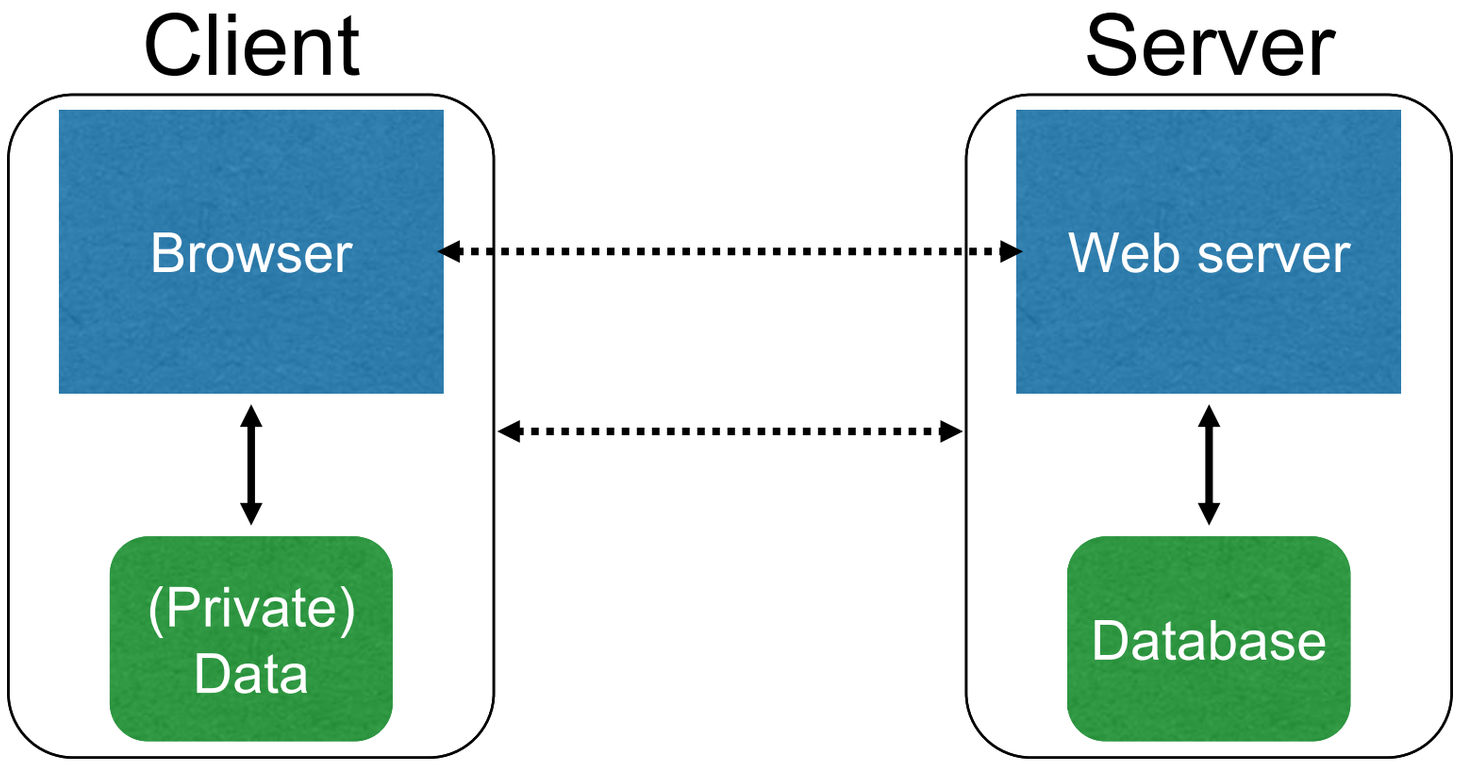
Web security: SQL

Slides from

- Michelle Mazurek 414-fall2016
  - includes stuff from Dave Levin, Mike Hicks, Lujo Bauer
- Dave Levin 414-spring2016

Web Basics

The web, basically



(Much) user data is part of the browser

DB is a separate entity, logically (and often physically)

Interacting with web servers

Resources* which are identified by a *URL

(Universal Resource Locator)

`http://www.umiacs.umd.edu/~mmazurek/index.html`

Protocol

`ftp`
`https`
`tor`

Hostname/server

Translated to an IP address by DNS
(eg, `128.8.127.3`)

Path to a resource

Here, the file `index.html` is static content
i.e., a fixed file returned by the server

Interacting with web servers

Resources* which are identified by a *URL

(Universal Resource Locator)

Path to a resource

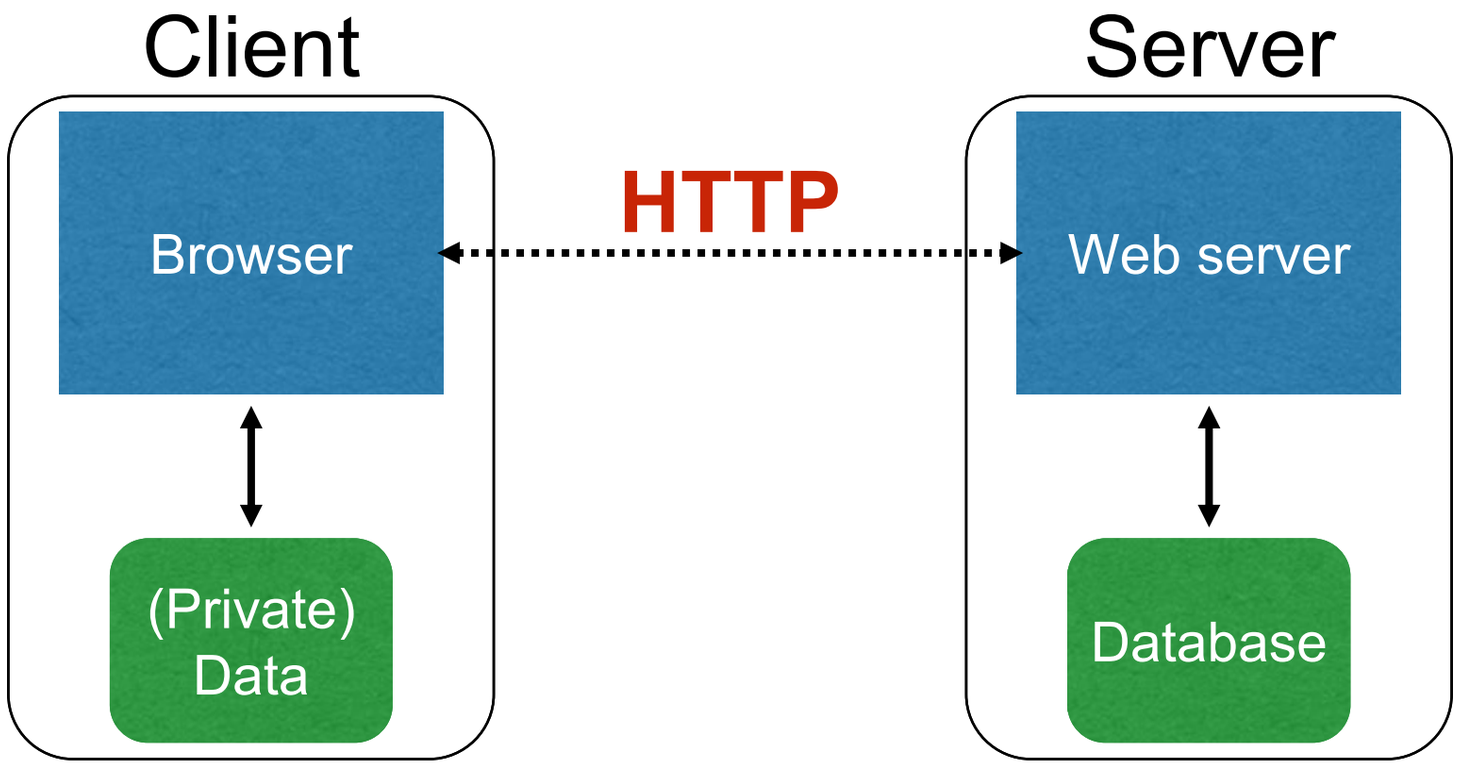
`http://facebook.com/delete.php?f=joe123&w=16`

Arguments

Here, the file **`delete.php`** is dynamic content

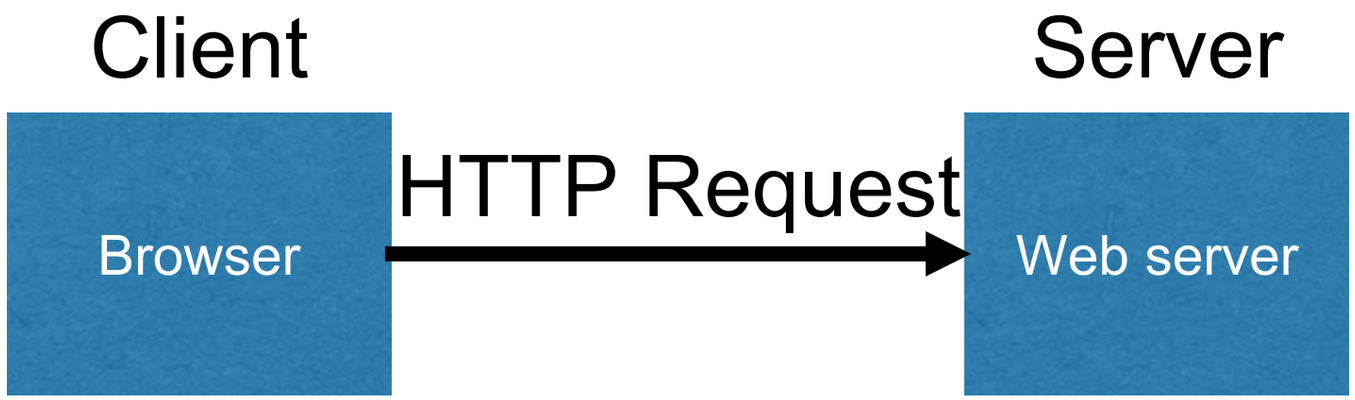
i.e., the server generates the content on the fly

Basic structure of web traffic



- HyperText Transfer Protocol (**HTTP**)
  - An “application-layer” protocol for exchanging data

Basic structure of web traffic



User clicks

- Requests contain:
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do

- Request types can be **GET** or **POST**
  - **GET**: all data is in the URL itself
  - **POST**: has data in separate fields

HTTP GET requests

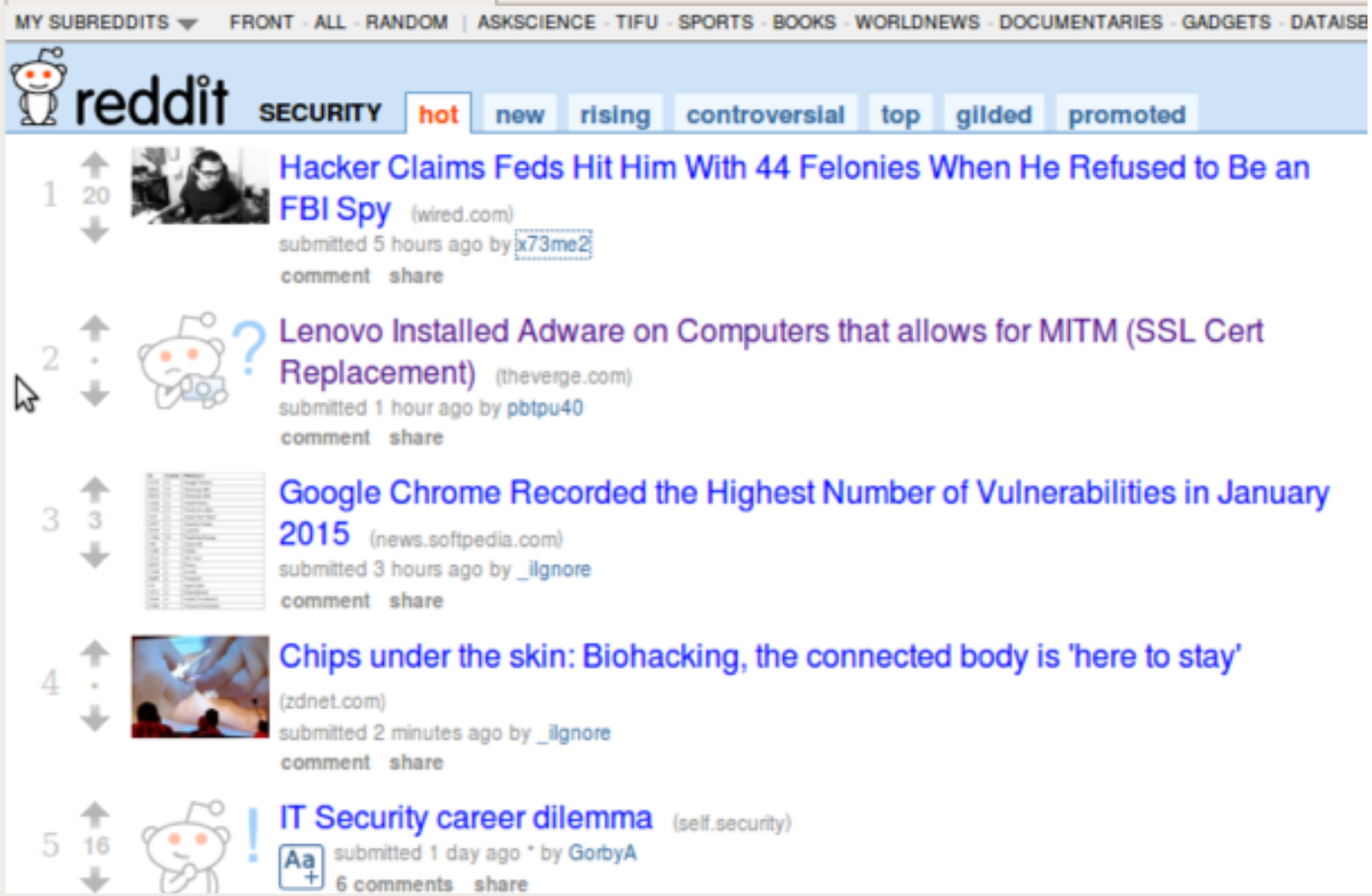
http://www.reddit.com/r/security

HTTP Headers

http://www.reddit.com/r/security

GET /r/security HTTP/1.1
Host: www.reddit.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive

User-Agent is typically a **browser**
but it can be `wget`, JDK, etc.



HTTP Headers

http://www.theverge.com/2015/2/19/8067505/lenovo-installs-adware-private-data-hackers

GET /2015/2/19/8067505/lenovo-installs-adware-private-data-hackers HTTP/1.1
Host: www.theverge.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.reddit.com/r/security

Referer URL: the site from which this request was issued.

HTTP POST requests

Posting on Piazza

HTTP Headers

https://piazza.com/logic/api?method=content.create&aid=hrteve7t83et

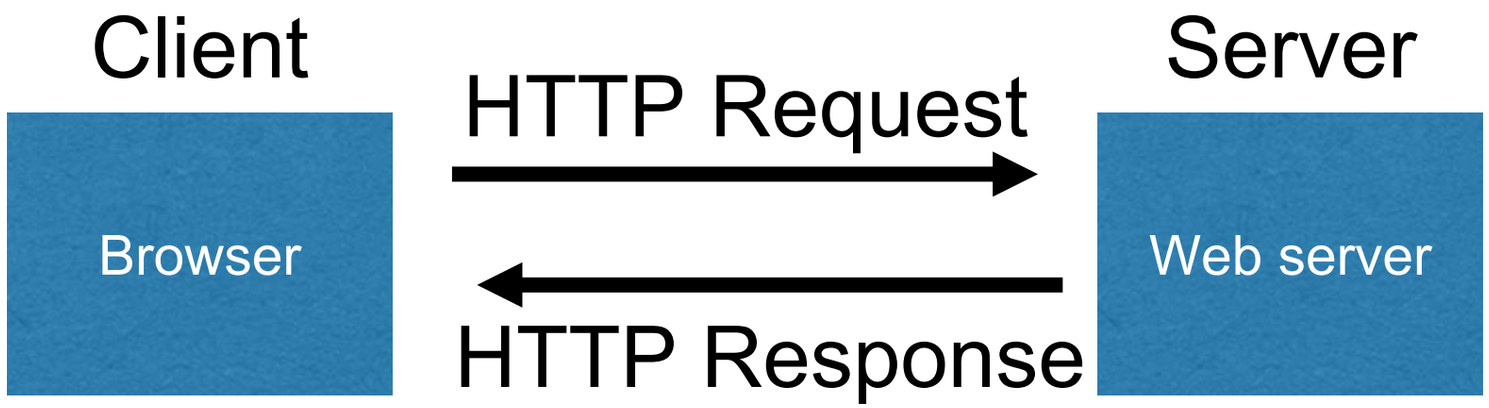
POST /logic/api?method=content.create&aid=hrteve7t83et HTTP/1.1
Host: piazza.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: https://piazza.com/class
Content-Length: 339
Cookie: piazza_session="DFwuCEFIGvEGwwHLJyuCvHIGtHKECCKL.5%25x+x+ux%255M5%22%215%3F5%26x%26%26%7C%22%21r...
Pragma: no-cache
Cache-Control: no-cache

{"method":"content.create","params":{"cid":"hrpng9q2nndos","subject":"<p>Interesting.. perhaps it has to do with a change to the ...

Implicitly includes data as a part of the URL

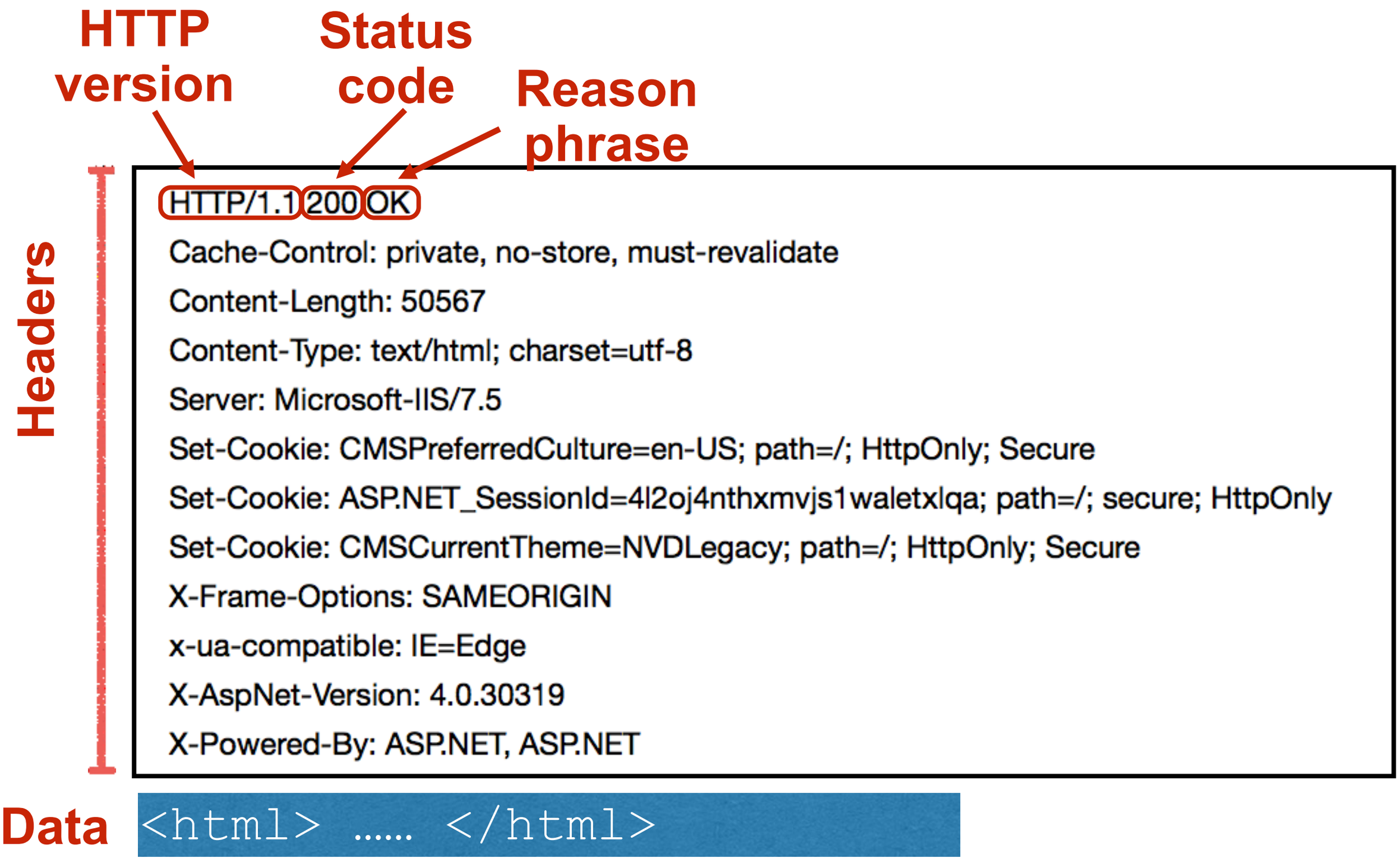
Explicitly includes data as a part of the request's content

Basic structure of web traffic

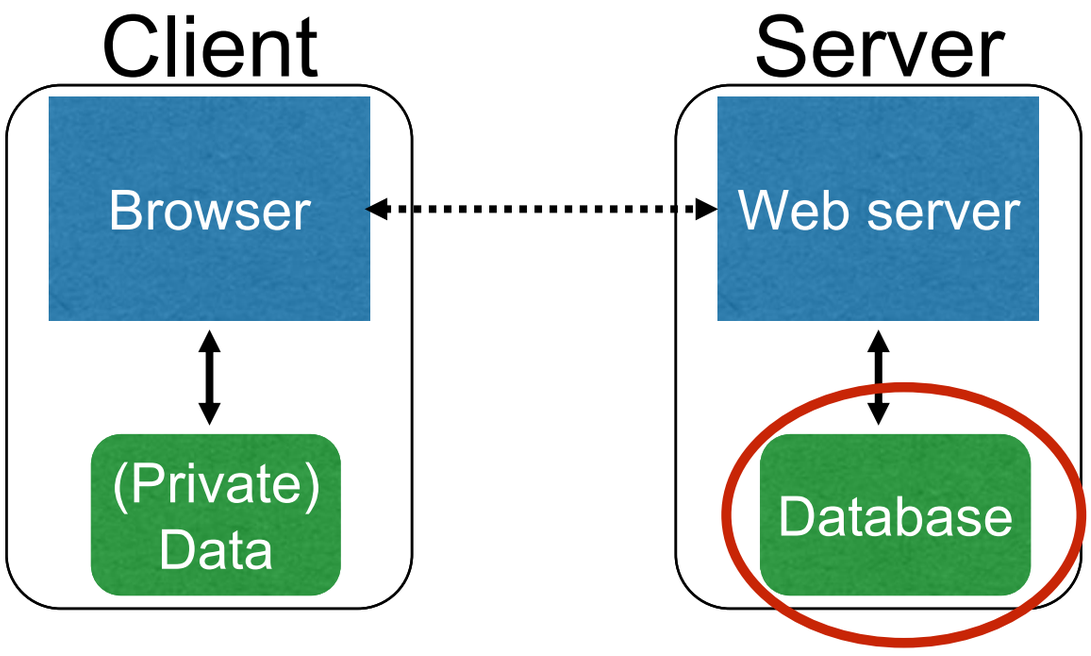


- **Responses** contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent s*tate* the server would like the browser to store

HTTP responses



Server-side data



Database: provides long-lived data storage and manipulation

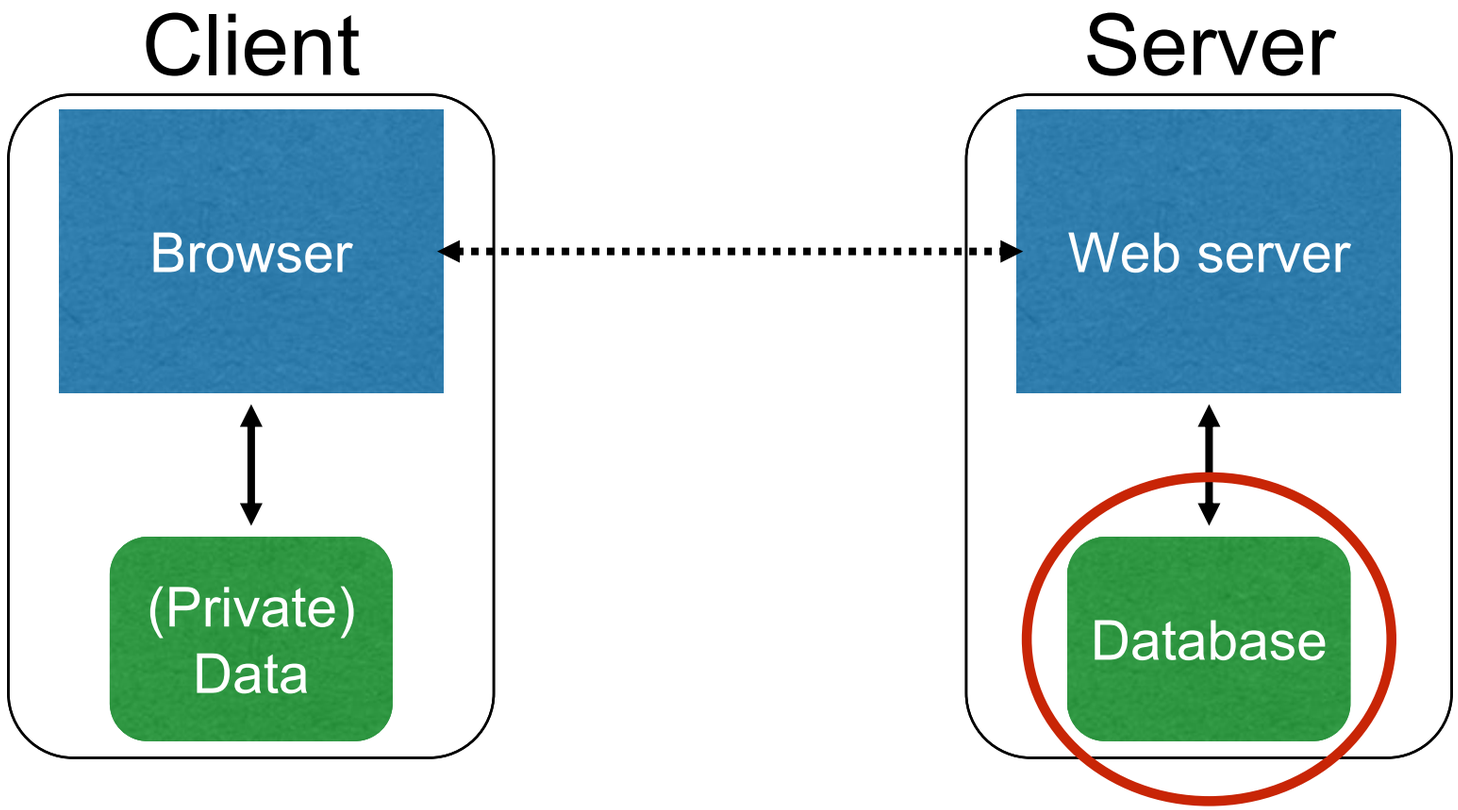
- Database Management System (DBMS) provides
  - **Transactions**: add / retrieve / modify / restructure data
  - **ACID** properties
    - **A**tomicity: transaction completes entirely or not at all
    - **C**onsistency: database stays in a valid state
    - **I**solation: transaction's effects visible only upon completion
    - **D**urability: once committed, its effects persist, eg, despite power failures

SQL databases

- There are different kinds of DBMSes, differing in
  - organization of data
  - structure of transactions
  - etc
- SQL DBMS are the most common
  - SQL: Structured Query Language
  - data is organized in **tables** (aka **relations**)
  - transactions work with the rows and columns of tables
- Newer types of DBMSes
  - data remains unstructured
  - We're not looking at these

SQL injection

Server-side data



Long-lived state, stored in a separate *database*

Need to **protect this state** from illicit access and tampering

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

SQL (Structured Query Language)

Table

Users **Table name**

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

Row (Record)

Column

Database transactions

Transactions are the unit of work on a database

"Give me everyone in the User table who is listed as taking CMSC414 in the Classes table" 2 reads

1 transaction

"Deduct $100 from Alice; Add $100 to Bob" 2 writes

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Greg';
```

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Greg';  34
```

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	greg@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Greg';  34

UPDATE Users SET Email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment
```

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	mr.uni@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Greg';  34

UPDATE Users SET Email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment
```

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	mr.uni@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Greg';  34

UPDATE Users SET Email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment

INSERT INTO Users Values('Pearl', 'F', ...);
```

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	mr.uni@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93
Pearl	F	10000	pearl@bc.com	ziog9gga

```
SELECT Age FROM Users WHERE Name='Greg';  34

UPDATE Users SET Email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment

INSERT INTO Users Values('Pearl', 'F', ...);
```

SQL (Structured Query Language)

Users

Name	Gender	Age	Email	Password
Connie	F	12	connie@bc.com	j3i8g8ha
Steven	M	14	steven@bc.com	a0u23bt
Greg	M	34	mr.uni@bc.com	0aergja
Vidalia	M	35	vidalia@bc.com	1bjb9a93
Pearl	F	10000	pearl@bc.com	ziog9gga

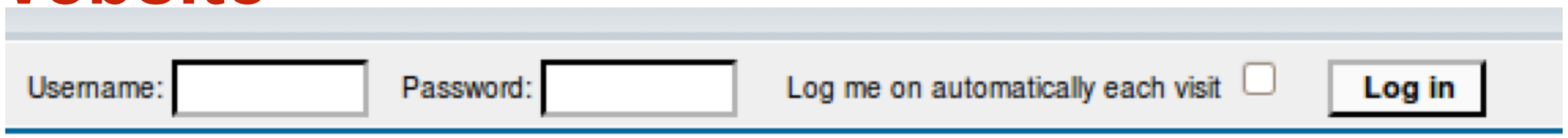
```
SELECT Age FROM Users WHERE Name='Greg';  34

UPDATE Users SET Email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment

INSERT INTO Users Values('Pearl', 'F', ...);

DROP TABLE Users;
```

SQL (Structured Query Language)

```
SELECT Age FROM Users WHERE Name='Greg';  34

UPDATE Users SET Email='mr.uni@bc.com'
   WHERE Age=34; -- this is a comment

INSERT INTO Users Values('Pearl', 'F', ...);

DROP TABLE Users;
```

Server-side code

Website



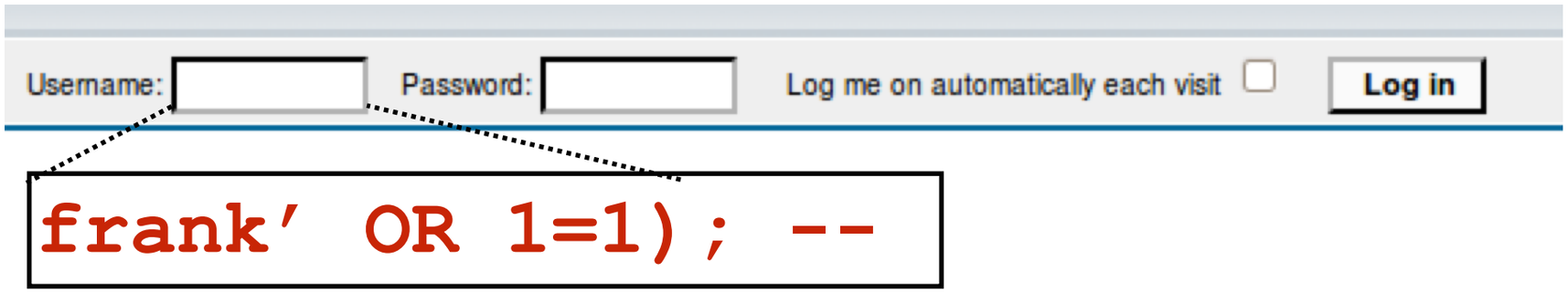
“Login code” (PHP)

```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);
```

Suppose you successfully log in as $user
if this returns any results

How could you exploit this?

SQL injection



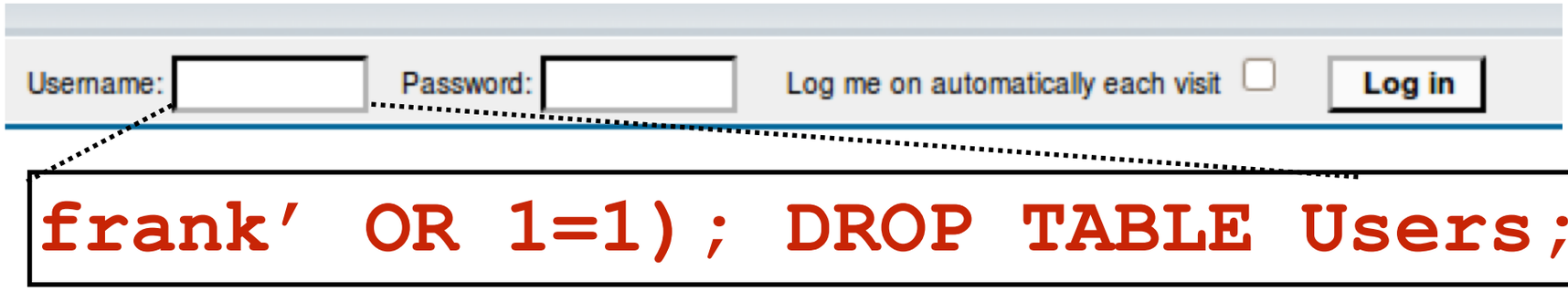

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

```
$result = mysql_query("select * from Users
        where(name='frank' OR 1=1); --
          and password='whocares');");
```

Login successful!

Problem: Data and code mixed up together

SQL injection: Worse

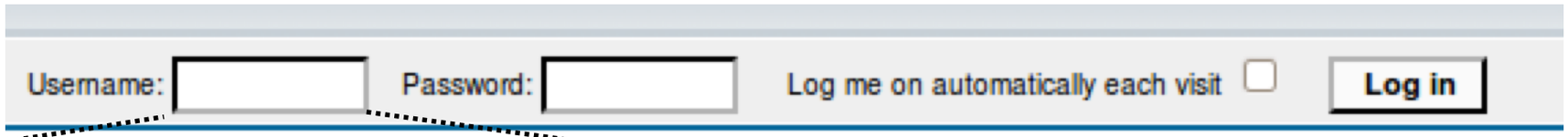



```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

```
$result = mysql_query("select * from Users
        where(name='frank' OR 1=1);
        DROP TABLE Users; --
          and password='whocares');");
```

Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2

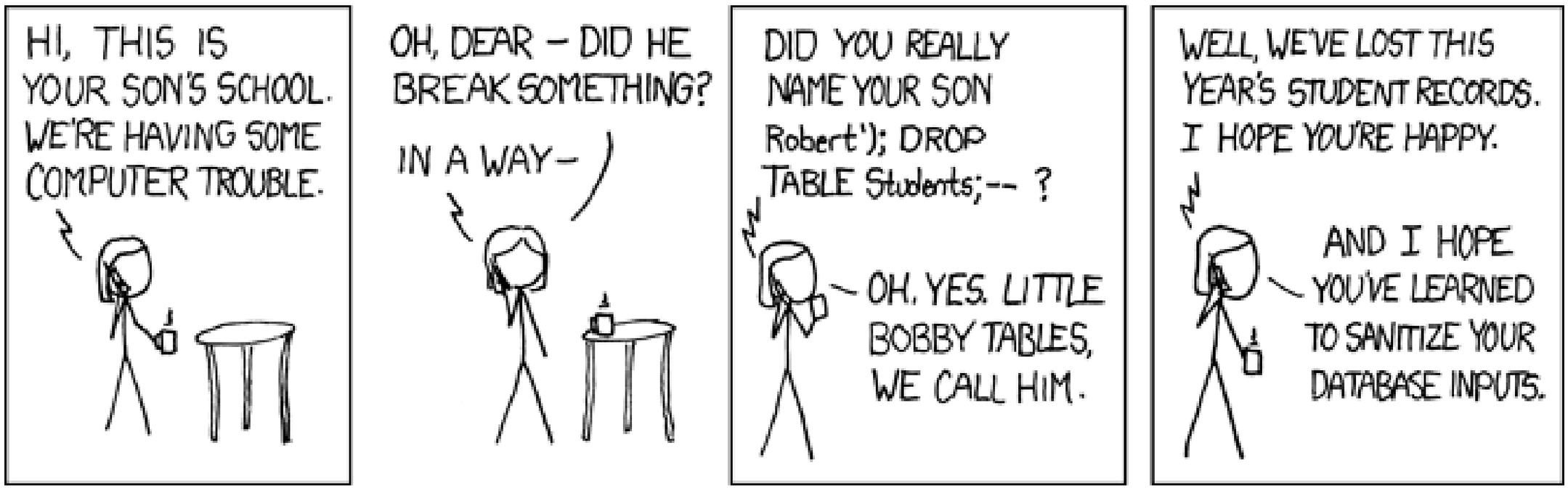
SQL injection: Even worse




```
'); EXEC cmdshell 'net user badguy backdoor / ADD'; --
```

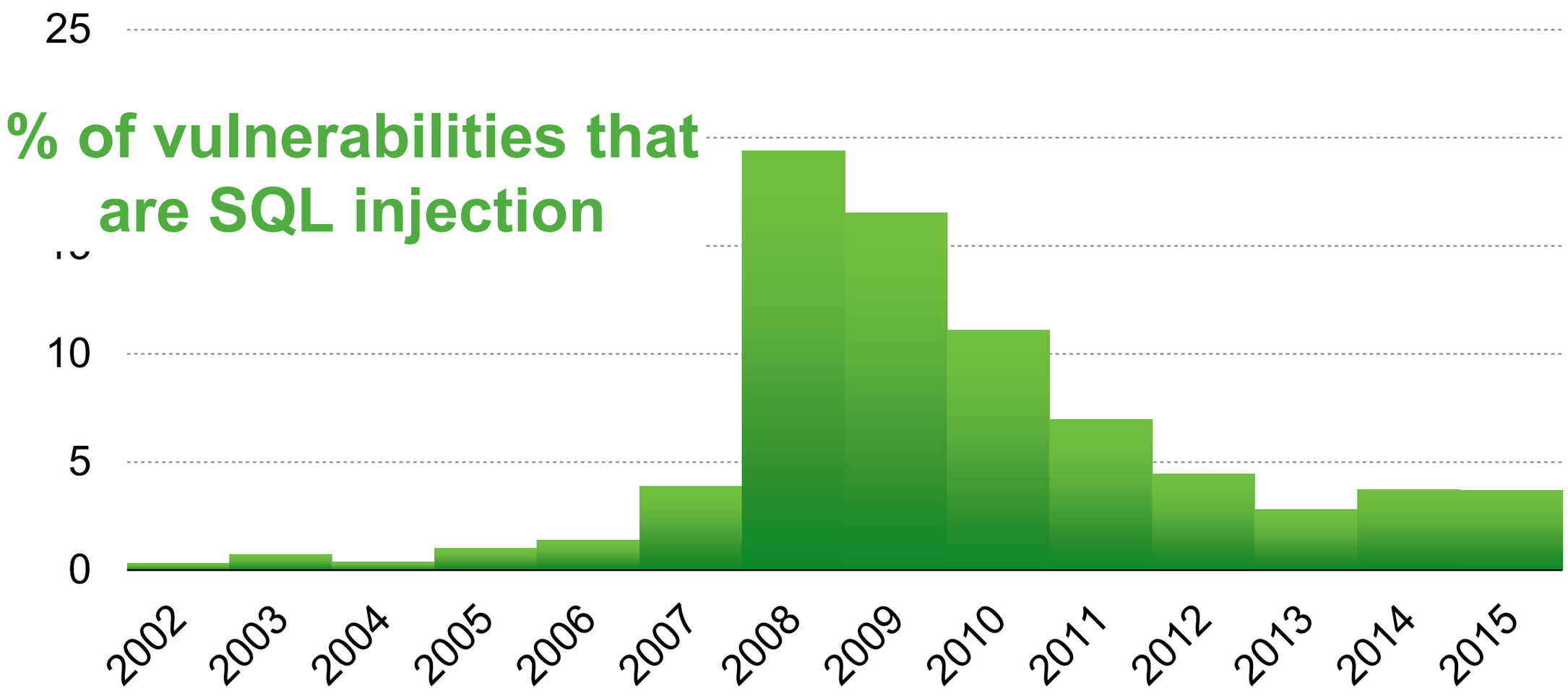
```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

```
$result = mysql_query("select * from Users
        where(name='');
        EXEC cmdshell 'net user badguy backdoor / ADD'; --
          and password='whocares');");
```



http://xkcd.com/327/

SQL injection attacks are common



http://web.nvd.nist.gov/view/vuln/statistics


ZU 0666',0,0); DROP DATABASE TABLICE;--

SQL injection countermeasures

The underlying issue

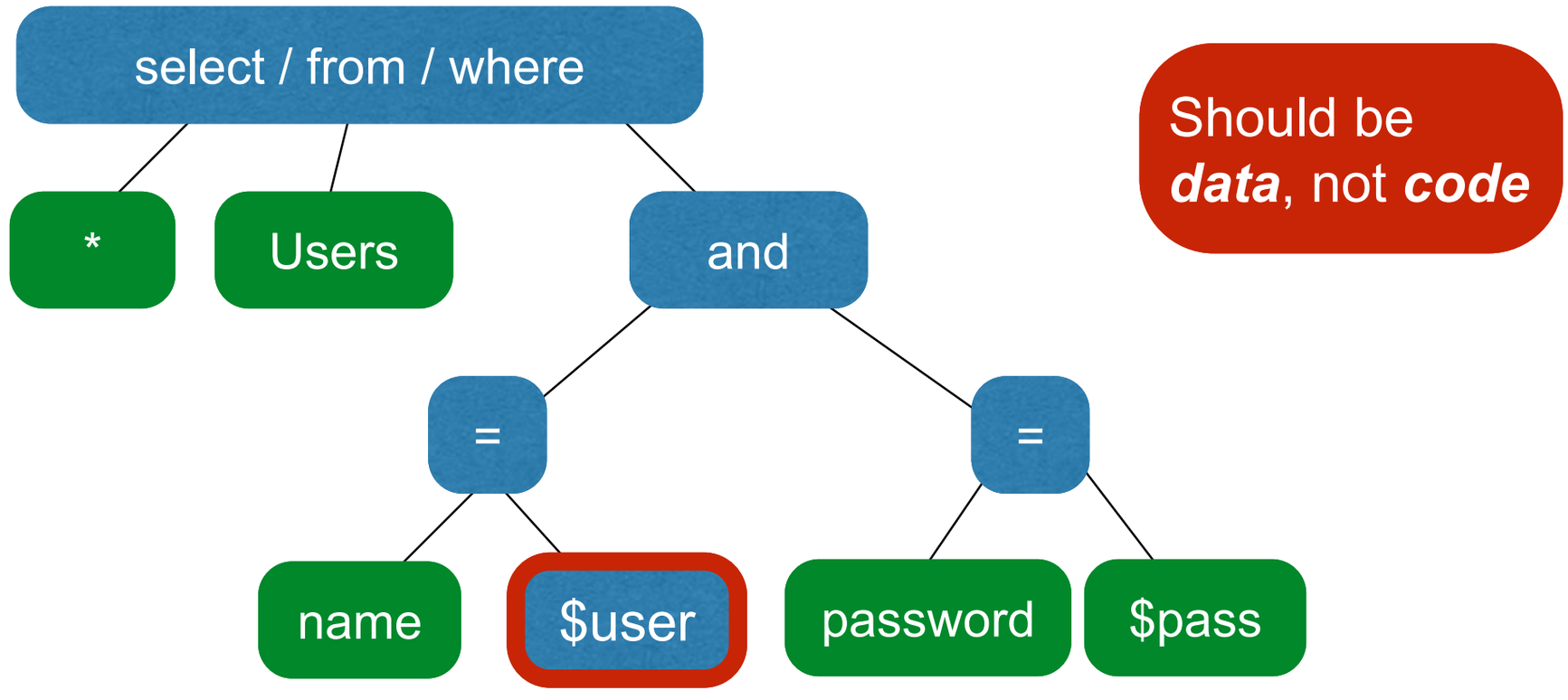
```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

- This one string combines the **code** and the **data**
- Similar to buffer overflows

When the boundary between code and data blurs, we open ourselves up to vulnerabilities

The underlying issue

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```



Should be ***data***, not ***code***

Prevention: Input validation

- We require input of a certain form, but we cannot guarantee it has that form, so we must **validate it**
  - Just like we do to avoid buffer overflows
- Making input trustworthy
  - **Check** it has the expected form, reject it if not
  - **Sanitize** by modifying it or using it such that the result is correctly formed

Sanitization: Blacklisting

`' ; --`

- **Delete** the characters you don't want
- **Downside**: "`Lupita Nyong'o`"
  - You want these characters sometimes!
  - How do you know if/when the characters are bad?
- **Downside**: How to know you've ID'd all bad chars?

Sanitization: Escaping

- **Replace** problematic characters with safe ones
  - Change `'` to `\'`
  - Change `;` to `\;`
  - Change `–` to `\–`
  - Change `\` to `\\`
- Hard by hand, there are many libs & methods
  - `magic_quotes_gpc = On`
  - `mysql_real_escape_string()`
- **Downside**: Sometimes you want these in your SQL!
  - And escaping still may not be enough

Checking: Whitelisting

- Check that the user input is **known to be safe**
  - E.g., integer within the right range
- Rationale: Given invalid input, **safer to reject than fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - Principle of fail-safe defaults
- **Downside**: Hard for rich input!
  - How to whitelist usernames? First names?

Sanitization via escaping, whitelisting, blacklisting is HARD.

Can we do better?

Sanitization: Prepared statements

- Treat user data according to its *type*
- Decouple the code and the data

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

connect to DB

```
$db = new mysql("localhost", "user", "pass", "DB");
```

prepare statement

```
$statement = $db->prepare("select * from Users
        where(name=? and password=?);");
```

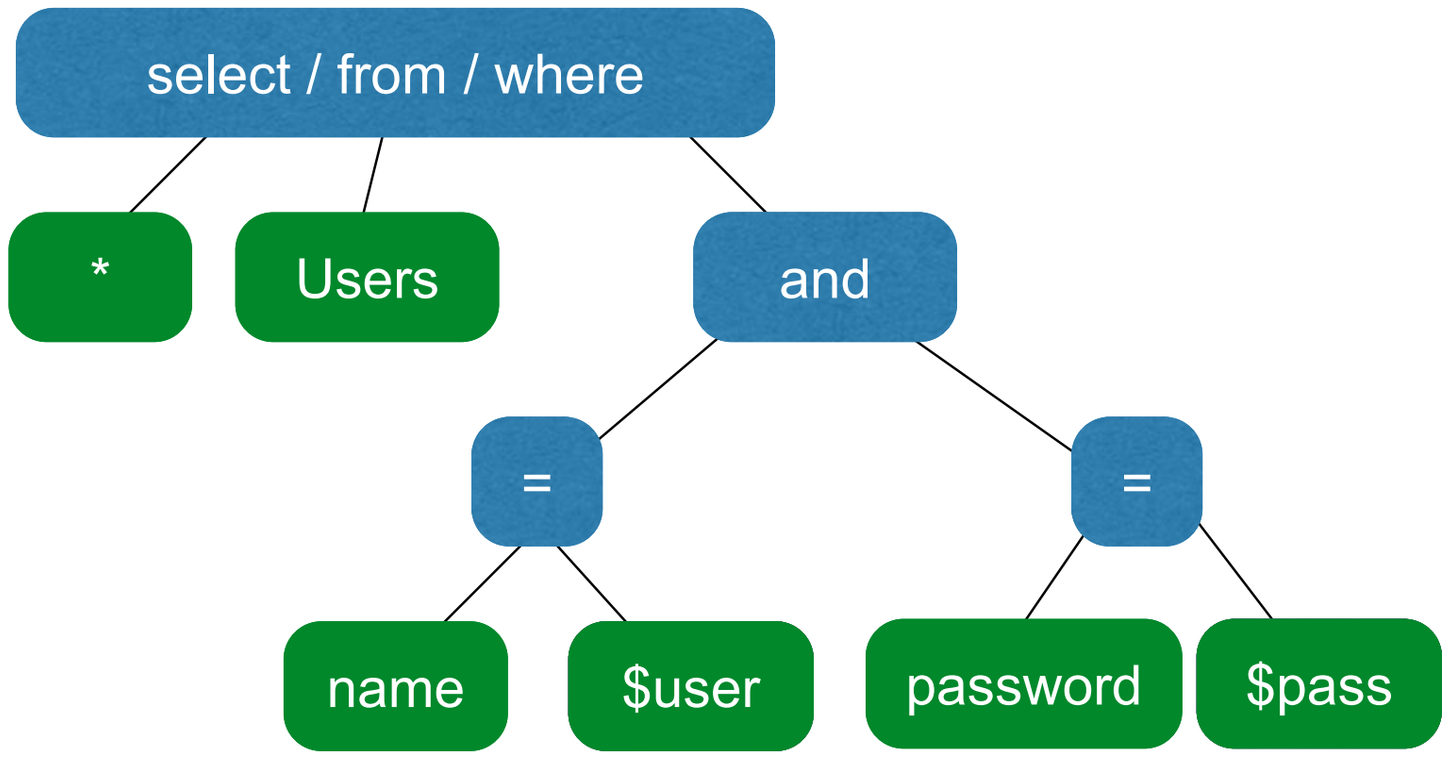
bind variables to typed data

```
$statement->bind_param("ss", $user, $pass);

$statement->execute();
```

Decoupling lets us compile now, before binding the data

Using prepared statements

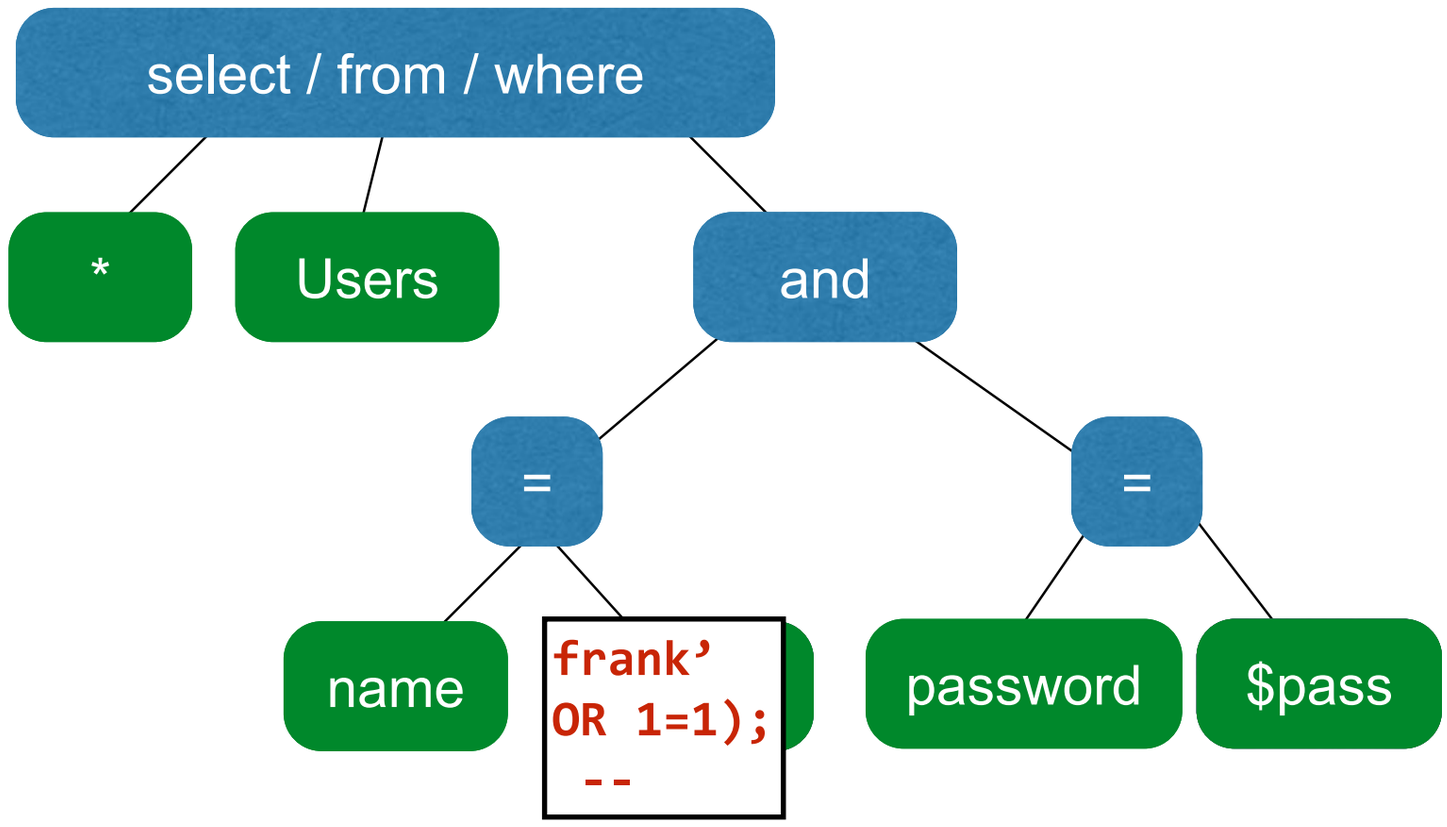
```
$statement = $db->prepare(“select * from Users
      where(name=?       and password=?);”);
$stmt->bind_param("ss", $user, $pass);
```



Binding is only applied to the leaves, so the structure of the tree is *fixed*

Using prepared statements

```
$statement = $db->prepare(“select * from Users
      where(name=?       and password=?);”);
$stmt->bind_param("ss", $user, $pass);
```



Binding is only applied to the leaves, so the structure of the tree is *fixed*

Additional mitigation

- For **defense in depth**, *also* try to mitigate any attack
  - But should **always do input validation** in any case!

- **Limit privileges**; reduces power of exploitation
  - Limit commands and/or tables a user can access
  - e.g., allow SELECT on Orders but not Creditcards

- **Encrypt sensitive data**; less useful if stolen
  - May not need to encrypt Orders table
  - But certainly encrypt creditcards.cc_numbers