EECS 388

Introduction to Computer Security

Lecture 7:

The Web Platform

September 17, 2024
Prof. Chen

Web Security

This week:

- **The Web Platform**
- Web Attacks and Defenses

Next week:

- HTTPS and the Web PKI
- HTTPS Pitfalls

Later in the course:

- User Authentication
- Privacy and Online Tracking

Web Platform Security

The **Web platform** is the collection of technologies developed as **open standards** that powers Web sites and applications.

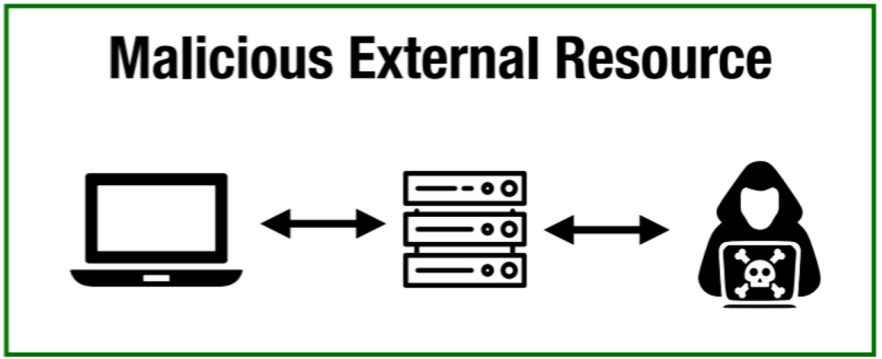
Open: *Anyone* can build a Web browser or server, participate in standard design.
Standards: **URLs, HTML**, **JavaScript**, **HTTP**, **TLS**, etc.

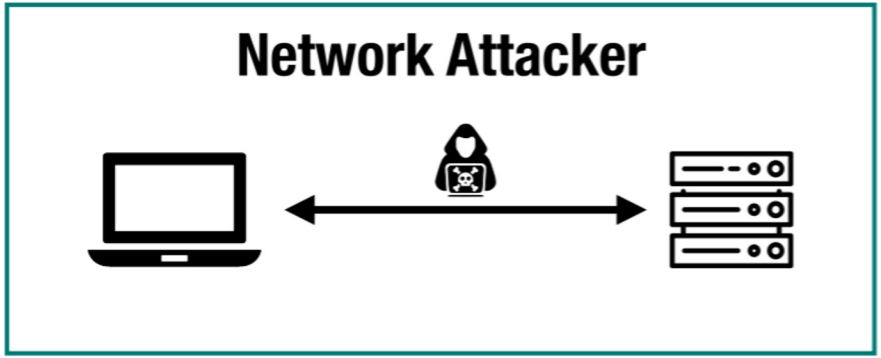
Web security goals:

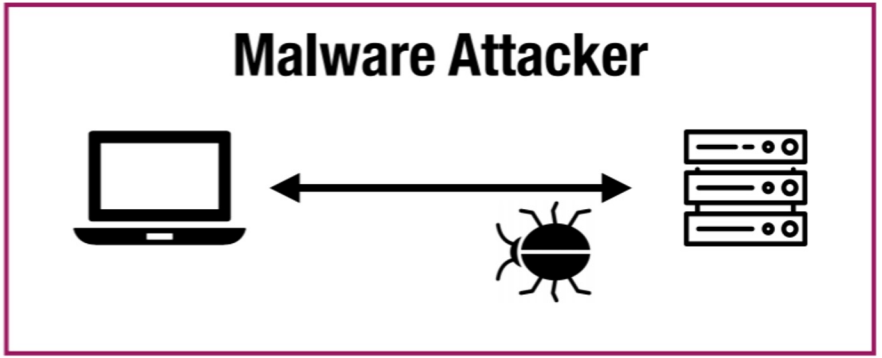
- **Protect users from malicious sites and networks**
- **Isolate sites from each other within the browser:**

  Integrity: Site A cannot **affect user's session** on Site B
  Confidentiality: Site A cannot **steal user's data** from Site B

Attack Models





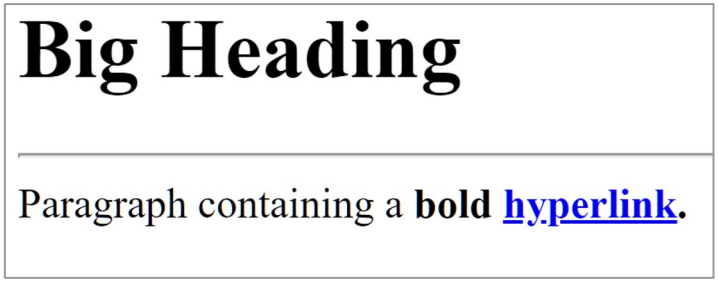


HTML Basics

HTML (Hypertext Markup Language) is a standard language for documents displayed in a browser. Defines document structure and hints at appearance.

Built from nested **elements** specified using **<tags>**, which may have **attributes=""**.

```
<html>
  <!-- comment -->
  <head><title>Page Title</title></head>
  <body>
    <h1>Big Heading</h1>
    <hr/><!-- horizontal rule (self-closing tag) -->
    <p>
      Paragraph containing a <b>bold <a href="https://example.com/">hyperlink</a>.</b>
    </p>
  </body>
</html>
```



Caution: Some literal characters must be **escaped** or they may be misinterpreted as part of the HTML markup.

Replace `<` `>` `&` `"` `'` with `<` `>` `&` `&quot;` `&apos;`

JavaScript and the DOM

JavaScript running in the browser can read and modify page content. The **document object model (DOM)** and other APIs provide a standard programming interface.

```
<html><body>
  <input id="textbox"/><button id="add">+</button>
  <ul id="todo"/>
  <script>// embedded JavaScript:
    function addTodo(text) {
      var newitem = document.createElement('li');
      newitem.appendChild(document.createTextNode(text));
      document.getElementById('todo').appendChild(newitem);
    }
    document.getElementById('add').onclick = function(e) {
      addTodo(document.getElementById('textbox').value);
    }
  </script>
</body></html>
```

Pages can also include scripts loaded from a separate URL (called a **subresource**):

```
<script src="https://code.jquery.com/jquery-3.4.1.js"
 integrity="sha256-WpOohJOqMqqyKL9FccASB9O0KwACQJpFTUBLTYOVvVU="></script>
```

The hash is a security mechanism called **subresource integrity**. *[Why use it?]*

URLs (Uniform Resource Locators)

A **URL** is a string specifying a unique resource on the Web.

`scheme://host:port/path?query#fragment`

scheme: protocol used to access resource
- `https` (HTTP with end-to-end encryption)
- `http` (plaintext HTTP — unsafe!)

host: server's domain name or IP address
- `eecs388.org` (DNS name)
- `141.212.118.72` (IPv4 address)
- `[2607:f018:600:8::20]` (IPv6 address)

port: TCP port (default 443 for HTTPS and 80 for HTTP)

path: identifies resource to server
- `/papers/index.html` (format up to server)

query: parameter string passed to server
- `?key1=value1&key2=value2` (key-value pairs)
- `?7265de6a2c4c8068ed3e75` (arbitrary string)

fragment: parameter string visible only to client
- `#Section4` (tells browser to scroll to named location)

Paths, Queries, Fragments: % Encoding

Most punctuation and non-ASCII characters must be **escaped** by encoding each UTF-8 byte as % followed by two hex characters.

" " → "%20" "<" → "%3C" ">" → "%3E"

https://google.com/?q=hello%20world

Hosts: IDN Encoding

The host field uses a different encoding for internationalized names called **IDN encoding**.

Portions of the name are replaced by `xn--` followed by a Punycode encoding, e.g.:

https://xn--ls8h.la/

HTTP Protocol

Hypertext Transport Protocol (HTTP) allows fetching individual resources, such as HTML documents.

Structured as a sequence of **requests** and **responses** (not as a stream of data).

Client sends:

- **Method**
  GET (retrieve data; shouldn't change server state)
  POST (can submit data; causes change or side-effect)
- **Path and query**
- Headers

Server returns:

- **Response code**
  200 OK, 302 Redirect, 404 Not Found, ...
- Headers
- **Content data** (arbitrary bytes)

User follows a link from Google to https://example.com/index.html

HTTP 1.1 Request

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0)
Referer: https://www.google.com/
```

HTTP 1.1 Response

```
HTTP/1.1 200 OK
Server: Nginx
Content-Type: text/html
Last-Modified: Fri, 13 Sep 2019 14:27:25 GMT
Set-Cookie: …
Content-Length: 13429

<html><body> …
```

HTTP Cookies

A **cookie** is a piece of data that a server sends to the browser. The browser stores it and returns it in later requests to the same server.

Used for:

- **Maintaining session state** across HTTP requests (logins, shopping carts, etc.)
- **Personalization** (storing preferences)
- **Tracking** user behavior on or across sites

Can also be read and set by JavaScript on page.

Clients can *read*, *change*, or *erase* cookie data!

Two general approaches for using them securely:

1. Store cryptographically protected data.
2. Use a unique, random, unguessable session identifier that's tied to a server-side database.

Initial HTTP Request

HTTP Response (server sets cookie)

```
HTTP/1.1 200 OK
Server: Nginx
Content-Type: text/html
Set-Cookie: trackingID=6bb4ad8baf953b6f;
      Domain=reddit.com; Path=/; Secure
```

Later HTTP Requests (brower returns cookie)

```
GET /r/uofm HTTP/1.1
Cookie: trackingID=6bb4ad8baf953b6f
User-Agent: Mozilla/5.0 (Windows NT 10.0)
```

Browser Execution Model

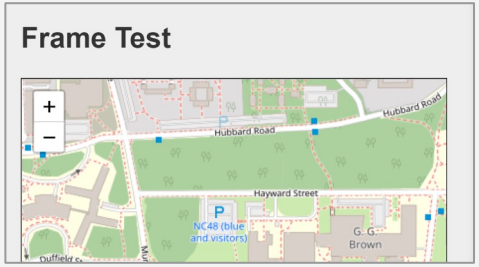
When **loading a document**, the browser:

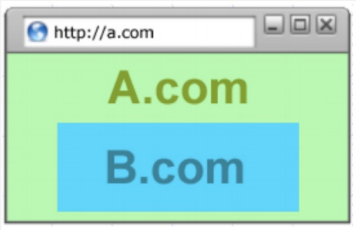
1. Loads content at URL
2. Parses HTML and runs any inline JavaScript code
3. Recursively fetches and renders subresources (JavaScript, images, CSS, frames)

After loading:

Calls JavaScript functions in response to user inputs, timeouts, other events

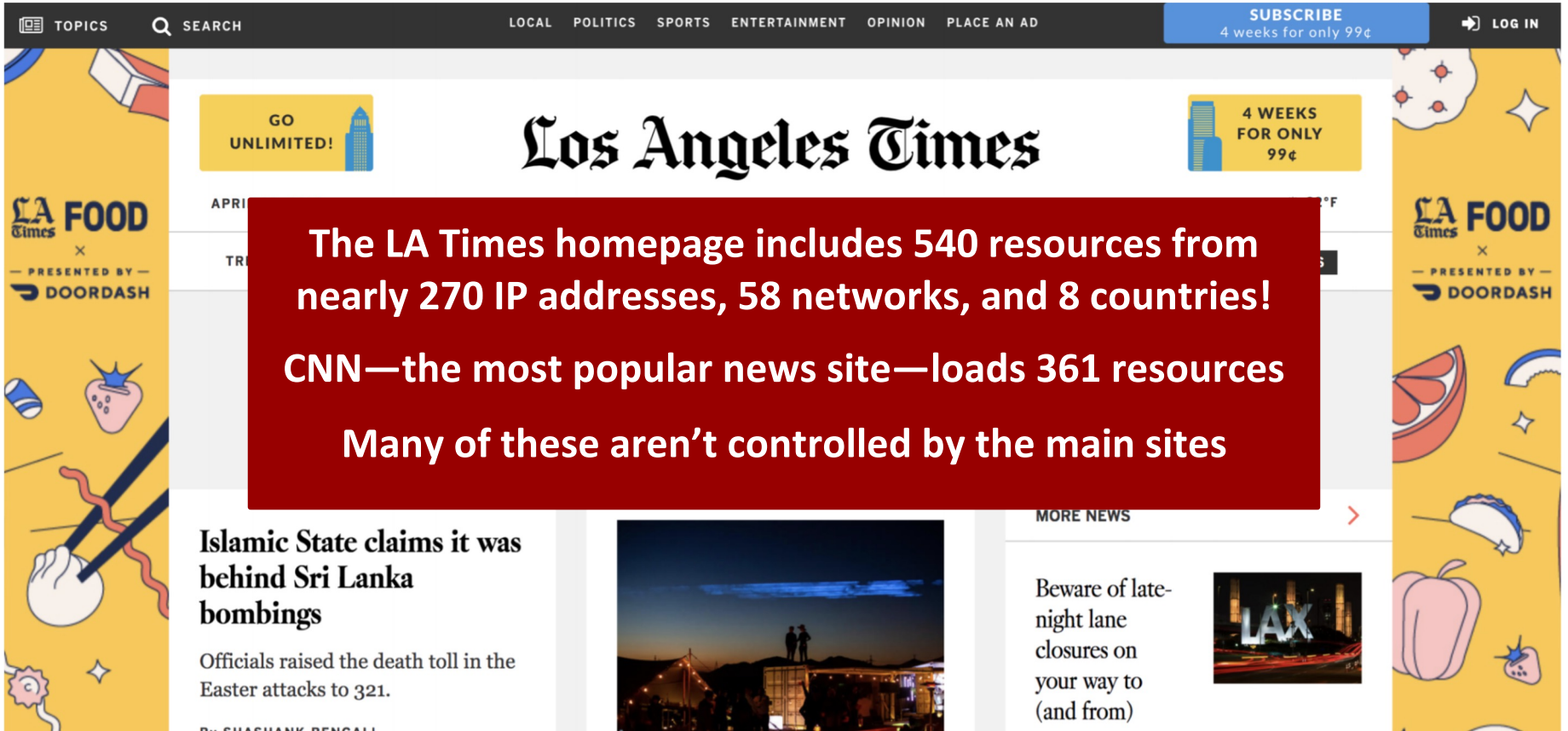
Document can also include **frames**, which display another HTML page. Browsers isolate frames from the parent document.



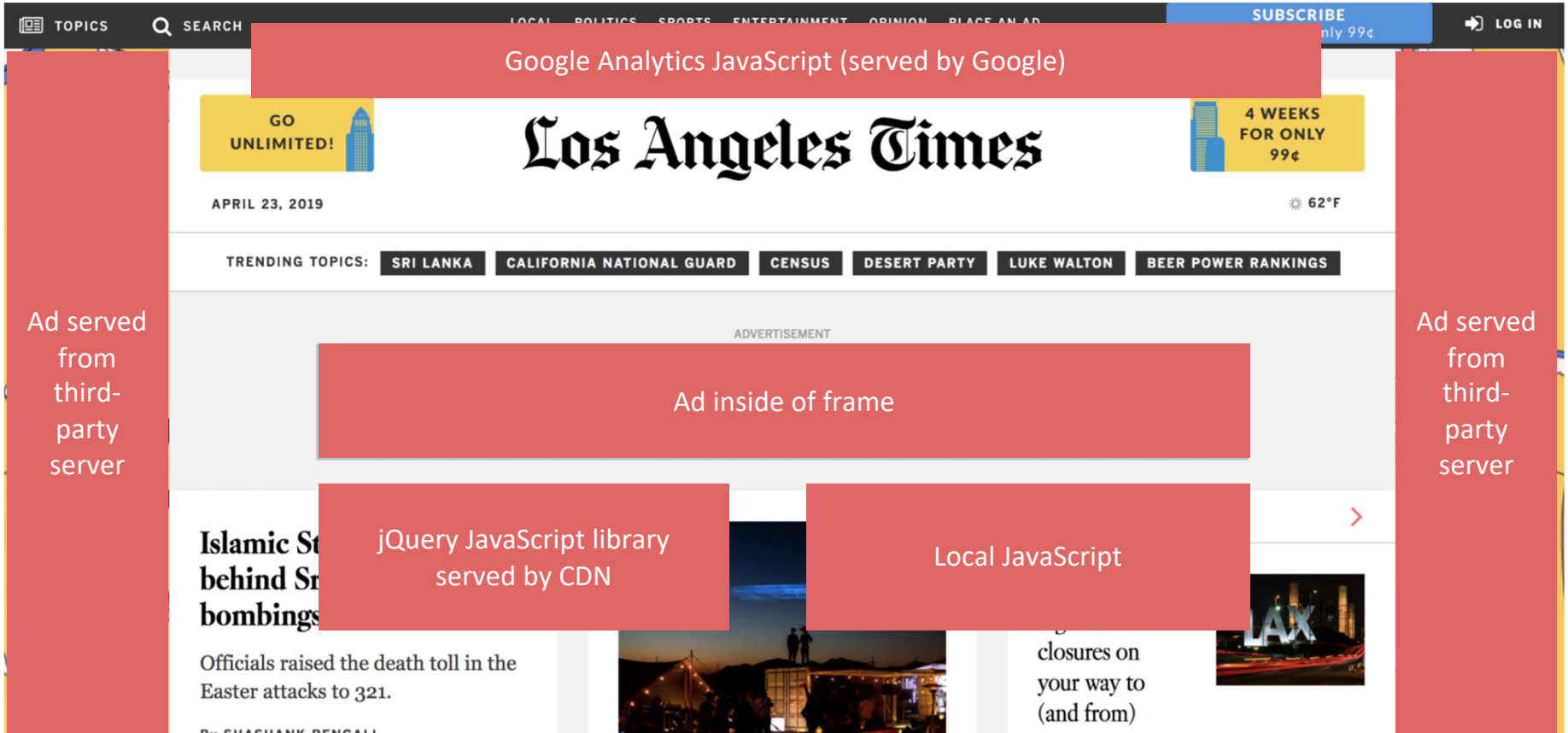



```
<h1>Frame Test</h1>
<iframe width="425" height="350"
src="https://www.openstreetmap.org/export
/embed.html?bbox=-
83.71982216835023%2C42.291433288630564%2C
-
83.71388375759126%2C42.294587952588785"></
/iframe>
```

A Modern Website



A Modern Website



Same-Origin Policy

Essential security question:

When can one site access data contained in another site?

Example:

If you visit **attacker.com**, what stops it from reading your Gmail messages?

What if **attacker.com** loads Gmail in a frame or runs JavaScript files from **gmail.com**?

Browsers enforce isolation between sites by applying **Same-Origin Policy** (**SOP**).

The SOP separates content into different trust domains ("**origins**") and restricts data flows between them.

What defines an origin?

scheme://domain:port

example: **https://eecs388.org:443**

What's isolated?

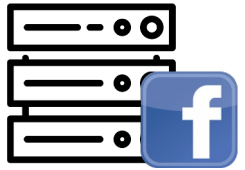
Each origin has local client-side resources that are protected:

- Cookies (local state)
- DOM storage
- DOM tree
- JavaScript namespace
- Permission to use local hardware (e.g., camera or GPS)

SOP: Cross-Origin Image

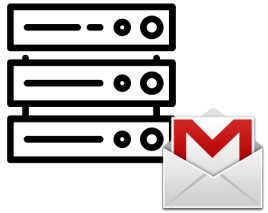
GET / HTTP/1.1
Host: facebook.com

facebook.com



HTTP/1.1 200 OK
…

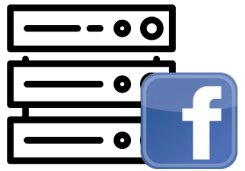
gmail.com



SOP: Cross-Origin Image

GET / HTTP/1.1
Host: facebook.com

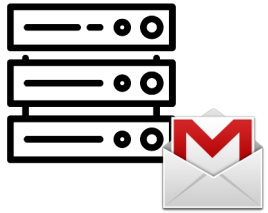
facebook.com



HTTP/1.1 200 OK
…

GET /img.png HTTP/1.1
Host: gmail.com

gmail.com



HTTP/1.1 200 OK
…
<89>PNG^M …

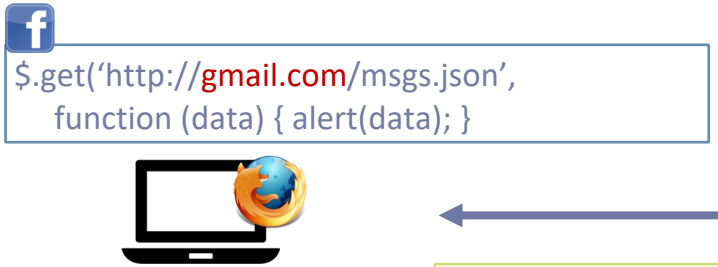
Cross-Origin Images are Allowed
Show on screen, but script can't read the pixels with <canvas>.

SOP: Fetching Cross-Origin Data with JS

GET / HTTP/1.1
Host: facebook.com

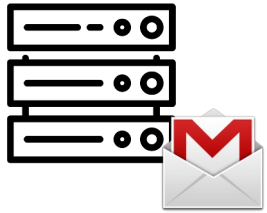
(evil!)
facebook.com

HTTP/1.1 200 OK
…
<script>
$.get('http://gmail.com/msgs.json',
function (data) { alert(data); }
</script>



GET /msgs.json HTTP/1.1
Host: gmail.com

gmail.com



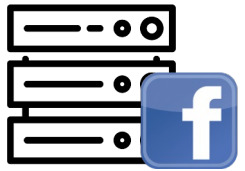
Blocked by SOP
facebook.com JS can't read data from gmail.com

HTTP/1.1 200 OK
…
{ new_msgs: 3 }

SOP: Fetching Cross-Origin Data with JS

GET / HTTP/1.1
Host: facebook.com

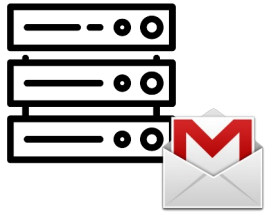
facebook.com



HTTP/1.1 200 OK
…
<script src="http://gmail.com/chat.js"/>

?

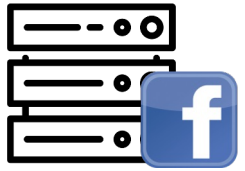
gmail.com



SOP: Fetching Cross-Origin Data with JS

GET / HTTP/1.1
Host: facebook.com

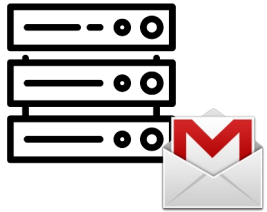
facebook.com



HTTP/1.1 200 OK
…
<script src="http://gmail.com/chat.js"/>

GET /chat.js HTTP/1.1
Host: gmail.com

gmail.com



HTTP/1.1 200 OK
…
$.get('http://gmail.com/chat.json', function(data){ … });

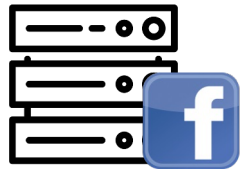
GET /chat.json HTTP/1.1
Host: gmail.com

Blocked by SOP
Script came from gmail.com but the origin is still facebook.com

SOP: A Public API with CORS

GET / HTTP/1.1
Host: facebook.com

facebook.com



$.get('http://gmail.com/chat.json',
function (data) { alert(data); }

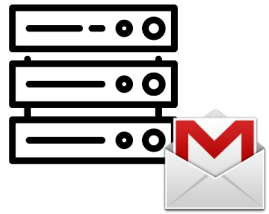
OPTIONS /chat.json HTTP/1.1
Host: gmail.com
Origin: https://facebook.com

Cross-Origin Resource Sharing (CORS) "Preflight" request

Server "opts out" of SOP enforcement for this resource

HTTP/1.1 200 OK
Access-Control-Allow-Origin: *

gmail.com



Allowed via CORS
Server has granted special permission to bypass SOP

GET /chat.json HTTP/1.1
Origin: https://facebook.com

SOP: Cookies

Cookie Scope

Sending cookies only to the right websites is really important!

Frequent source of problems:

Cookies use a different scope than DOM

DOM:

Scoped based on (scheme, host, port)

Cookies:

Scoped based on ([scheme], domain*)

* Complicated rules shown in later slides

Cookie Security Attributes

Weakness: By default, cookies set over HTTPS will also be sent on requests over HTTP (and visible to network eavesdroppers).

Solution: Server can set "`Secure`" attribute to limit cookie to HTTPS requests.

```
Set-Cookie: id=a3fWa; Secure;
```

Weakness: By default, cookies can be read by any JavaScript running in the origin.
E.g., if **bank.com** includes Google Analytics script, Google can read authentication cookies.

Solution: Server can set "`HttpOnly`" attribute to prevent cookie from being accessed by DOM.

```
Set-Cookie: id=a3fWa; Secure; HttpOnly;
```

Rules for Setting and Reading Cookies

Setting a cookie:

A site can set a cookie for its own domain or any *parent domain*, as long as the parent domain is not a **public suffix**.

Example: **login.site.com** attempts to set cookie

login.site.com	allowed
site.com	allowed
other.site.com	prohibited
different.com	prohibited
com	
prohibited (public suffix)	

Caution: You don't know which domain set a cookie when you receive it.

Example: **club.eecs.umich.edu** can set cookies for **umich.edu** (since latter isn't a public suffix).

Rules for Setting and Reading Cookies

Setting a cookie:

A site can set a cookie for its own domain or any *parent domain*, as long as the parent domain is not a **public suffix**.

Example: **login.site.com** attempts to set cookie

login.site.com	allowed
site.com	allowed
other.site.com	prohibited
different.com	prohibited
com	prohibited (public suffix)

Caution: You don't know which domain set a cookie when you receive it.

Example: **club.eecs.umich.edu** can set cookies for **umich.edu** (since latter isn't a public suffix).

Reading a cookie:

A site can read cookies set for its own domain **or any parent domains**.*

Example: **login.site.com** can read cookies for

login.site.com	yes
site.com	yes
other.site.com	no
different.com	no

Caution: Cookies also specify a *path* on the site, but (without `HttpOnly`) it's for efficiency only. DOM origins aren't isolated by path, so scripts can read cookies set for any path in the origin.

Suppose cookie set for **x.com/b**. Then **x.com/a** can do: `alert(frames[0].document.cookie);`

* If **domain=** attribute is *unset*, some browsers disallow reading by subdomains, but can't rely on this behavior.

When are these cookies sent?

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

	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

Coming Up

Reminders:

Crypto Project, Part 2 due Thursday at 6 PM

Wednesday

Web Attacks & Defenses

XSS
CSRF
SQL-injection

Next week

HTTPS

TLS and the CA ecosystem
Attacks and defenses