

# Browser code isolation

John Mitchell

# Modern web sites are complex

The screenshot displays the New York Times website's regional page for New York. The browser's address bar shows the URL: [www.nytimes.com/pages/nyregion/index.html?module=HPMiniNav&contentCollection=New York&pgtype=Homepage&region=TopBar&action=click&t](http://www.nytimes.com/pages/nyregion/index.html?module=HPMiniNav&contentCollection=New York&pgtype=Homepage&region=TopBar&action=click&t). The page features a navigation bar with the New York Times logo, a search icon, and the text "N.Y. / Region". On the right side of the navigation bar, there are buttons for "SUBSCRIBE NOW", "SIGN IN", and "Register" with a settings gear icon.

A prominent advertisement for Volkswagen is displayed below the navigation bar. The ad includes the text "Volkswagen TDI CleanDieselEvent", "Explore Offers", "SERRAMONTE VOLKSWAGEN", and "Lease a 2014 Golf TDI for \$289/mo for 36 months. \$0 Down". It also features an image of a red Volkswagen Golf and the Volkswagen logo with the slogan "Das Auto."

The main content area is divided into several sections. On the left, there is a news article titled "In New York, Hard Choices on Health Exchange Spell Success" by ANEMONA HARTOCCOLIS, published 49 minutes ago. The article includes a photograph of a woman at a desk and a caption: "Cheng W. Lee/The New York Times". The text of the article states: "Maika Percal, 63, had her blood pressure checked and got an EKG test at a doctor's appointment. Her co-pay was \$75. More than 900,000 residents signed up for health plans, and premiums have dropped, though the state limited consumers' choices."

To the right of the article is a "SIDE STREET" section titled "An Artist Takes His Pay in Coffee and Community" by DAVID GONZALEZ. The text describes David Ellis, who gives back to his neighborhood by painting a mural in a bodega in Fort Greene, Brooklyn. Below this is a "BIG CITY BOOK CLUB" section titled "Brooklyn today is a destination for celebrities and wealthy creative types," accompanied by a small image of a building with "Brooklyn" written on it.

On the far right, there is a social media section for "@NYTMETRO ON TWITTER" with a "FOLLOW" button and a "More New York Videos" section featuring a video player with a play button icon and a portrait of a man.

# 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, ...

# Others want this information

- ◆ Financial data
  - Black-hat hackers, ...
- ◆ Health data
  - Insurance companies, ...
- ◆ Personal data
  - Ad companies, big government, ...



# Modern web "site"



Code from many sources  
Combined in many ways

# Basic questions

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

## Third-party APIs



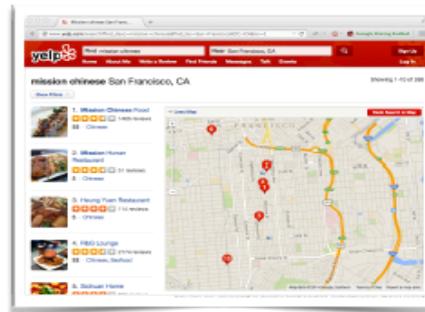
New password:  Password strength: **Strong**

## Third-party mashups



let's get started. **Capital One**  
WEB: CAPITALONE.COM  
1 Find your bank or credit card. User name for your Capital One Credit Card account  
2 Connect it to Mint. Password for your Capital One Credit Card account

## Mashups



## Third-party libraries

## Extensions



# Example: Library



## Third-party libraries

- ◆ Library included using tag
  - `<script src="jquery.js"></script>`
- ◆ No isolation
  - Same frame, same origin as rest of page
- ◆ May contain arbitrary code
  - Library developer error 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

# Second example: 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)

```

```



# Second example: Ad vs Ad

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

INDIANTAGS Home » Register      Sort news by: Recently Popular | [Top Today](#) | [Yesterday](#) | [Week](#) | [Month](#) | [Year](#) |

Published News      Upcoming News

**Register**

Register

Username:  [Verify](#)

Email:

 **\$1 Buy Now**

**What is INDIANTAGS?**

INDIANTAGS is social news submitting site.vote for best [stories](#) [read more](#)

Replay

**Fabulous Festive Offers**



**SHOP NOW @**  
[www.shoppersstop.com](http://www.shoppersstop.com)

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";`

# Third example: Browser Extensions

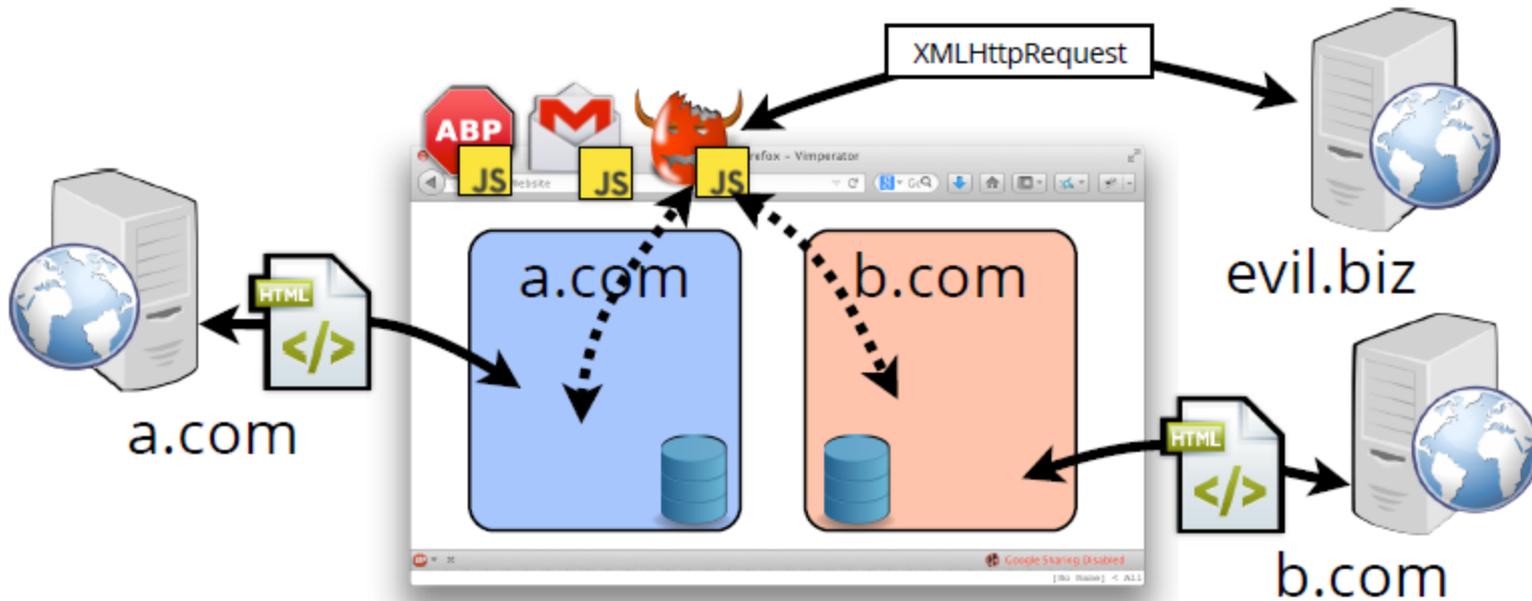
- ◆ Firefox user interface written in JavaScript and XUL, an XML grammar that provides buttons, menus, ...
- ◆ The browser is implemented in a XUL file containing, e.g., this code defining the status bar

```
<statusbar id="status-bar">  
  ... <statusbarpanel>s ...  
</statusbar>
```

- ◆ Extend the browser by inserting new XUL DOM elements into the browser window and modifying them using script and attaching event handlers

# Third example: Browser Extensions

- ◆ Run with privileges of *browser*



# Goal: Password-strength checker

The diagram illustrates a password strength checker interface. It consists of two main components within a container:

- Form (a.com):** Contains a label "New password:" and a text input field with three dots inside, representing a masked password.
- Strength Checker (b.ru/chk.html):** A separate frame that displays the result "Password strength: Strong" in green text, accompanied by a green progress bar.

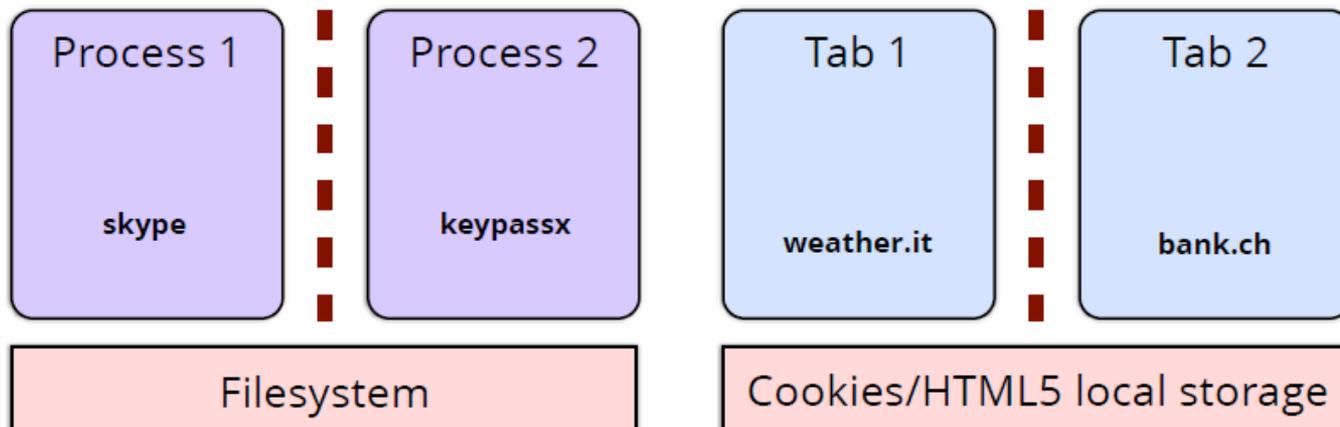
- ◆ Strength checker can run in a separate frame
  - Communicate by `postMessage`
  - But we give password to *untrusted* code!
- ◆ Is there any way to make sure untrusted code does not export our password?

# Modern Structuring Mechanisms

- ◆ HTML5 Web Workers
  - Separate thread; isolated but same origin
- ◆ HTML5 Sandbox
  - Load with unique origin, limited privileges
- ◆ Cross-Origin Resource Sharing (CORS)
  - Relax same-origin restrictions
- ◆ 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>`, ...)



# 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`

```
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
```

main  
thread

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

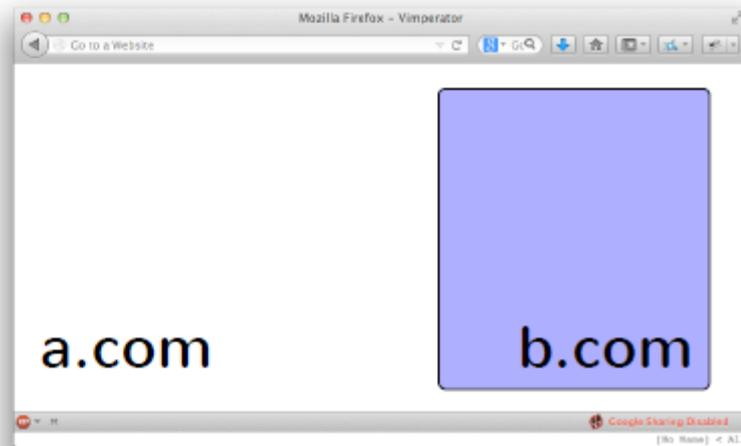
doWork

# Modern Structuring Mechanisms

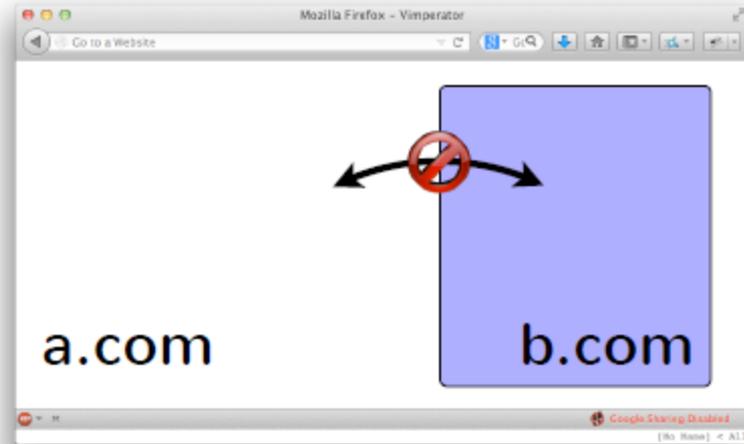
- ◆ HTML5 Web Workers
  - Separate thread; isolated but same origin
- ◆ HTML5 Sandbox
  - Load with unique origin, limited privileges
- ◆ Cross-Origin Resource Sharing (CORS)
  - Relax same-origin restrictions
- ◆ Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources

# Recall Same-Origin Policy (SOP)

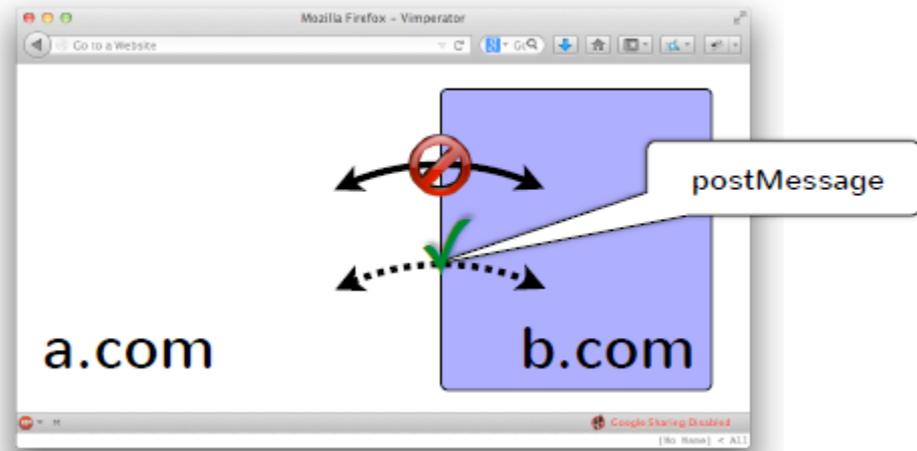
- ◆ Idea: Isolate content from different origins
  - Restricts interaction between compartments
  - Restricts network request and response



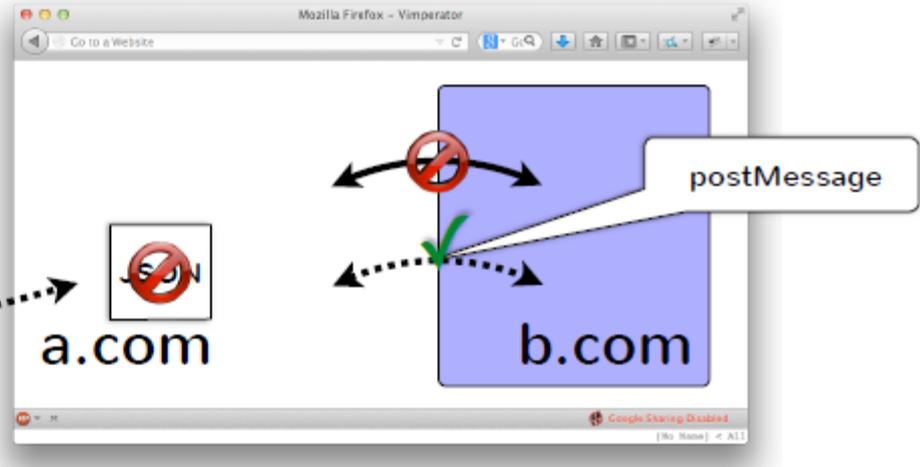
# Recall Same-Origin Policy (SOP)



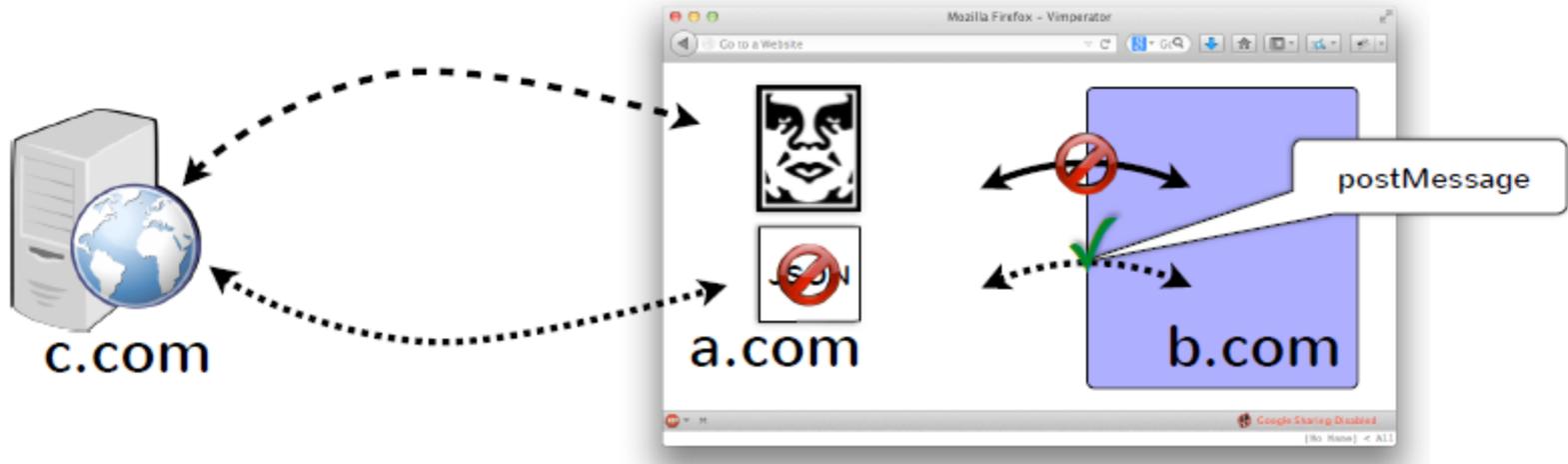
# Recall Same-Origin Policy (SOP)



# Recall Same-Origin Policy (SOP)



# Recall Same-Origin Policy (SOP)



# Same-Origin Policy

## ◆ Limitations:

- Some DOM objects leak data
  - ◆ Image size can leak whether user logged in
- Data exfiltration is trivial
  - ◆ Any XHR request can contain data from page
- Cross-origin scripts run with privilege of page
  - ◆ Injected scripts can corrupt and leak user data!

# Modern Structuring Mechanisms

- ◆ HTML5 Web Workers

- Separate thread; isolated but same origin



- ◆ HTML5 Sandbox

- Load with unique origin, limited privileges

- ◆ Cross-Origin Resource Sharing (CORS)

- Relax same-origin restrictions

- ◆ Content Security Policy (CSP)

- Whitelist instructing browser to only execute or render resources from specific sources

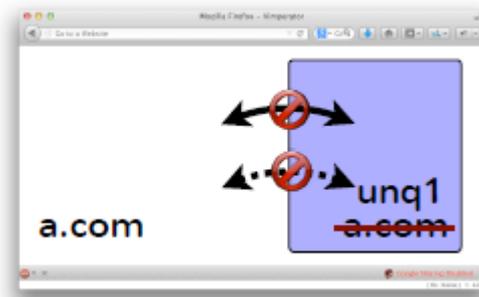
# HTML5 Sandbox

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



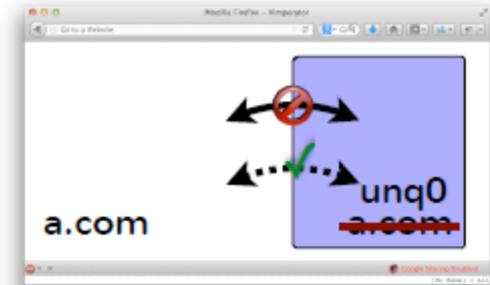
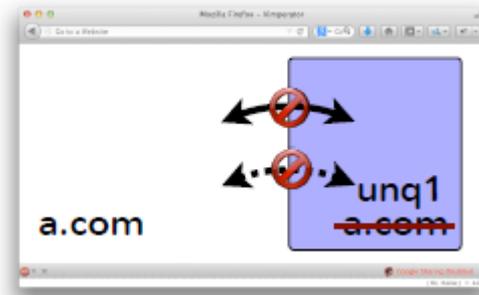
# HTML5 Sandbox

- ◆ **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.

# Modern Structuring Mechanisms

- ◆ HTML5 Web Workers

- Separate thread; isolated but same origin

- ◆ HTML5 Sandbox

- Load with unique origin, limited privileges

- ➔ Cross-Origin Resource Sharing (CORS)

- Relax same-origin restrictions

- ◆ Content Security Policy (CSP)

- Whitelist instructing browser to only execute or render resources from specific sources

# Cross-Origin Resource Sharing (CORS)

- ◆ Idea: Explicitly allow resources to be readable cross-origin



# Modern Structuring Mechanisms

- ◆ HTML5 Web Workers

- Separate thread; isolated but same origin

- ◆ HTML5 Sandbox

- Load with unique origin, limited privileges

- ◆ Cross-Origin Resource Sharing (CORS)

- Relax same-origin restrictions

- ➔ 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 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

# 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`

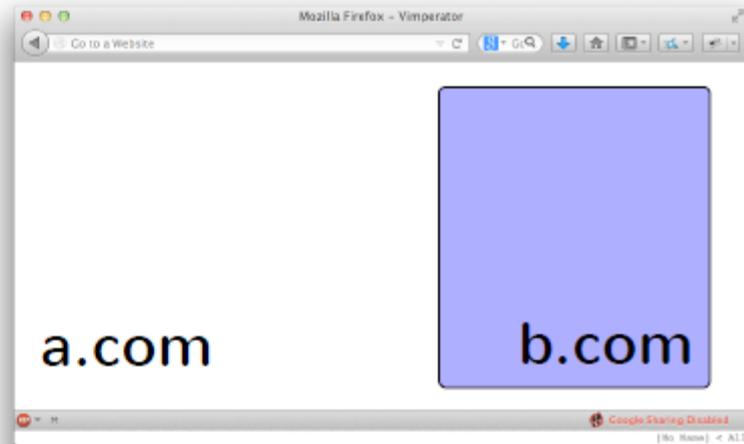
# Content Security Policy (CSP)

- ◆ **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 \*



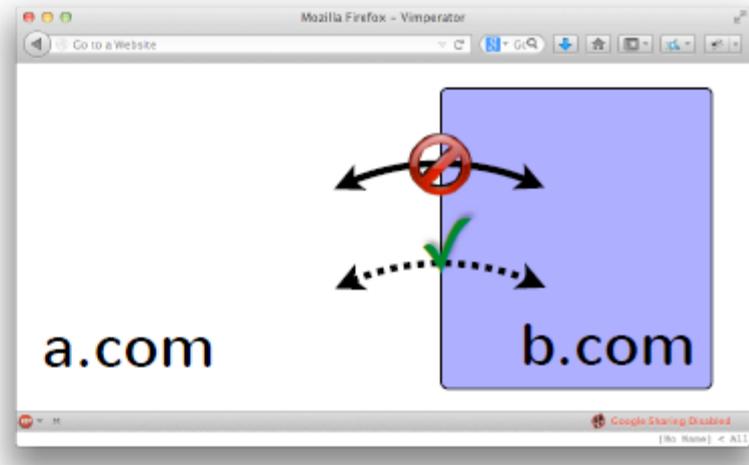
# Content Security Policy (CSP)

- ◆ **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)

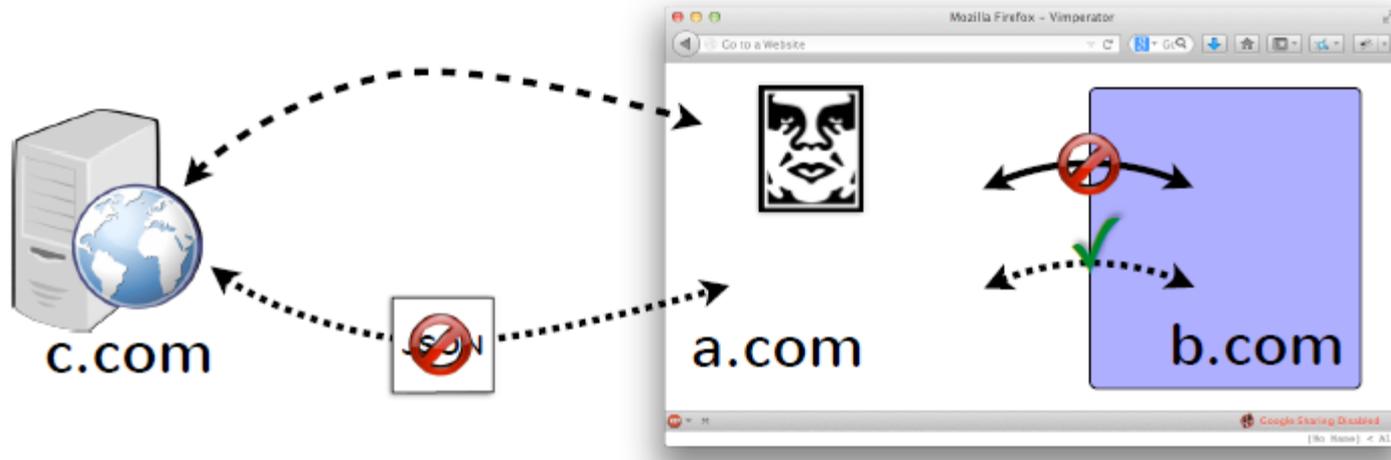
- ◆ **Goal:** prevent and limit damage of XSS attacks
- ◆ **Approach:** restrict resource loading to a white-list
  - E.g., `default-src 'self' http://b.com; img-src *`





# Content Security Policy (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?

# Modern Structuring Mechanisms

- ◆ HTML5 Web Workers
  - Separate thread; isolated but same origin
- ◆ HTML5 Sandbox
  - Load with unique origin, limited privileges
- ◆ Cross-Origin Resource Sharing (CORS)
  - Relax same-origin restrictions
- ◆ Content Security Policy (CSP)
  - Whitelist instructing browser to only execute or render resources from specific sources

# Recall: Password-strength checker

New password:

[Password strength:](#) Strong

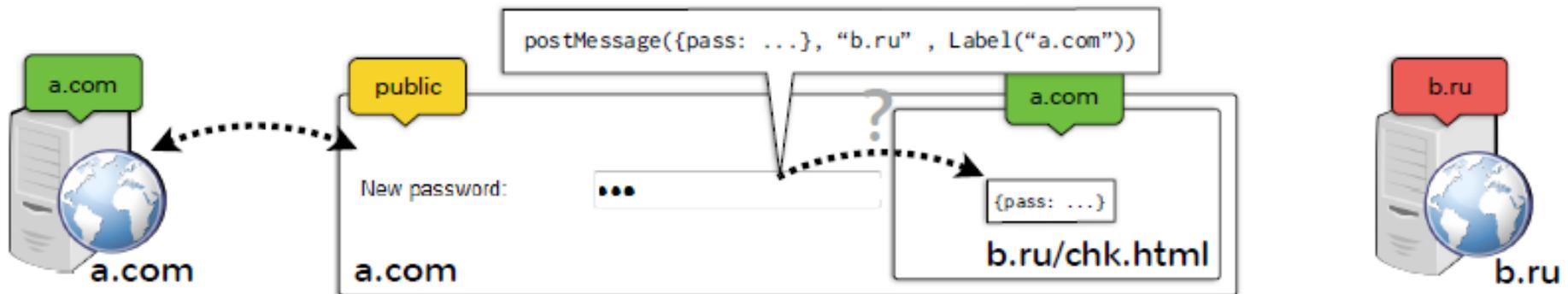
a.com

b.ru/chk.html

- ◆ 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 SWAPI

- ◆ 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



# Modern web site



Code from many sources  
Combined in many ways

# Challenges

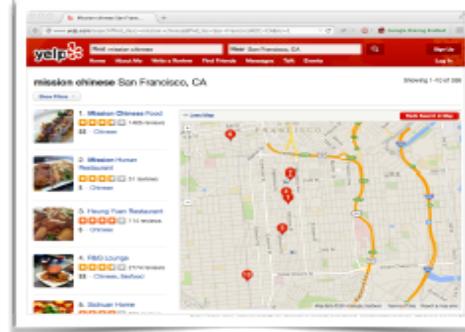
## Third-party APIs



## Third-party mashups



## Mashups



## Extensions



## Third-party libraries