Defenses

against **low-level attacks**

Outline

- **Memory safety** and **type safety**
  - Properties that, if satisfied, ensure an application is immune to memory attacks
- Automatic defenses
  - **Stack canaries**
  - Address space layout randomization (**ASLR**)
- Return-oriented programming (**ROP**) attack
  - How Control Flow Integrity (**CFI**) can defeat it
- **Secure coding**

Detecting overflows with canaries

<u>19th century coal mine integrity</u>

- Is the mine safe?
- Dunno; bring in a canary
- If it dies, abort!

We can do the same for stack integrity

ASLR today

- **Available on modern operating systems**
  - Available on Linux in 2004, and adoption on other systems came slowly afterwards; **most by 2011**
- Caveats:
  - **Only shifts the offset** of memory areas
    - Not locations within those areas
  - **May not apply to program code**, just libraries
  - **Need sufficient randomness**, or can brute force
    - 32-bit systems typically offer 16 bits = 65536 possible starting positions; sometimes 20 bits. Shacham demonstrated a brute force attack could defeat such randomness in 216 seconds (on 2004 hardware)
    - **64-bit systems more promising**, e.g., 40 bits possible

Security
for the
Web

The Web

- Previously: **Applications written in C and C++**
  - Issues like *remote code injection* and *sensitive data theft* arise from **violations of memory safety**
- Now: **Security for the World-Wide Web** (**WWW**)
  - New vulnerabilities to consider: **SQL injection**, Cross-site Scripting (**XSS**), **Session Hijacking**, and Cross-site Request Forgery (**CSRF**)
  - These share some common causes with memory safety vulnerabilities; like **confusion of code and data**
    - **Defense** also similar: **validate untrusted input**
  - New wrinkle: **Web 2.0's use of mobile code**
    - How to protect your applications and other web resources?

Web Security Outline

- Web 1.0: the basics
  - **Attack**: SQL (“sequel”) injection

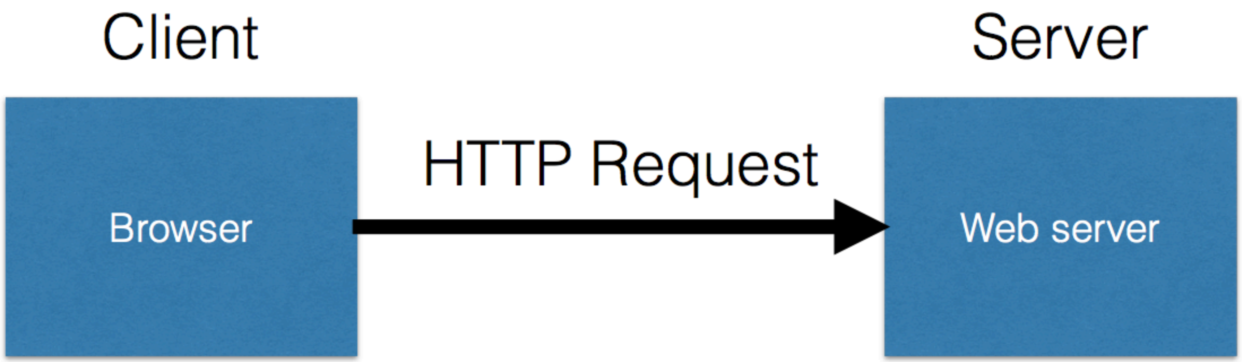
- The Web with state
  - **Attack**: Session Hijacking
  - **Attack**: Cross-site Request Forgery (CSRF)

- Web 2.0: The advent of Javascript
  - **Attack**: Cross-site Scripting (XSS)

- **Defenses throughout**
  - *Theme*: **validate or sanitize input**, then trust it

Web Basics

Basic structure of web traffic

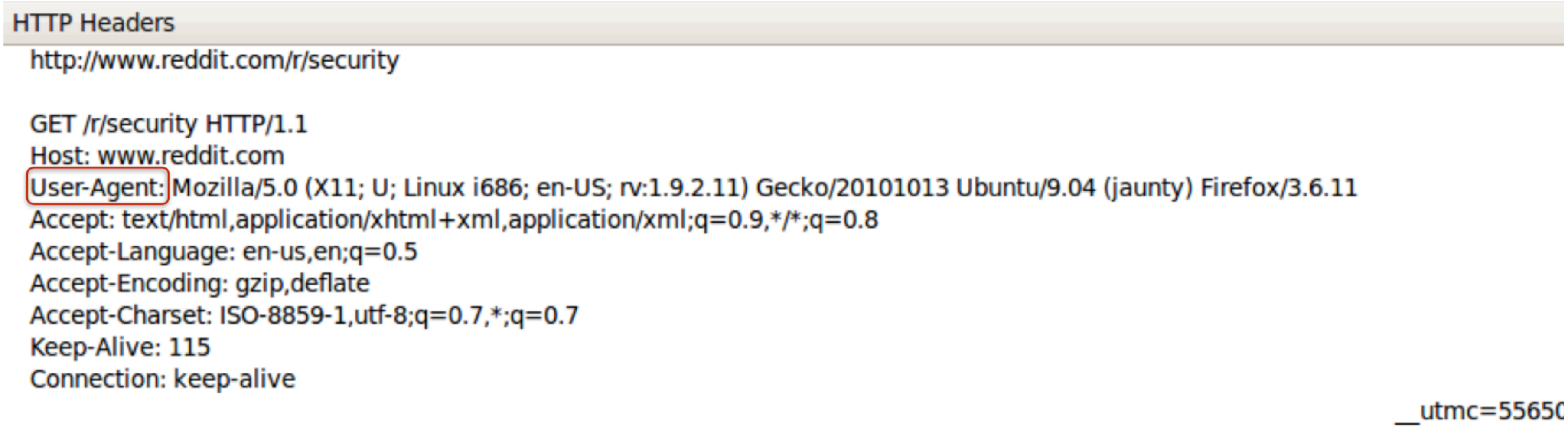


User clicks

- **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 (no server side effects)
  - **POST**: includes the data as separate fields (can have side effects)

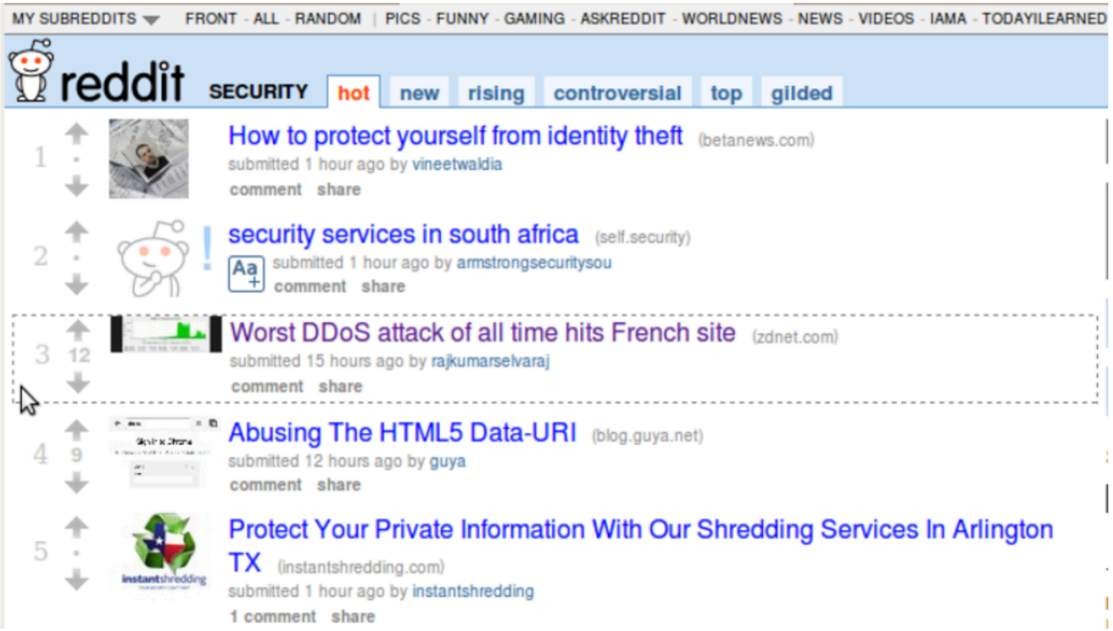
HTTP GET requests

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



```
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
                                                                                          __utmc=55650
```

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



MY SUBREDDITS ▾ FRONT - ALL - RANDOM | PICS - FUNNY - GAMING - ASKREDDIT - WORLDNEWS - NEWS - VIDEOS - IAMA - TODAYILEARNED

reddit SECURITY hot new rising controversial top gilded

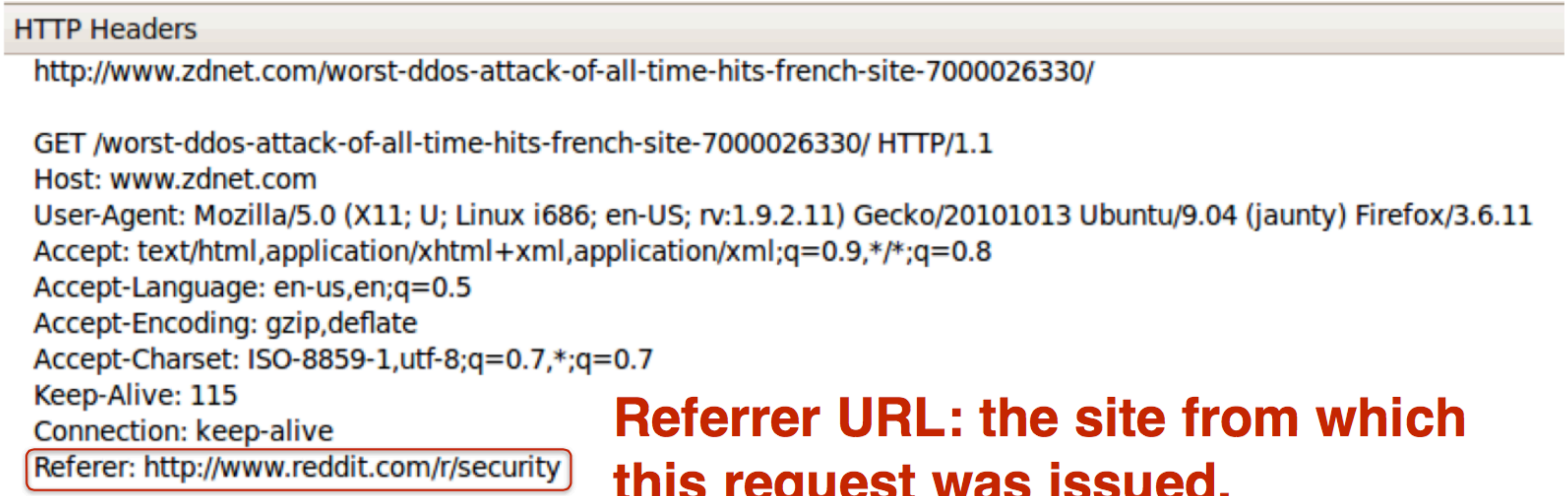
1 · How to protect yourself from identity theft (betanews.com)
submitted 1 hour ago by vineetwaldia
comment share

2 · security services in south africa (self.security)
submitted 1 hour ago by armstrongsecuritysou
comment share

3 12 Worst DDoS attack of all time hits French site (zdnet.com)
submitted 15 hours ago by rajkumarselvaraj
comment share

4 9 Abusing The HTML5 Data-URI (blog.guya.net)
submitted 12 hours ago by guya
comment share

5 · Protect Your Private Information With Our Shredding Services In Arlington TX (instantshredding.com)
submitted 1 hour ago by instantshredding
1 comment share



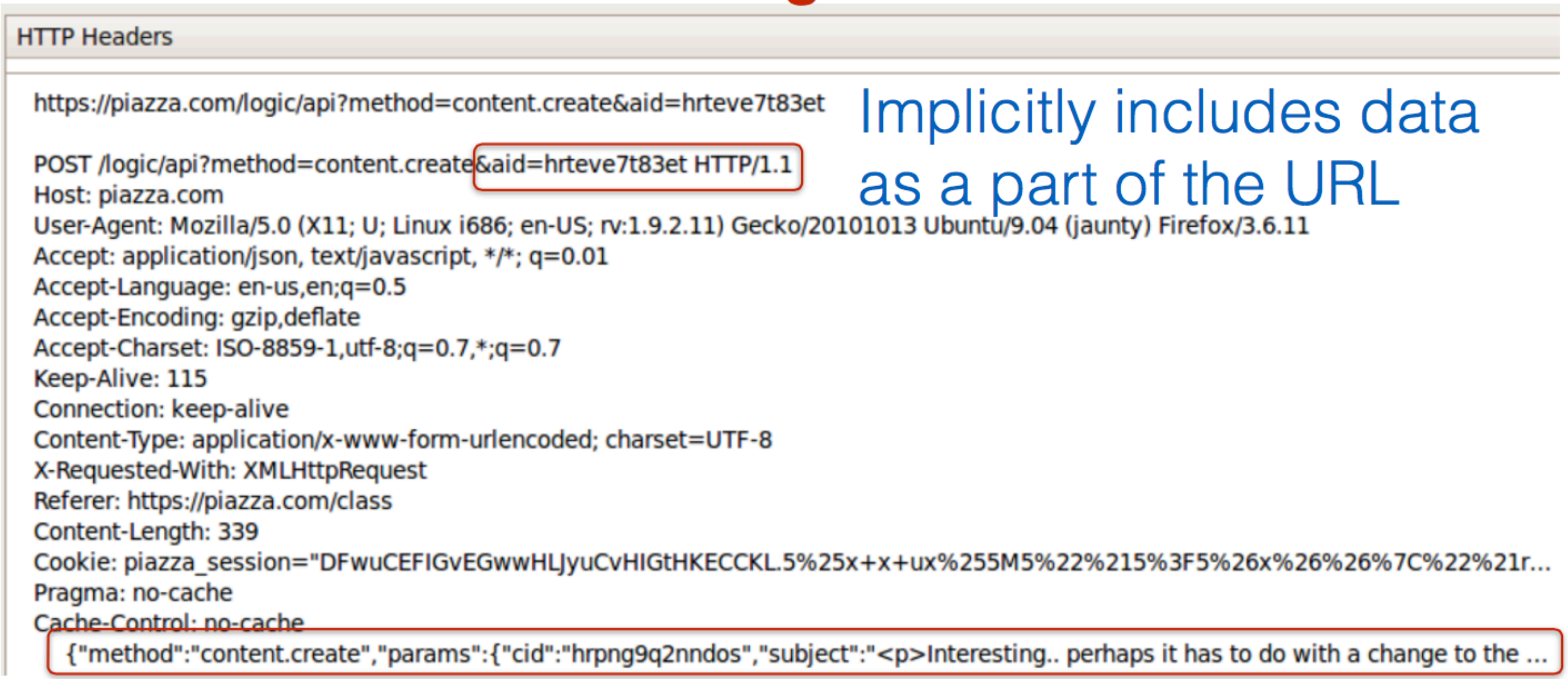
HTTP Headers

```
http://www.zdnet.com/worst-ddos-attack-of-all-time-hits-french-site-7000026330/

GET /worst-ddos-attack-of-all-time-hits-french-site-7000026330/ HTTP/1.1
Host: www.zdnet.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
```

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

HTTP POST requests

Posting on Piazza



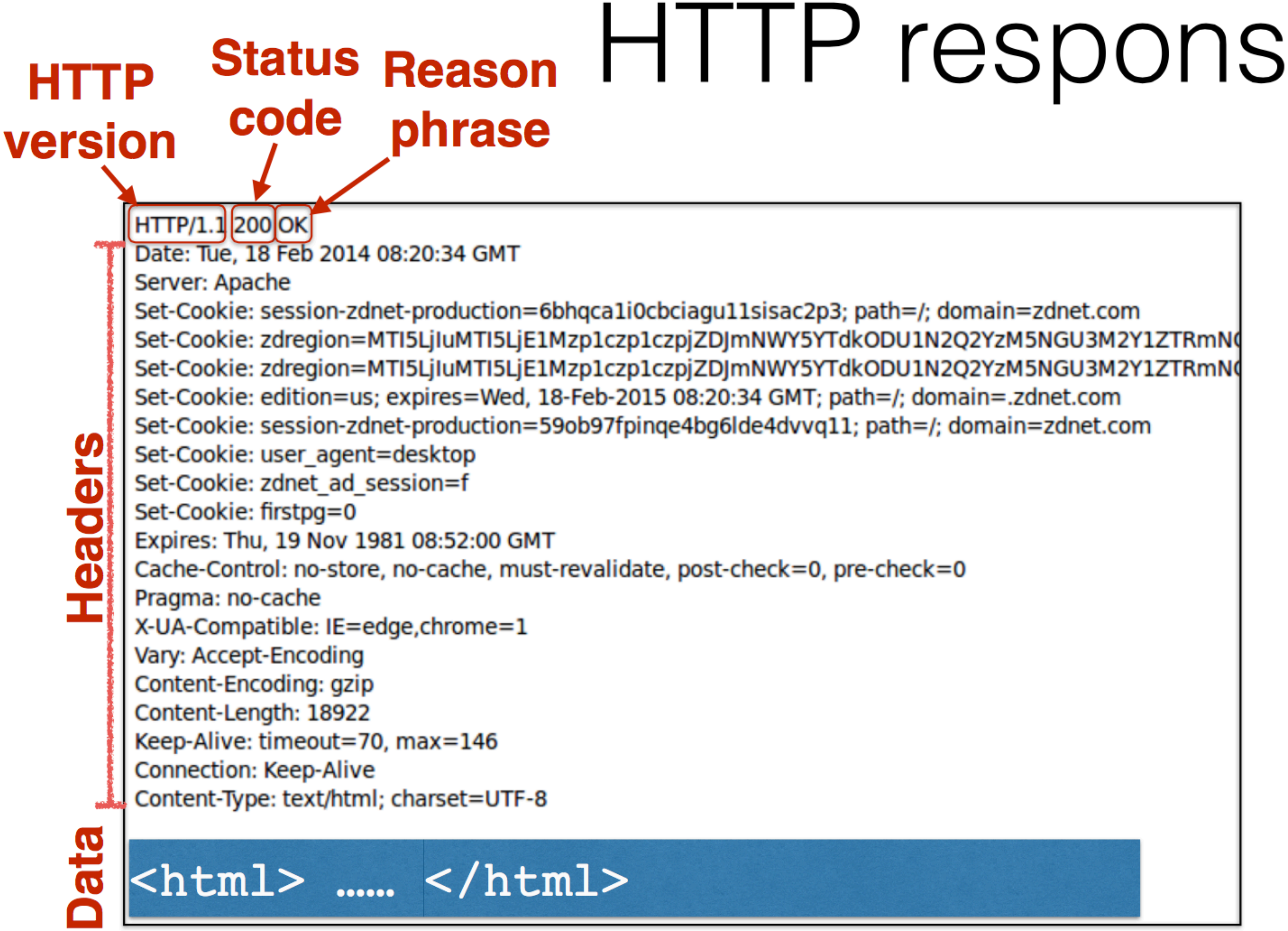
HTTP Headers

```
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

HTTP responses



HTTP version

Status code

Reason phrase

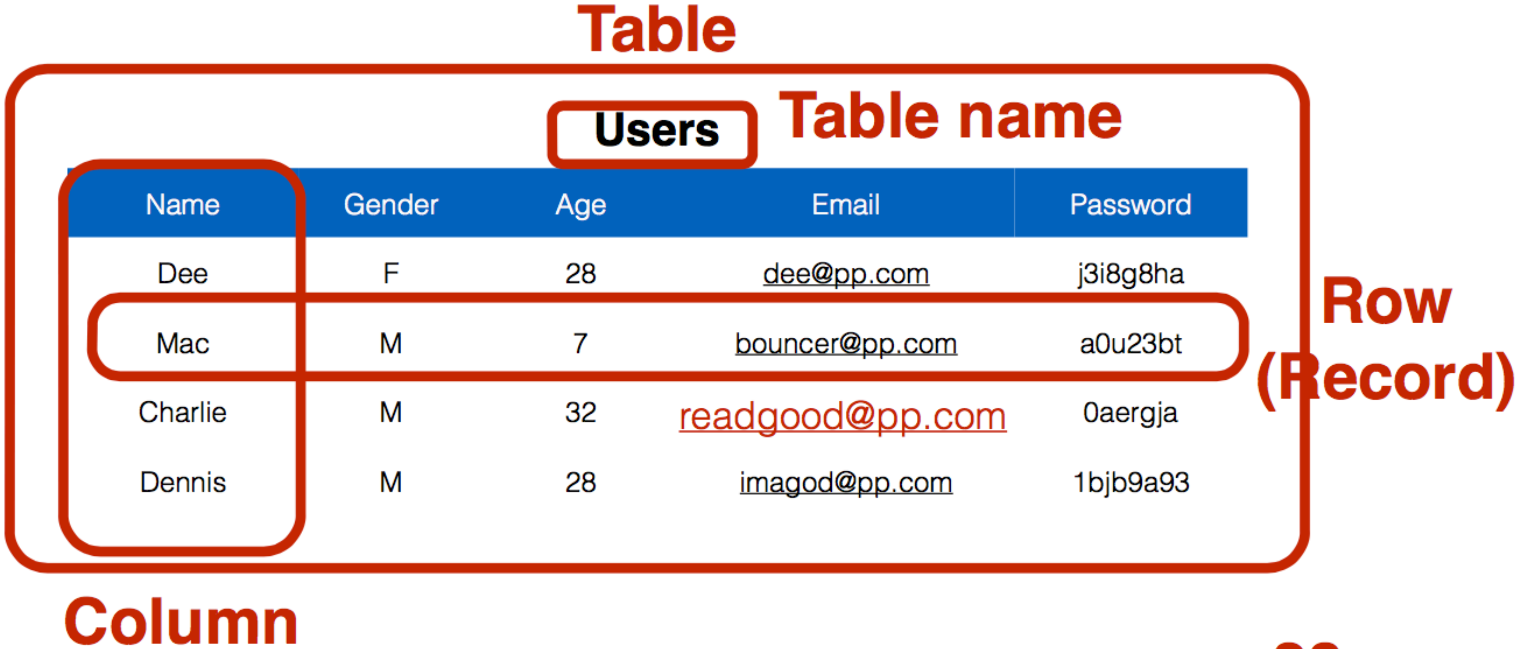
Headers

Data

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<html> ...... </html>
```

SQL injection

SQL (Standard Query Language)



Table

Table name

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

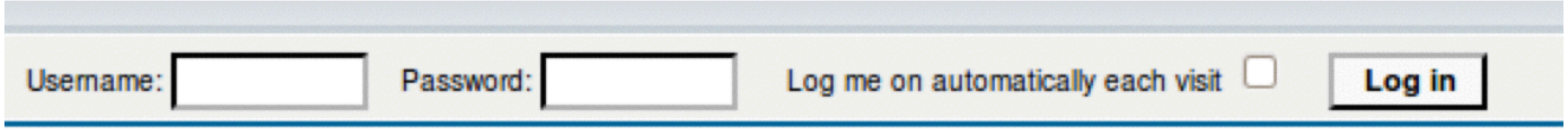
Row (Record)

Column

```
SELECT Age FROM Users WHERE Name='Dee';      28

UPDATE Users SET email='readgood@pp.com'
   WHERE Age=32; -- this is a comment

INSERT INTO Users Values('Frank', 'M', 57, ...);
DROP TABLE Users;
```

Server-side code

Website



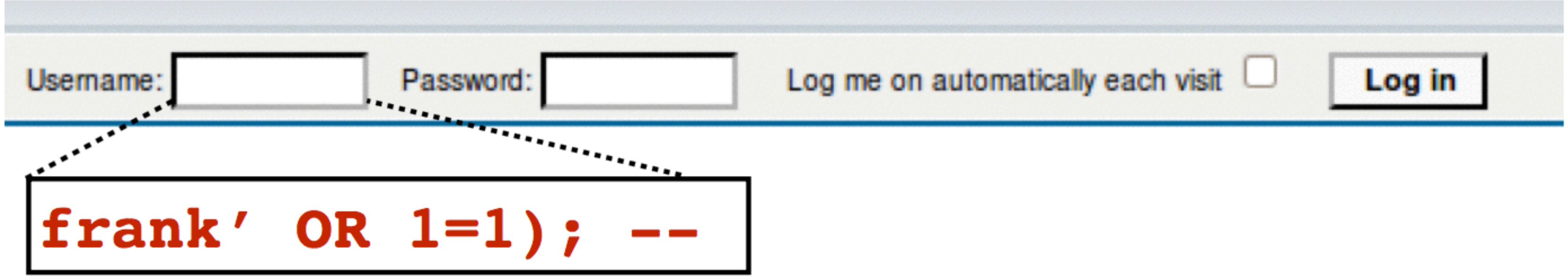
“Login code” (PHP)

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

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

How could you exploit this?

SQL injection

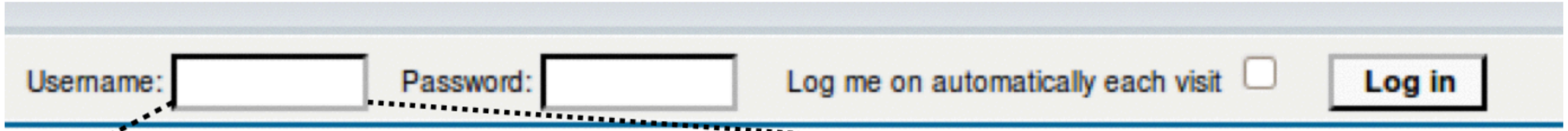


```
frank’ OR 1=1); --
```

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

```
$result = mysql_query(“select * from Users
        where(name=‘frank’ OR 1=1); --
      and password=‘whocares’);”);
```

SQL injection

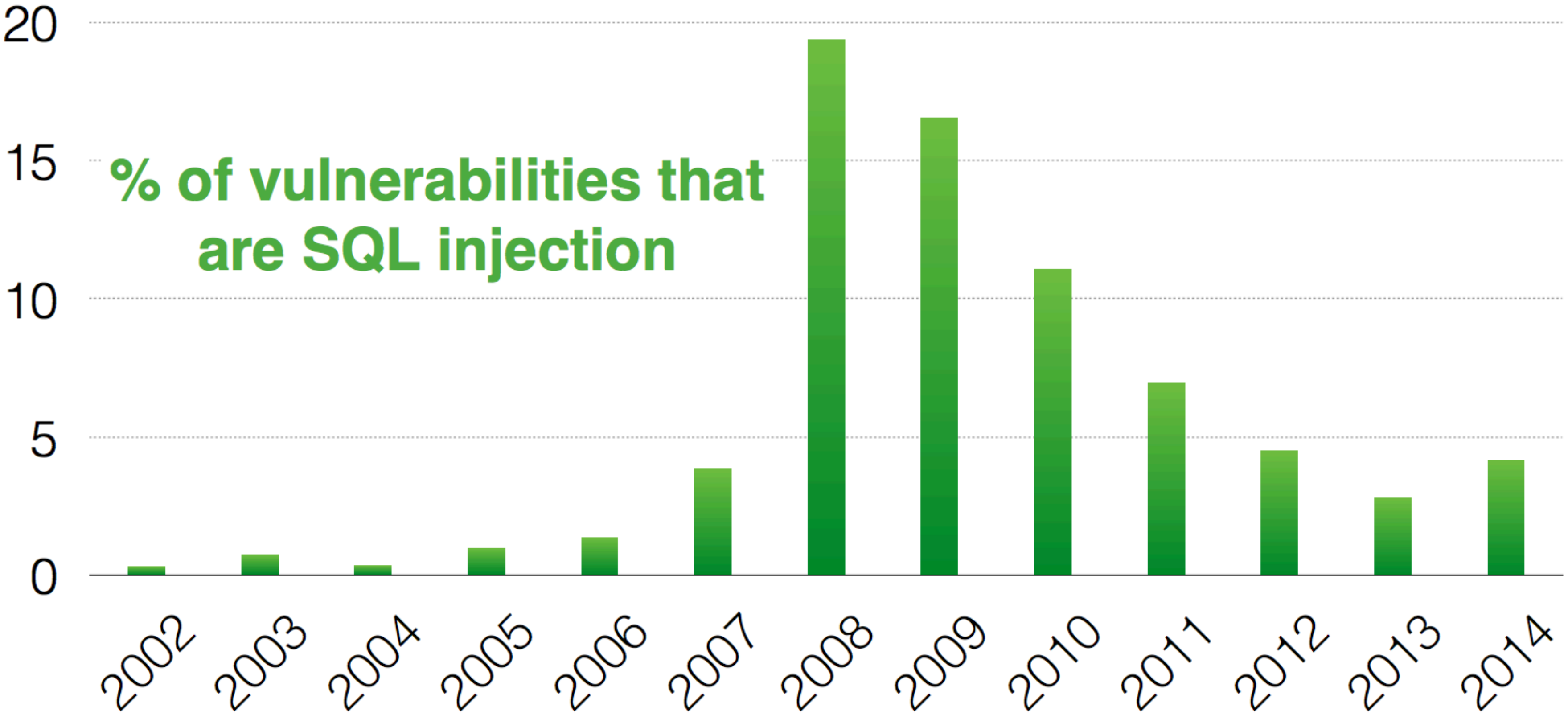


```
frank’ OR 1=1); DROP TABLE Users; --
```

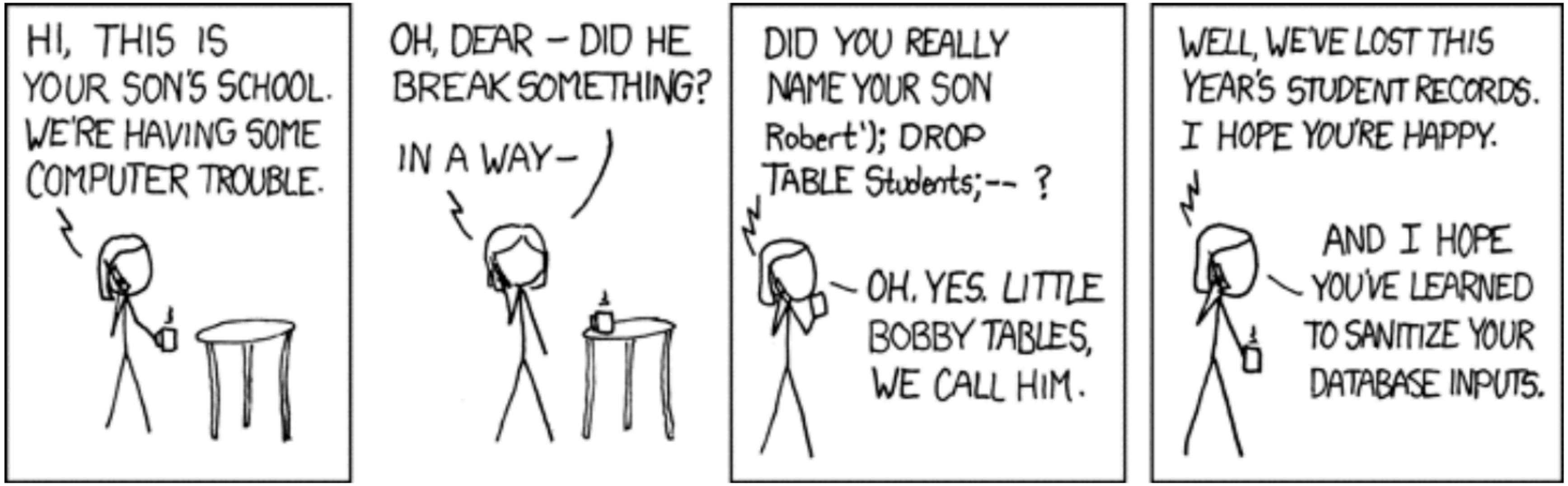
```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);

$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 attacks are common



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


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.

http://xkcd.com/327/

The underlying issue

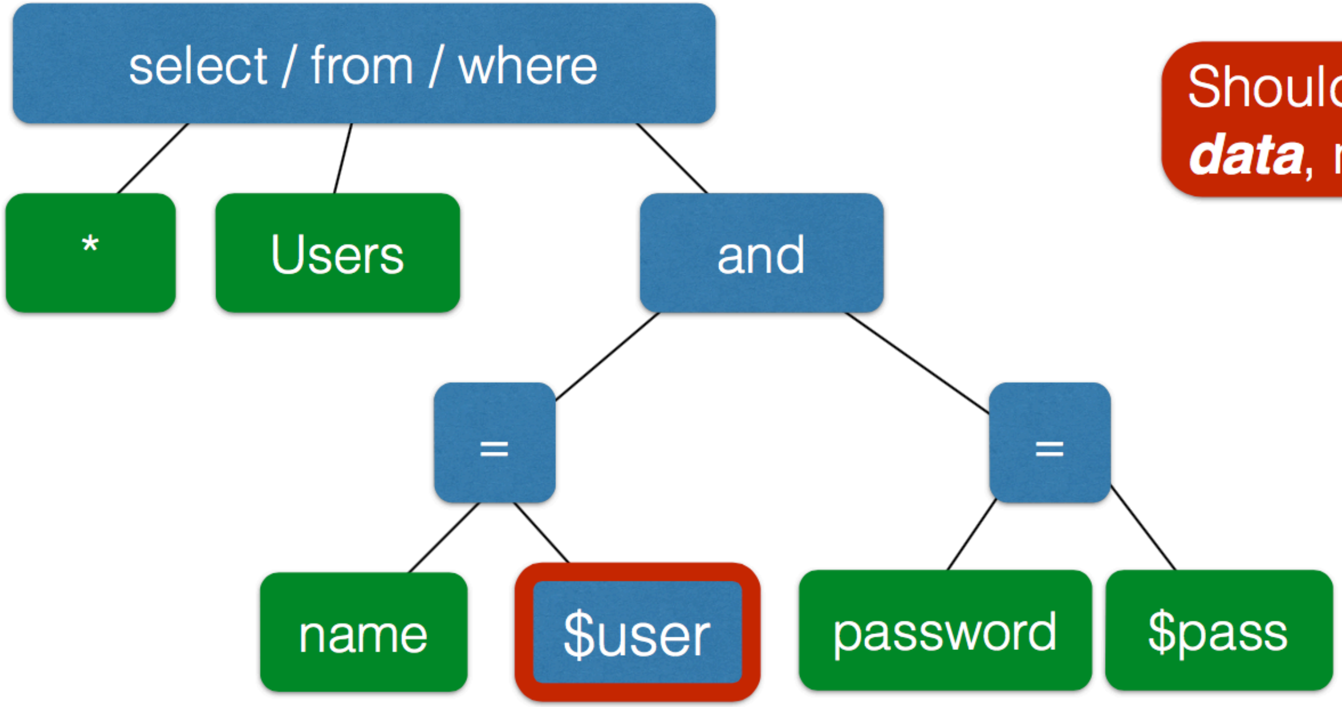
```
$result = mysql_query(“select * from Users
       where(name=‘$user’ and password=‘$pass’);”);
```

- 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

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

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’);”);
```

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

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

Bind variables

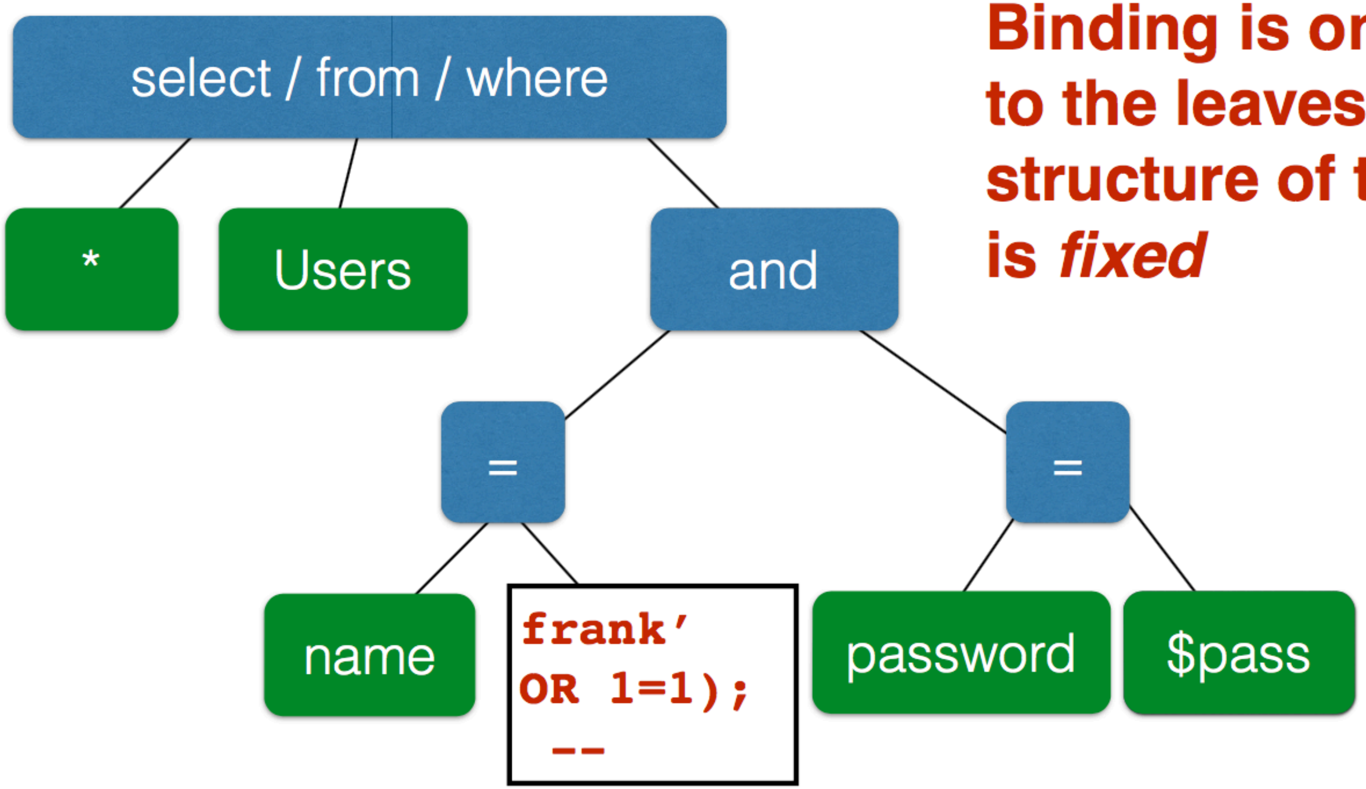
Decoupling lets us compile now, before binding the data

```
$statement->bind_param(“ss”, $user, $pass);
$statement->execute();
```

Bind variables are typed

Using prepared statements

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



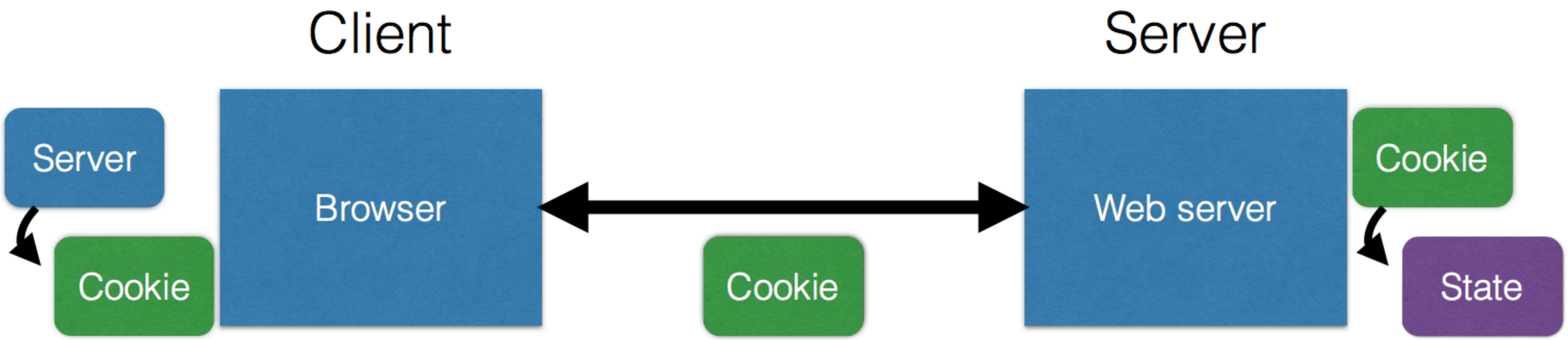
Binding is only applied to the leaves, so the structure of the tree is *fixed*

Session Hijacking

Cookies and web authentication

- An *extremely common* use of cookies is to track users who have already authenticated
- If the user already visited
  `http://website.com/login.html?user=alice&pass=secret`
  with the correct password, then the server associates a *“session cookie”* with the logged-in user’s info
- Subsequent requests include the cookie in the request headers and/or as one of the fields:
  `http://website.com/doStuff.html?sid=81asf98as8eak`
- The idea is to be able to say “I am talking to the same browser that authenticated Alice earlier.”

Stealing Session Cookies



- **Compromise** the server or user's machine/browser
- **Predict** it based on other information you know
- **Sniff** the network
- **DNS cache poisoning**
  - Trick the user into thinking you are Facebook
  - The user will send you the cookie

Network-based attacks

Mitigating Hijack

- Sad story: **Twitter**
- Uses one cookie (**auth_token**) to validate user, which is a function of
  - User name, password
- **auth_token** weaknesses
  - *Does not change* from one login to the next
  - *Does not become invalid* when the user logs out
  - Thus: **steal this cookie once**, and you can **log in as the user any time you want** (until password change)!
- **Defense**: **Time out** session IDs and **delete** them once the session ends

http://packetstormsecurity.com/files/119773/twitter-cookie.txt

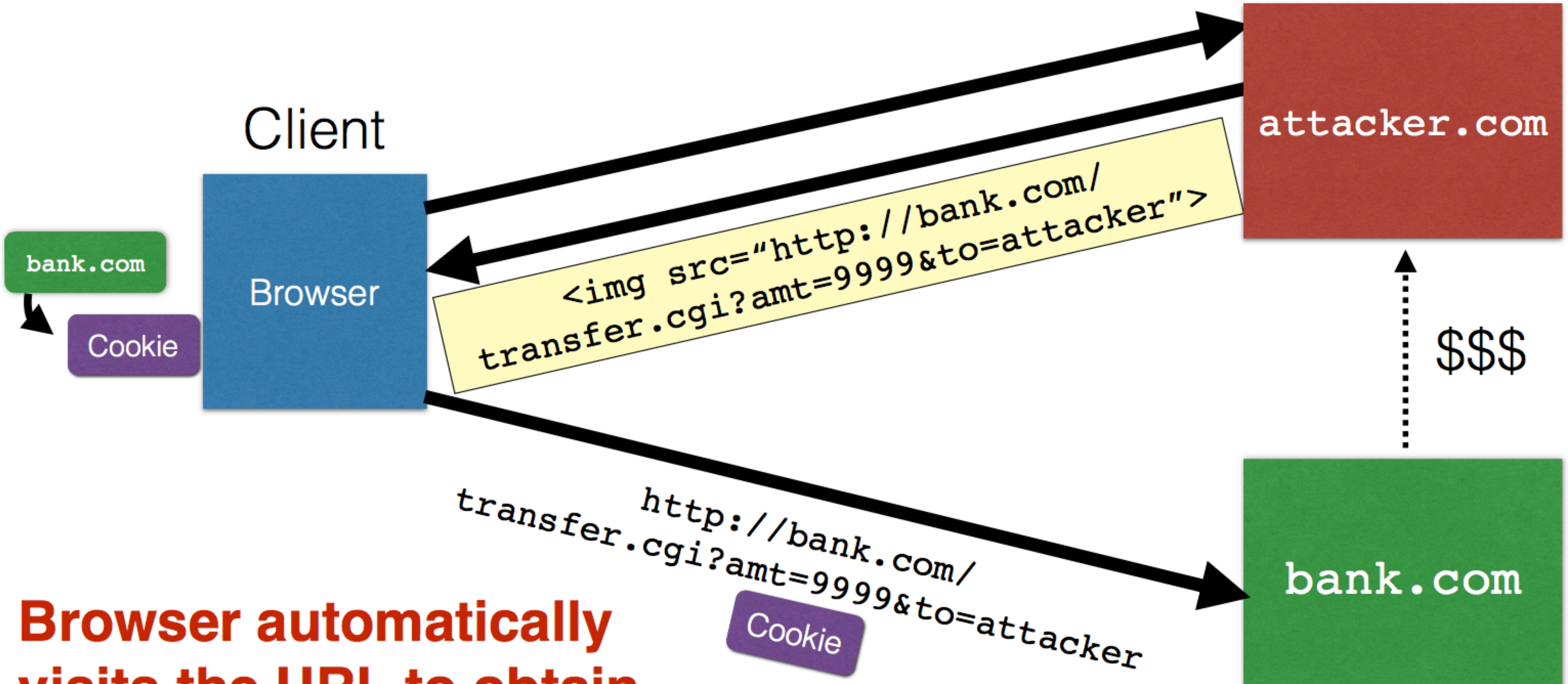
Cross-Site Request Forgery (CSRF)

URLs with side effects

```
http://bank.com/transfer.cgi?amt=9999&to=attacker
```

- GET requests often have **side effects on server state**
  - Even though they are not supposed to
- What happens if
  - the **user is logged in** with an active session cookie
  - a **request is issued for the above link?**
- How could you get a user to visit a link?

Exploiting URLs with side-effects



Client

Browser

bank.com

Cookie

attacker.com

<img src="http://bank.com/transfer.cgi?amt=9999&to=attacker">

$$$

bank.com

http://bank.com/transfer.cgi?amt=9999&to=attacker

Cookie

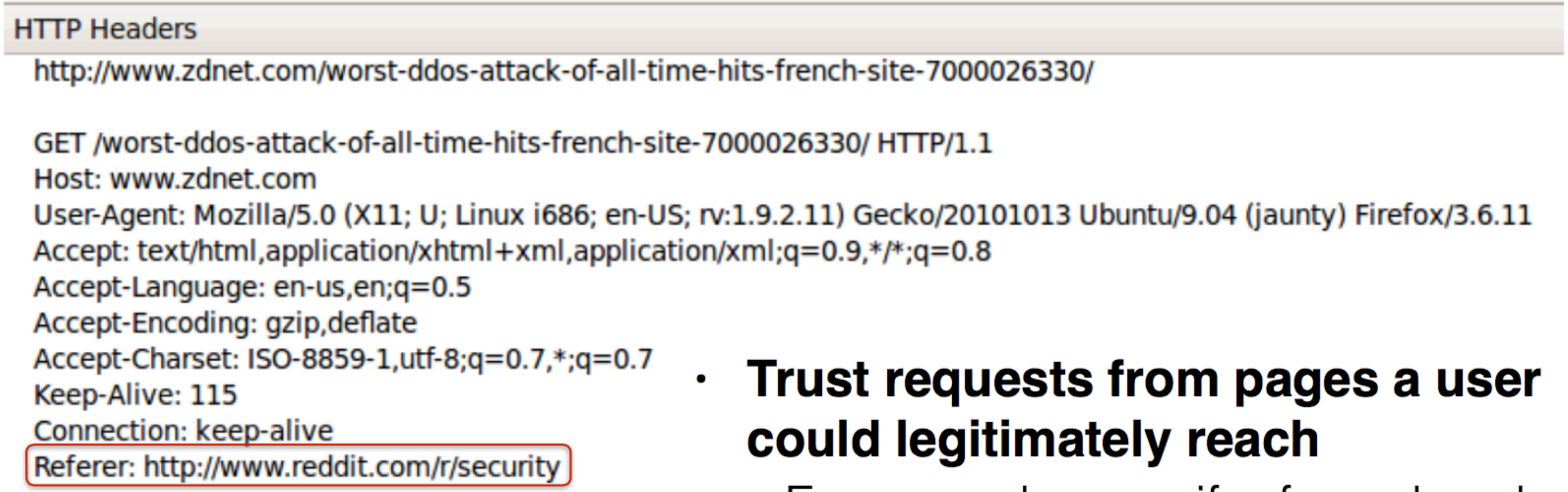
Browser automatically visits the URL to obtain what it believes will be an image

Cross-Site Request Forgery

- Target: User who has an account on a vulnerable server
- Attack goal: make requests to the server *via the user's browser* that look to the server like the user intended to make them
- Attacker tools: ability to get the user to "click a link" crafted by the attacker that goes to the vulnerable site
- Key tricks:
  - Requests to the web server have predictable structure
  - Use of something like `<img src=…>` to force the victim to send it

CSRF protections: REFERER

- The browser will set the **REFERER** field to the page that hosted a clicked link



```
HTTP Headers
 http://www.zdnet.com/worst-ddos-attack-of-all-time-hits-french-site-7000026330/

 GET /worst-ddos-attack-of-all-time-hits-french-site-7000026330/ HTTP/1.1
 Host: www.zdnet.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
```

- **Trust requests from pages a user could legitimately reach**
  - From good users, if referrer header present, generally trusted
  - Defends against session hijacks too

Problem: Referrer optional

- Not included by all browsers
    - Sometimes other legitimate reasons not to have it
- Response: **lenient referrer checking**
    - Blocks requests with a bad referrer, but allows requests with no referrer
    - *Missing referrer always harmless?*
- **No:** attackers can **force the removal of referrer**
    - **Bounce** user off of `ftp:` page
    - **Exploit browser vulnerability** and remove it
    - **Man-in-the-middle** network attack

CSRF Protection: Secretized Links

- **Include a secret in every link/form**
  - Can use a hidden form field, custom HTTP header, or encode it directly in the URL
  - Must not be guessable value
  - Can be same as session id sent in cookie
- **Frameworks help**: Ruby on Rails embeds secret in every link automatically

http://website.com/doStuff.html?sid=81asf98as8eak