Match the attack to its description:

Attacks:

8 Using Components with Known Vulnerabilities
7 Missing Function Level Access Control
5 Sensitive Data Exposure
6 Security Misconfiguration
4 Insecure Direct Object References
2 Cross Site Scripting
3 Broken Authentication and Session
1 Injection

Descriptions:

1. Modifies back-end statement through user input.
2. Inserts Javascript into trusted sites.
3. Program flaws allow bypass of authentication methods.
4. Attackers modify file names.
5. Abuses lack of data encryption.
6. Exploits misconfigured servers.
7. Privilege functionality is hidden rather than enforced through access controls.
8. Uses unpatched third party components.

Goals of Web Security:

Browse the web safely

- No stolen information
- Site A cannot compromise session at site B

Support secure web applications

- Applications delivered over the web should be able to achieve the same security properties as stand alone applications

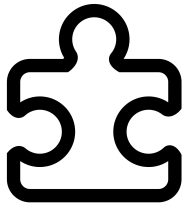
Threat Models

Web Security Threat Model:

- Attacker sets up a malicious site
- Attacker does not control the network

Network Security Threat Model:

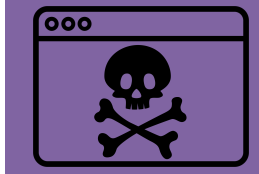
- Attacker intercepts and controls network



Rank these in order, 1 for the most common,
10 for the least common:

5	Security Misconfiguration
4	Insecure Direct Object References
7	Missing Function Level Access Control
6	Sensitive Data Exposure
9	Using Components with Known Vulnerabilities
3	Cross Site Scripting
10	Unvalidated Redirects and Forwards
2	Broken Authentication and Session
1	Injection
8	Cross Site request Forgery

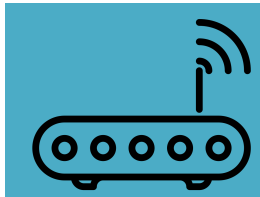
Web Threat Models



Web Attacker:

- Control attacker.com
- Can obtain SSL/TLS certificate for attacker.com
- User visits attacker.com
- Or: runs attacker's Facebook app, etc.

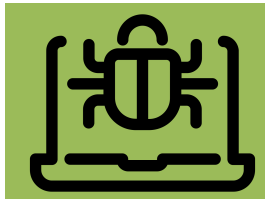
Web Threat Models



Network Attacker:

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

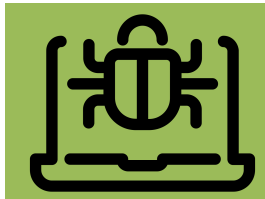
Web Threat Models



Malware Attacker:

- Attacker escapes browser isolation mechanisms and runs separately under control of OS

Web Threat Models



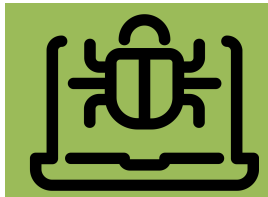
Malware Attacker:

- Browsers may contain exploitable bugs
  - Often enable remote code execution by web sites
- Even if browsers were bug-free, still lots of Vulnerabilities on the web
  - XSS, SQLi, CSRF, …

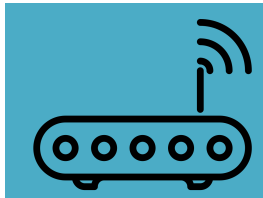
Web Threat Models

Most lethal

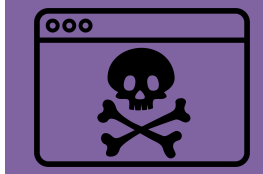
Least lethal



Malware Attacker



Network Attacker



Web Attacker

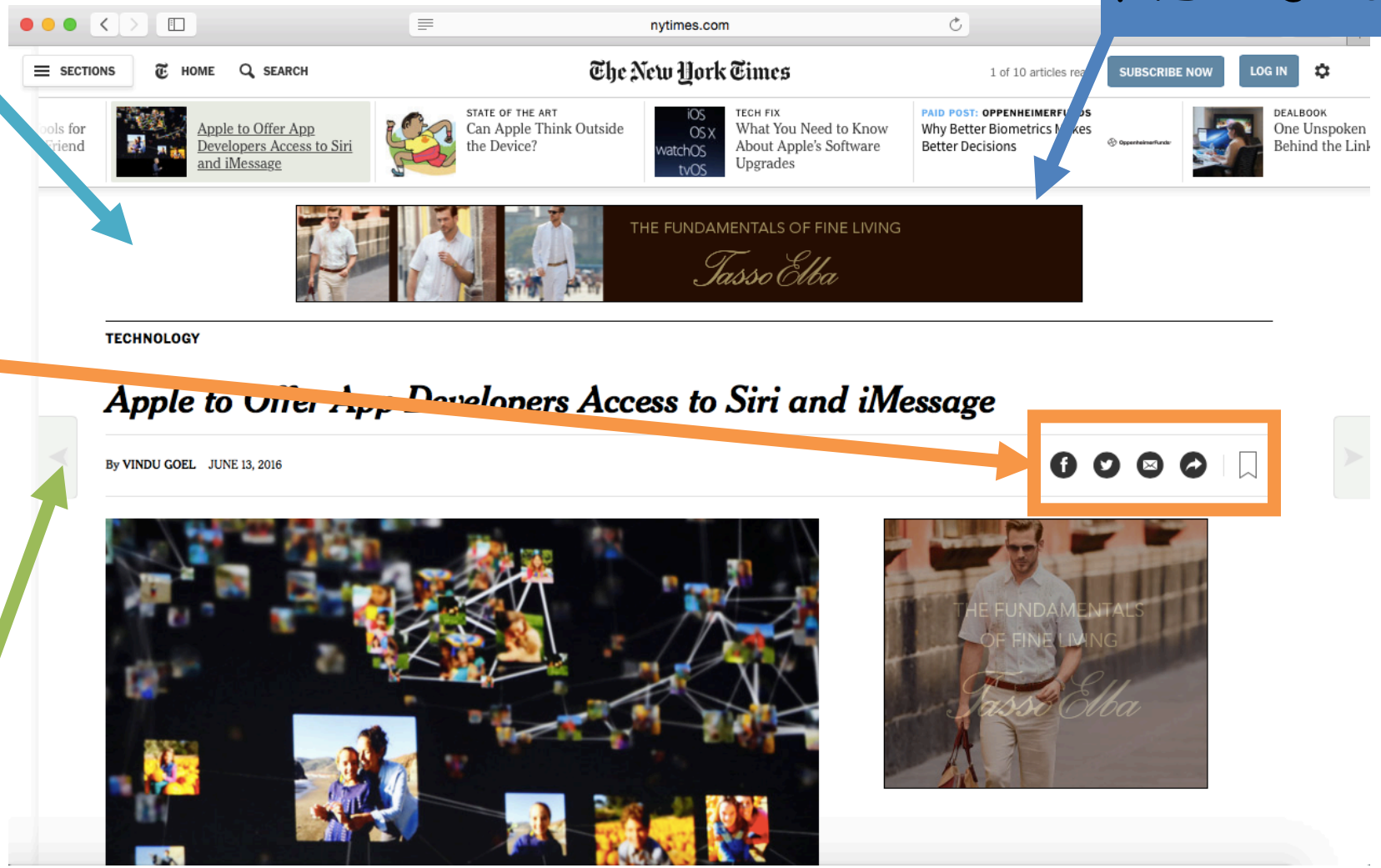
Modern Websites

Page code

Third-party API's

Third-party Libraries

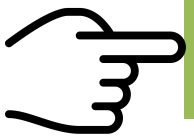
AD code



Acting parties on a website:

- Page developers
- Library developers
- Service providers
- Data providers
- Ad providers
- Other users
- Extension developers
- CDN's

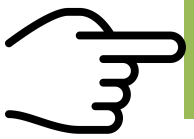
Basic Questions:



Specifically:

- Protecting sensitive information in browser
- Ensuring some form of integrity

Basic Questions:



Specifically:

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

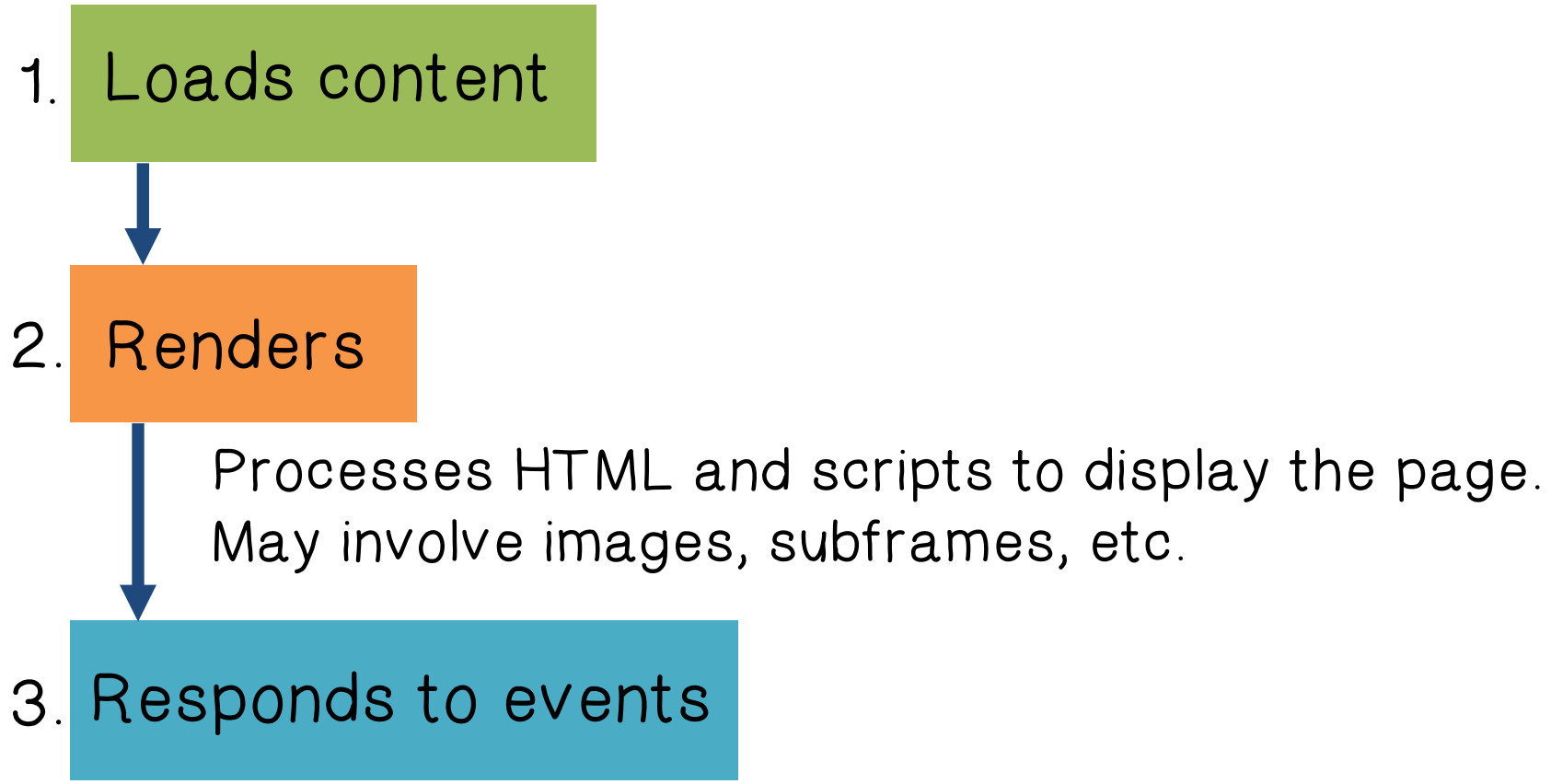
Website Quiz Solution

In 2015 how many active websites were on the internet?	1 billion
How many websites does Google quarantine each DAY?	10,000
How many malicious websites are identified every DAY?	30,000

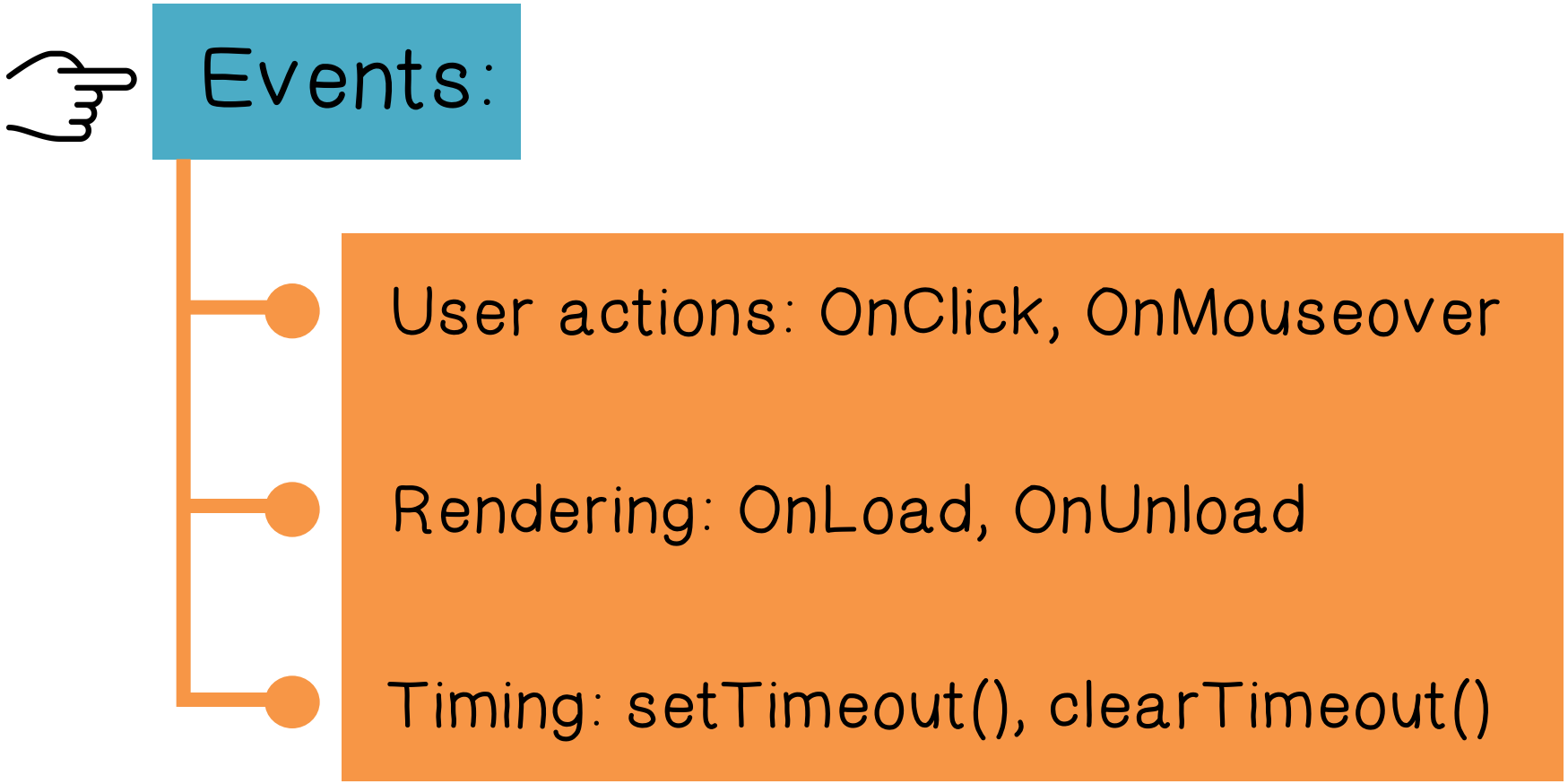
	Operating System	Web Browser
Primitives	Systems calls Processes Disk	Document Object model Frames Cookie/local storage
Principles	Users: Discretionary access Control	Origins: Mandatory Access Control
Vulnerabilities	Buffer Overflow Root Exploit	Cross-scripting Cross-site request forgery Cache history attacks

Basic Execution Model

Each browser window or frame:



Basic Execution Model



Browser content comes from many sources:

Scripts: <script src= “//site.com/script.js”> </script>
Frames: <iframe src= “//site.com/frame.html”> </iframe>
Stylesheets (CSS): <link rel=“stylesheet” type=“text/css”
href=“//site.com/theme.css”/>
Objects (Flash)- using swfobject.js script:

```
<script> var so= new SWFObject(‘//site.com/flash.swf’, ...);
      so.addParam(’allowscriptaccess’, ‘always’);
      so.write(‘flashdiv’);
</script>
```

Browser content comes from many sources:

```
Scripts: <script src= “//site.com/script.js”> </script>
Frames: <iframe src= “//site.com/frame.html”> </iframe>
Stylesheets (CSS): <link rel=“stylesheet” type=“text/css”
                                    href=“//site.com/theme.css”/>
Objects (Flash)- using swfobject.js script:
      <script> var so= new SWFObject(‘//site.com/flash.swf’, ...);
            so.addParam(’allowscriptaccess’, ‘always’);
            so.write(‘flashdiv’);
      </script>
```

Allows Flash object to communicate with external scripts, navigate frames, open windows

Browsers- Sandbox

- **Goal**: Safely execute JavaScript code provided by a remote website. No direct file access, limited access to OS, network, browser data, content that came from other websites
- **Same Origin Policy (SOP)**: Can only read properties of documents and windows from the same protocol, domain and port.
- User can grant privileges to signed scripts: UniversaBrowserRead/Write, UniversaFileRead, UniversalSendMail

Sandbox Quiz Solution

Next to each characteristic, put an S for Sandbox, V for virtual machine, or B for both.

B	Anything changed or created is not visible beyond its boundaries
S	If data is not saved, it is lost when the application closes
V	It is a machine within a machine
S	Lightweight and easy to setup
V	Disk space must be allocated to the application

Browser Same Origin Policy

protocol://domain:port/path?params

- Same Origin Policy (SOP) for DOM:
  - -Origin A can access origin B's DOM if A and B have same (protocol, domain, port)
- Same Origin Policy (SOP) for cookies:
  - -Generally, based on ([protocol], domain, path)
    - protocol is optional

Frame Security

Windows may contain frames from different sources:

Frame

Rigid division as part of frameset

iFrame

floating inline frame

Frame Security

iFrame example:

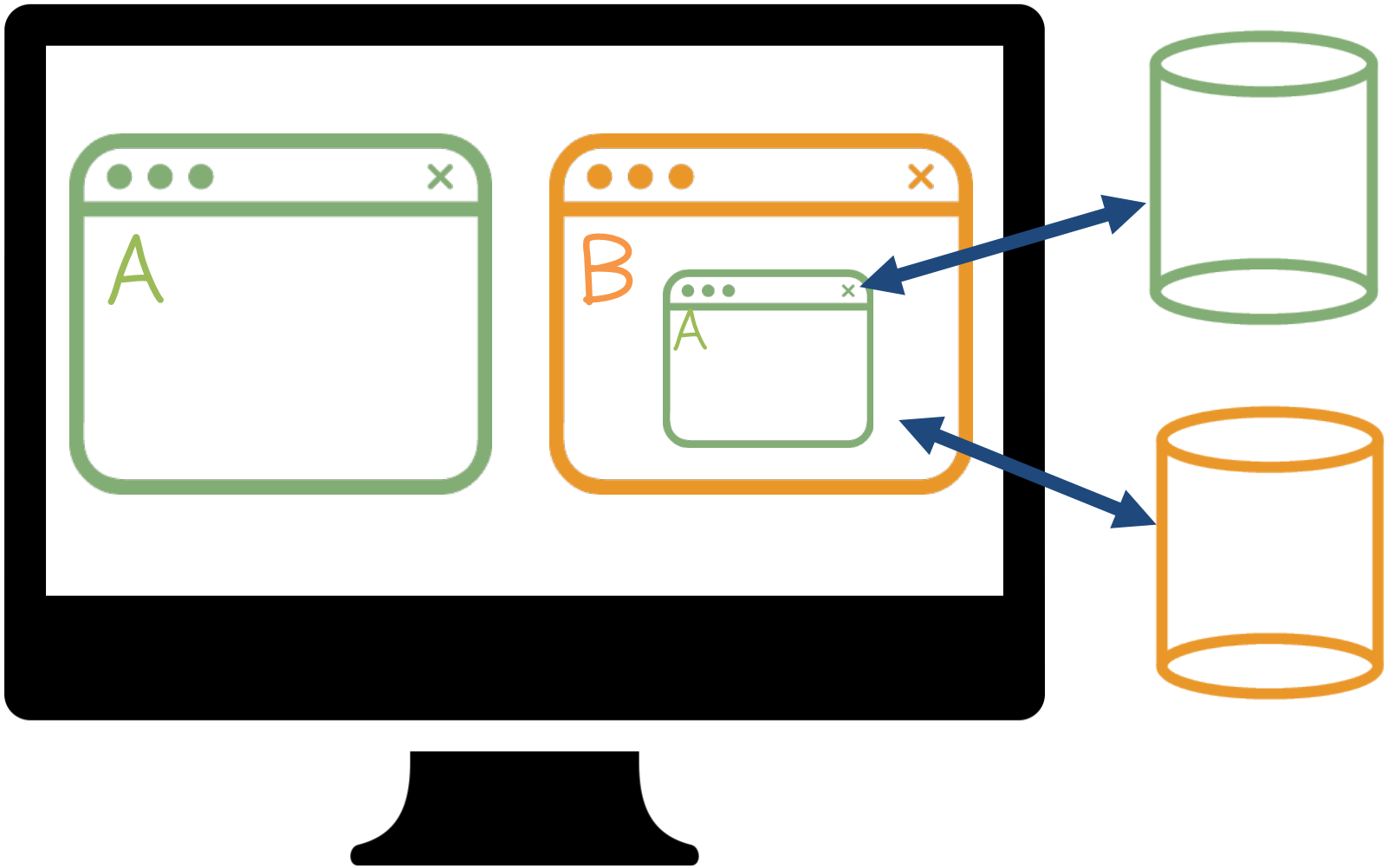
```
<iFrame src=‘hello.html” width=“450”height=“100”>
</iFrame>
```

Frame Security

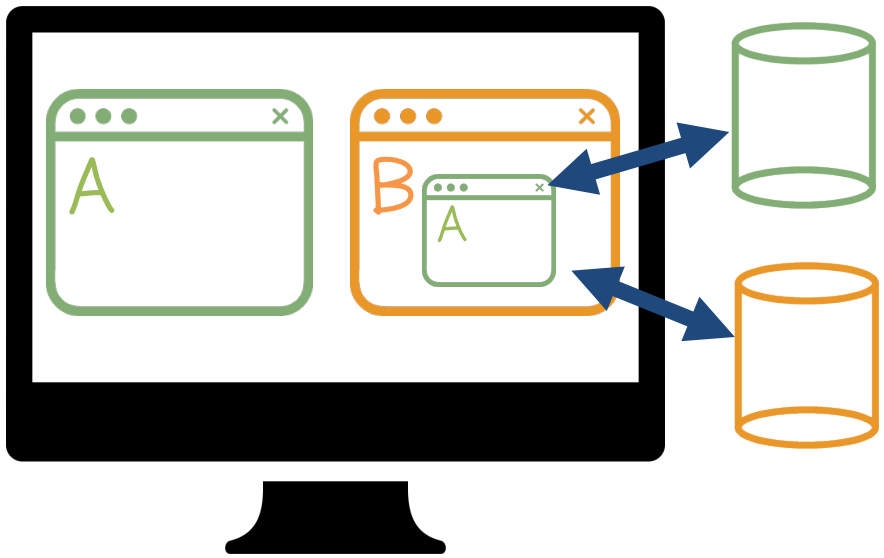
Why use frames?

- Delegate screen area to content from another source
- Browser provides isolation based on frames
- Parent may work even if frame is broken

Frame Security



Frame Security



- Each frame of a page has an origin
  Origin= protocol://host:port
- Frame can access its own origin
  Network access, Read/write
  DOM, Storage (cookies)
- Frame cannot access data associated with a different origin

Frame Security

Frame-Frame Relationships:

- canScript(A,B)

  Can Frame A execute a script that manipulates arbitrary/nontrivial DOM elements of Frame B?

- canNavigate(A,B)

  Can Frame A change the origin of content for Frame B?

Frame Security

Frame-Principle Relationships:

- readCookie(A,S), writeCookie(A,S)
  - Can Frame A read/write cookies from site S?

See: https://code.google.com/p/browsersec/wiki/Part 1
https://code.google.com/p/browsersec/wiki/Part 2

Browsing Context

A browsing context may be:

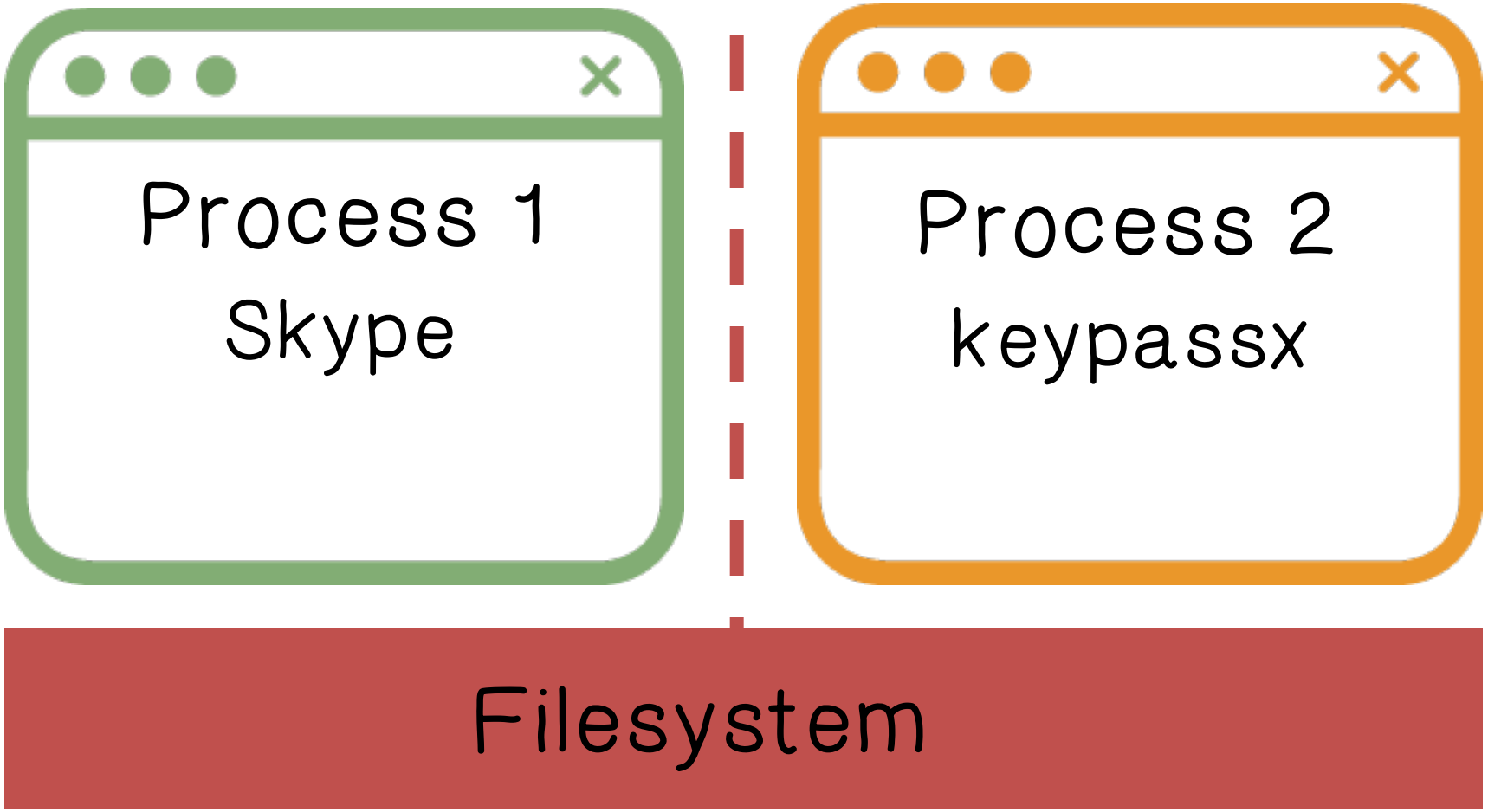
- A frame with its DOM
- A web worker (thread), which does not have a DOM

Browsing Context

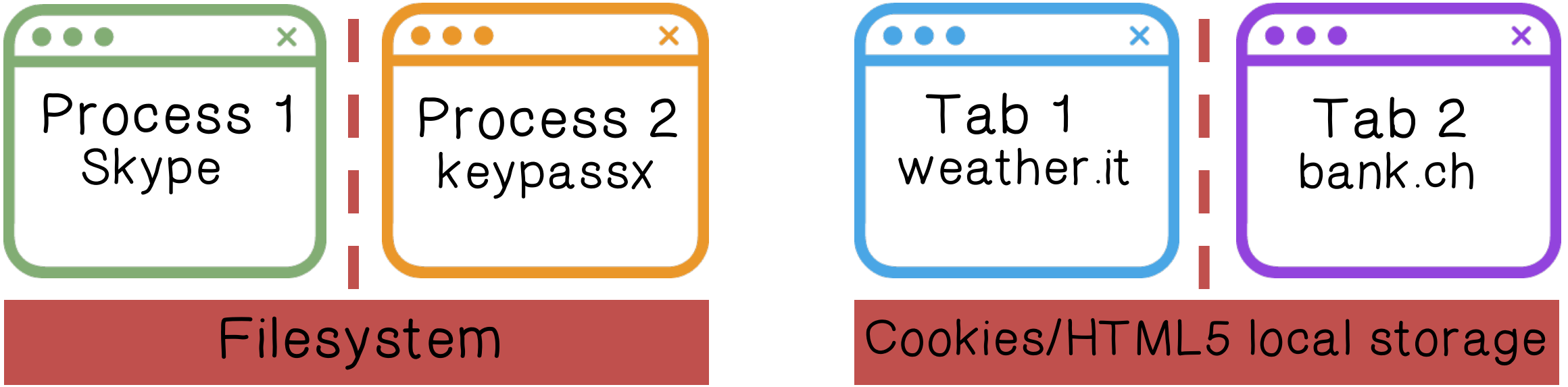
Every browsing context:

- Has an origin, determined by protocol, host, port
- Is isolated from other by same-origin policy
- May communicate to others using postMessage
- Can make network requests using XHR or tags (<image>,...)

OS Process Context



Comparing Process Context and Browsing Context



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.
- Cross-Origin Resource Sharing (CORS) Relax same-origin restrictions
- HTML5 Web Workers Separate thread; isolated but same origin.
  Not originally intended for security, but helps.
- SubResource integrity (SRI)

HTML Sandbox

Directive:	Outcome:
Sandbox	Ensures iframe has unique origin and cannot execute JavaScript, no form submission, disable API's, prevent content from using plugins, etc.
Sandbox allow-scripts	Ensures iframe has unique origin.

Modern Structuring Mechanisms
Sandbox example

Twitter button in iframe:

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

Modern Structuring Mechanisms

Sandbox: remove all permissions and then allow JavaScript, popups, form submission

```
<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

Sandbox Quiz Solution

Given the list of attributes, which 2 should not be combined?
Put a check next to the 2 attributes that should not be combined.

- [ ] `allow-forms`
- [ ] `allow-popups`
- [ ] `allow-pointer-lock`
- [x] `allow-same-origin`
- [x] `allow-scripts`
- [ ] `allow-top-navigation`

Content Security Policy

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

Content Security Policy

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.

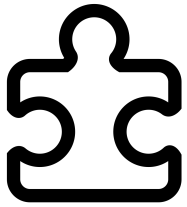
Directive	Outcome
script-src	limits the origins for loading scripts
connect-src	limits the origins to which you can connect (via XHR, WebSockets, and EventSource)
font-src	specifies the origins that can serve web fonts
frame-src	lists origins can be embedded as frames
img-src	lists origins from which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

CSP Source Lists

- 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-Java Script mechanisms like eval



CSP Quiz Solution

Which of the following statements are true?

- [ ] If you have third party forum software that has inline script, CSP cannot be used
- [x] CSP will allow third party widgets (e.g. Google +1 button) to be embedded on your site.
- [ ] For a really secure site, start with allowing everything, then restrict once you know which sources will be used on your site.

Web Worker

Run in an isolated thread, loaded from a separate file:

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

Web 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);
```

SubResource Integrity

Many pages pull scripts and styles from a wide variety of service and content delivery networks.

SubResource Integrity

How can we protect against:

- Downloading content from a hostile server (via DNS poisoning, or other such means)
- Modified file on the Content Delivery Network (CDN)

SubResource Integrity

Idea:

page author specifies has a (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">
```

SubResource Integrity

Idea:

page author specifies has a (sub) resource they are loading; browser checks integrity.

E.G., integrity for elements:

```
<script src= “https://code.jquery.com/jquery-1.10.2.min.
js” integrity= “sha256-
C6CB9UYIS9UJewinPHWTHVqh/E1uhG5Tw+Y5qFQmYg=“>
```

SubResource Integrity:

Case 1 (default)

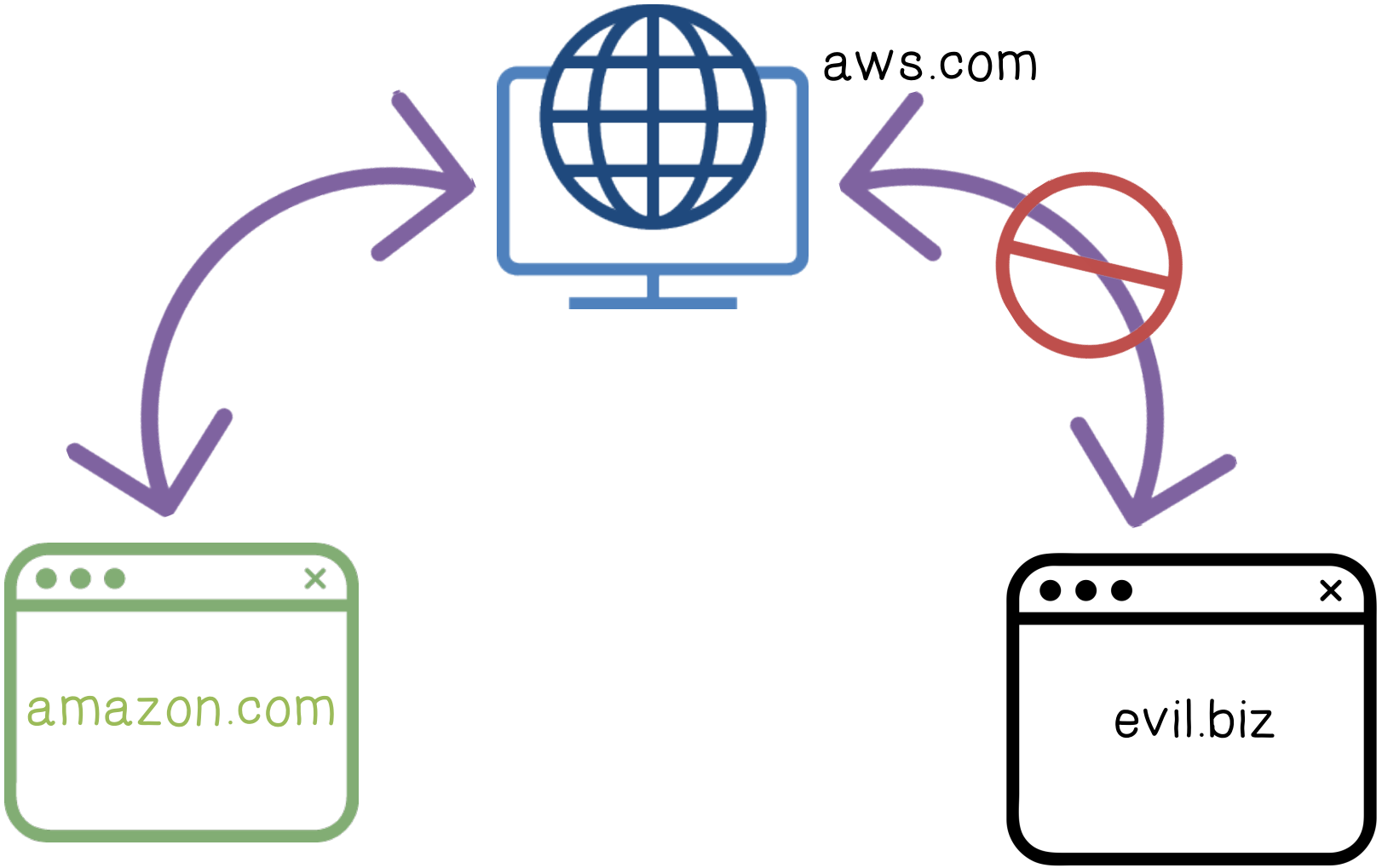
Browser reports violation and does not render/execute resource

Case 2

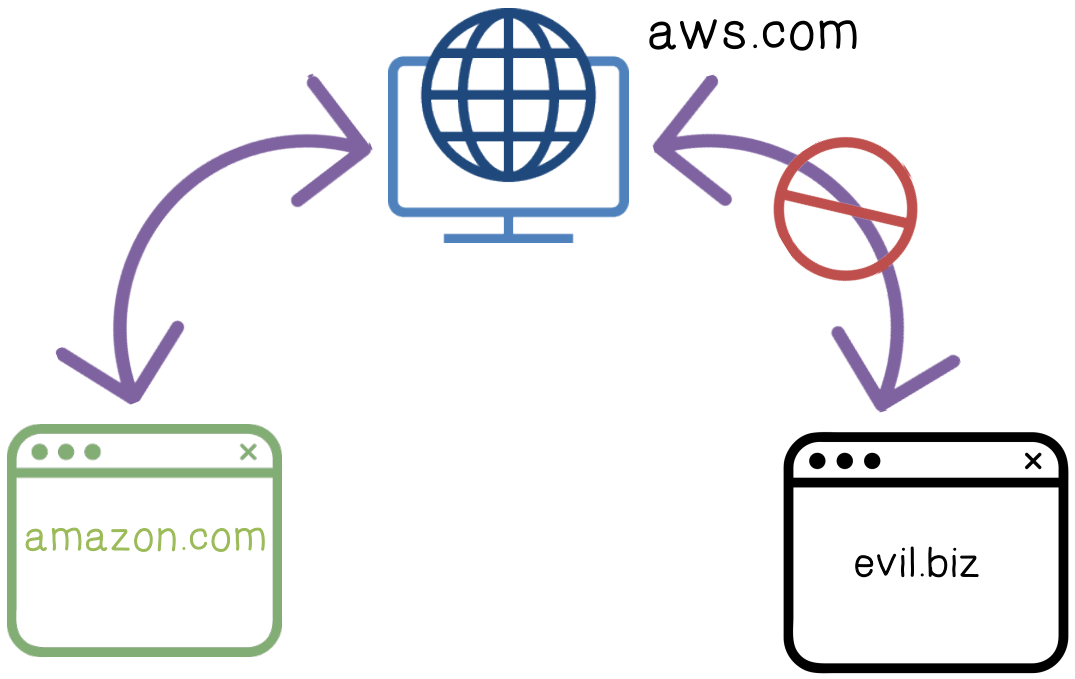
CSP directive with integrity-policy directive set to report

Browser reports violation, but may render/execute resource

Cross Origin Resource Sharing



Cross Origin Resource Sharing



- A technique for relaxing the same-origin policy, allowing JavaScript on a web page to consume content from a different origin.
- A website whitelists all domains

How CORS Works

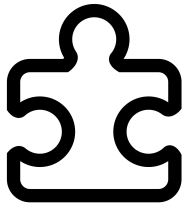
- Browser sends Origin header with XmlHttpRequest request

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: *



CORS Quiz Solution

Select all the statements that are true:

- [x] CORS allows cross-domain communication from the browser
- [x] CORS requires coordination between the server and client
- [ ] CORS is not widely supported by browsers
- [ ] The CORS header can be used to secure resources on a website