJ=s96-c",
    "id": 1
  },
  "expires": "2023-12-08T15:21:17.127Z"
}
```

Name
- secure
- webpack.js
- main.js
- _app.js
- secure.js
- _buildManifest.js
- _ssgManifest.js
- react-refresh.js
- _devMiddlewareManifest.json
- webpack-hmr
- session
- session

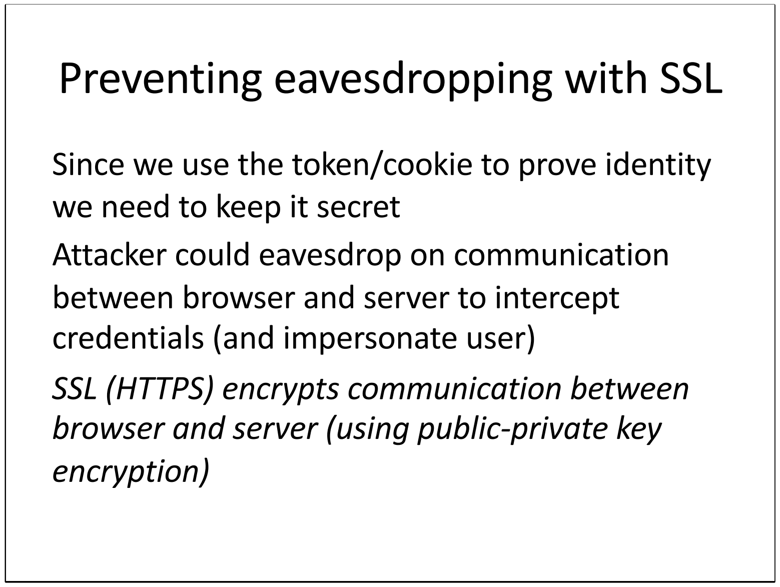
Request

```
GET /secure HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
Cookie: next-auth.session-token=eyJhbGciOiJkaXIiLCJlbmMiOiJBMjU2R0NNIn0..B2xhADmrPrBbIHve.cHD8UYcbS_
    fOn_ESYYCf7DtVyhgW_pNIGxqiAS1oqIMOCVRTvxmLL_q2aXuWhuBRb6oFvGhnyQr6BCRAowFYAKHoKD7YsR5q
    YI4T_czUcCYebraOmOoQ4OpHrZX9EQ6Tkz_nMwVBQsWmpWDhkT8NxND9nph0CVJa-Ny-wMybcmbBo0TgNM
    pmtjrcl5_N1jr12Uus8ZqcPQXWYEbzT9vDJhw9eq6PjBPKdofpOCBh95Fwv6pHu-LtuSMIeM5FFmFW0fVbvfMytZ9
    lLgQJtUp5cZeRw8yj28XCl76w7Cc3dEaGeoGLWuECkfcdmbs5MaRxiBbgSwHxvofI8jzja4jh_3qleLOpcO3471t51G
    qSPy59qNbYGfl91rTq8g.6_3LWN_m_3LZkZgTDg4RkA; next-auth.callback-url=http%3A%2F%2Flocalhost%3A3
    000%2F%23; next-auth.csrf-token=2b3f565bfe7257d3268a2f16a0dc98cfd875706dc3b9ec9aeeb7d85bbd95d
    899%7C6cf95a8271f4112714618eb3aa9850df0fb47fb92e1076cceb7fe0cd1f1ec715; pl_authn=MDk5NTBiMDUw
    MzlkMzJjYzNlYmY2ZGY3NGNmOGVkYWZiNTNlNzE2ZWU5OWFlZTE1NzYyMTY2Y2Q5Njg1YTU1MA.lobu8kme.ey
    J1c2VyX2lkIjoiMSIsImF1dGhuX3Byb3ZpZGVyX25hbWUiOiJkZXYifQ
Host: localhost:3000
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/1
```

2. Send cookie with subsequent requests to authenticate

1. Request session token on first connect/login (i.e., set cookie w/ encrypted token)

When we first login, or connect again (request a new page), we also request an updated session token. Doing so sets a cookie (in our browser) with the encrypted token that is sent with every subsequent request. That is when I request the ”secure” page from the server, or make a fetch, that cookies is being sent with my request to prove my identity.



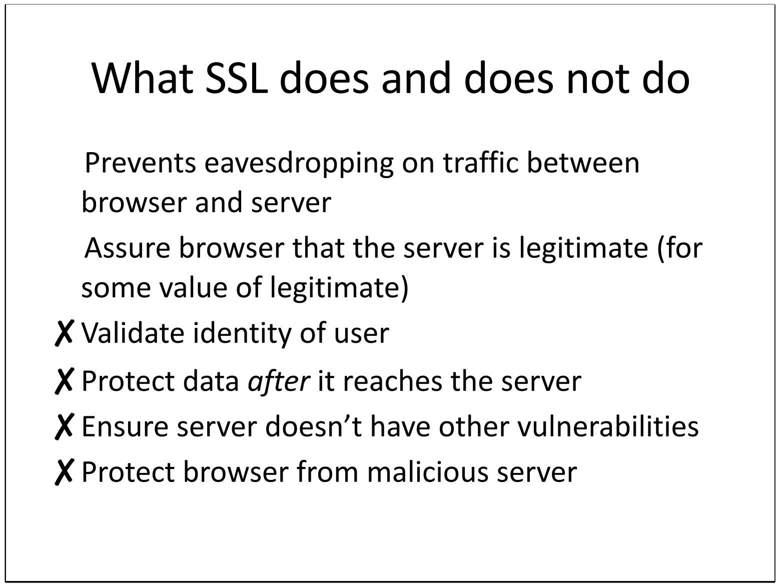
Preventing eavesdropping with SSL

Since we use the token/cookie to prove identity we need to keep it secret

Attacker could eavesdrop on communication between browser and server to intercept credentials (and impersonate user)

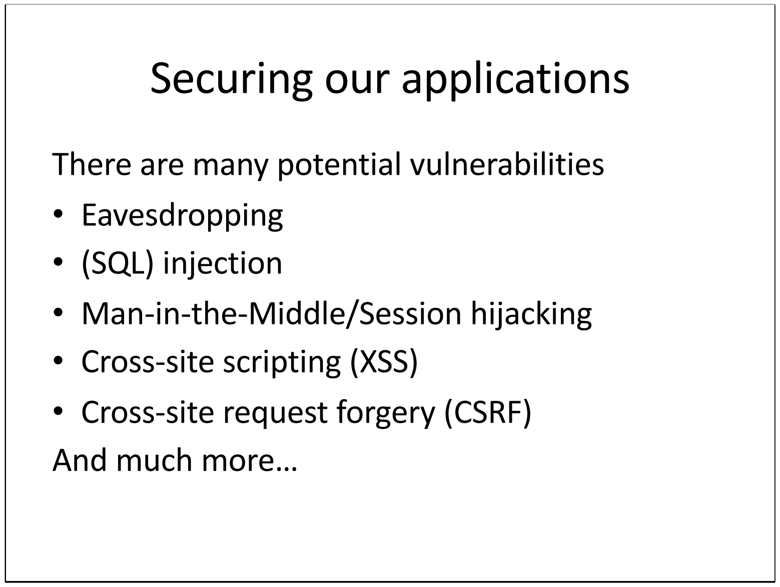
SSL (HTTPS) encrypts communication between browser and server (using public-private key encryption)

Recall that with public-private key encryption, the public key can decrypt messages communicated with private and vice versa. I give out the public key widely so that partners can decrypt my messages (and know those messages are from me) and encrypt messages to me (which only I can read). The browser and server use key exchange methods to bootstrap this encrypted channel after verifying that the server's certificate was signed by a trusted certificate authority. That indicates that the server is who it says it is…



What SSL does and does not do

- Prevents eavesdropping on traffic between browser and server
- Assure browser that the server is legitimate (for some value of legitimate)
- ✗ Validate identity of user
- ✗ Protect data *after* it reaches the server
- ✗ Ensure server doesn’t have other vulnerabilities
- ✗ Protect browser from malicious server



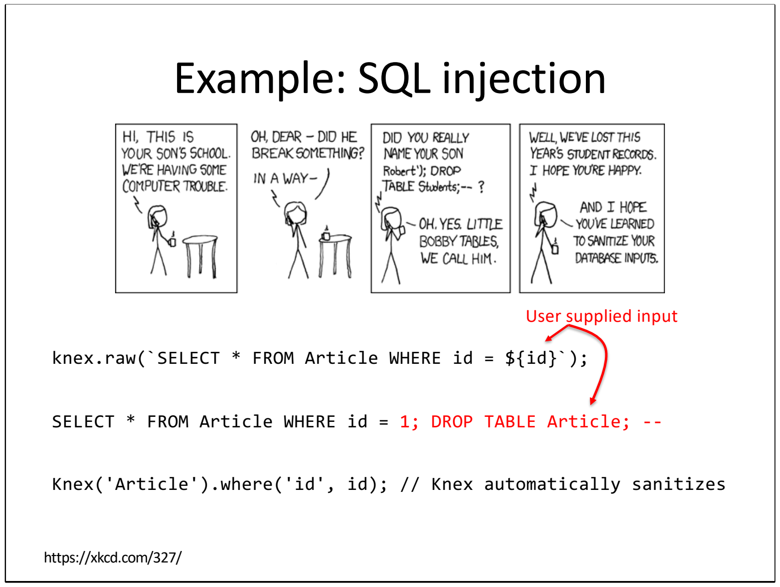
Securing our applications

There are many potential vulnerabilities

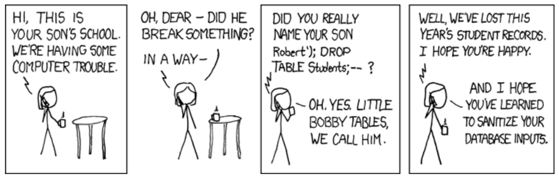
- Eavesdropping
- (SQL) injection
- Man-in-the-Middle/Session hijacking
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)

And much more…

Some of these we have talked about, e.g. eavesdropping and how we could use SSL to mitigate that risk. But many more we won’t discuss or only touch on briefly. For example…



Example: SQL injection


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

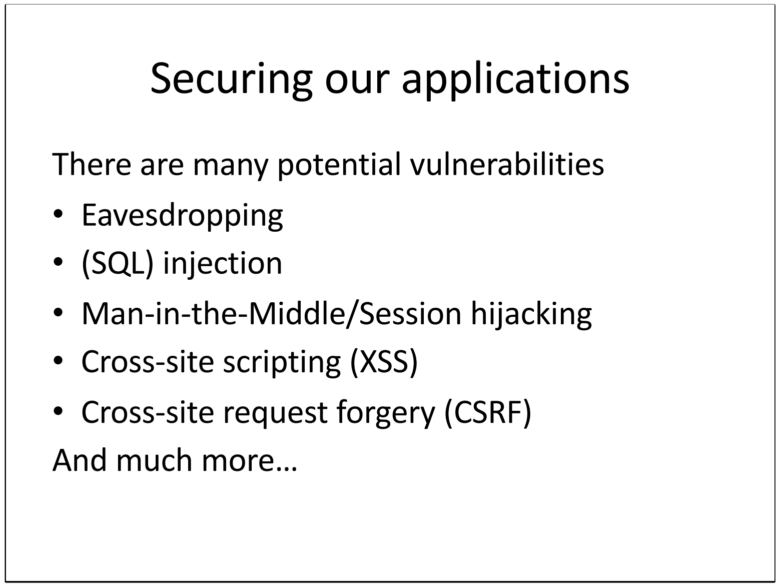
User supplied input

```
knex.raw(`SELECT * FROM Article WHERE id = ${id}`);

SELECT * FROM Article WHERE id = 1; DROP TABLE Article; --

Knex('Article').where('id', id); // Knex automatically sanitizes
```

https://xkcd.com/327/

Here is an example of where want to take advantage of the features of our tools/frameworks for mitigating potential vulnerabilities. This feature isn't unique to Knex, all frameworks/languages will implement safe substitution in some way. And we want to make sure to use that feature.

Behind the scenes Knex is parameterizing the query, e.g., "SELECT * FROM 'Article' WHERE id = $1" and passing the user supplied value separately to be inserted by the database engine. These are treated exclusively as values and so can't be executed directly, i.e., in this example we would get a type error or try to explicitly match "1; DROP
TABLE Article; --".



Securing our applications

There are many potential vulnerabilities

- Eavesdropping
- (SQL) injection
- Man-in-the-Middle/Session hijacking
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)

And much more…

We have only scratched the surface of potential vulnerabilities. It is important for us to review the security recommendations for our chosen frameworks and make sure we are following (and staying up-to-date) with best practices. One of the advantages of more comprehensive frameworks is that they often incorporate these best practices into the implementation by default. For example, you will see CSRF protection built into NextAuth. That is not to say that using a framework guarantees we aren't at risk, but a widely used, well tested framework is likely more robust than anything we would build ourselves if we weren't expects in that domain…

https://owasp.org/www-project-top-ten/