Web Security

Lecture 11: CPEN 400A

Based on

Nickolai Zeldovich. *6.858 Computer Systems Security, Fall 2014*. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu (Accessed 22 Nov, 2015).

And the

Wikipedia pages on XSS and CSRF attacks

Outline

- **Web security model**
- Same Origin Policy (SOP)
- Web Attacks
- Web Defences

Web Security Model in the old days…

- Before the advent of client-side JavaScript and single page web apps, the web security model was simple
  - Only two principals: client and server
  - Server decided who is trusted and who's not
  - Client simply prevented server from accessing machine local state (e.g., file-systems)

Web Security Today…

- Client integrates content/scripts from multiple domains (Servers) to create complex apps
  - Unclear who should have access to what

- Browser exposes a larger set of resources
  - Cookies, Authentication tokens
  - DOM state
  - Canvas, Local storage etc. (HTML5)

Security Problems

- How to prevent untrusted server side content from accessing private information of client ?
- How to protect two different server domains from each other (both read and write access)?
- How to protect server from malicious clients that attempt to exploit it ?

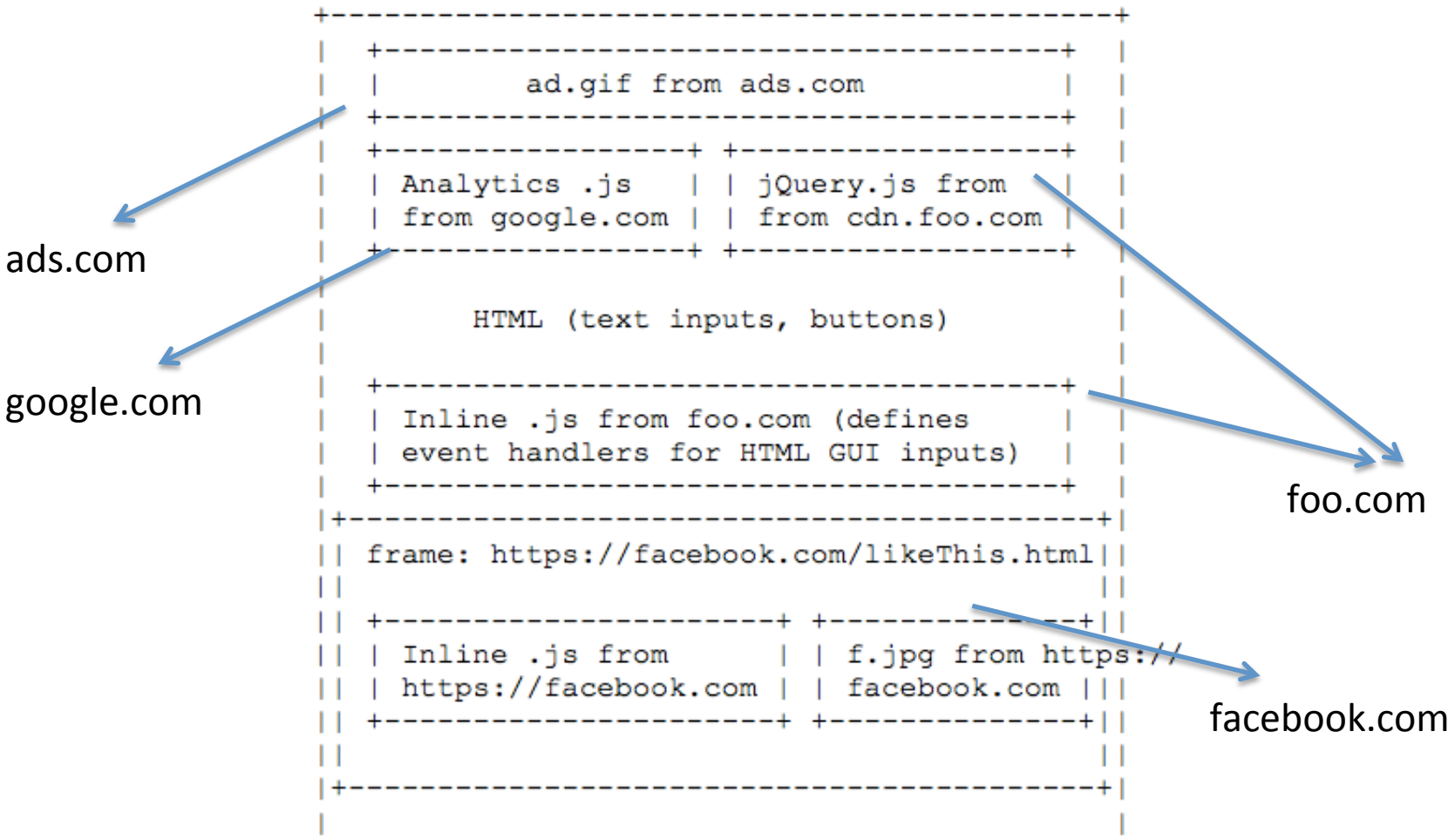
Mitigation Methods

- How to prevent untrusted server side content from accessing private information of client ?
  - Enforced by browser sandboxing model
- How to protect two different server domains from each other (both read and write access)?
  - Enforced by Same Origin Policy (SOP)
- How to protect server from malicious clients that attempt to exploit it ?
  - Enforced partly by Same Origin Policy (SOP)
  - Network firewalls for attacks below application level

Outline

- Web security model
- **Same Origin Policy (SOP)**
- Web Attacks
- Web Defences

Typical Web Application



Same Origin Policy

- Intent: Two different web domains should not be able to tamper with each other's contents

- Easy to state, but many exceptions in practice
  - Visual display is shared
  - Timing and DOM events are shared
  - Cookies can be shared
  - Send/receive messages for Cross-Origin Requests

Main idea

- Assign an origin for each resource in a web page (e.g., cookies, DOM sub-tree, network)
  - A script can only access elements belonging to the same origin as itself
- Definition of an origin (scheme, hostname, port)
  - Scheme: Protocol (typically http or https)
  - hostname: domain name (e.g., foo.com/index.html)
  - Port: foo.com:8080 (if unspecified, defaults to 80 for http and 443 for https))

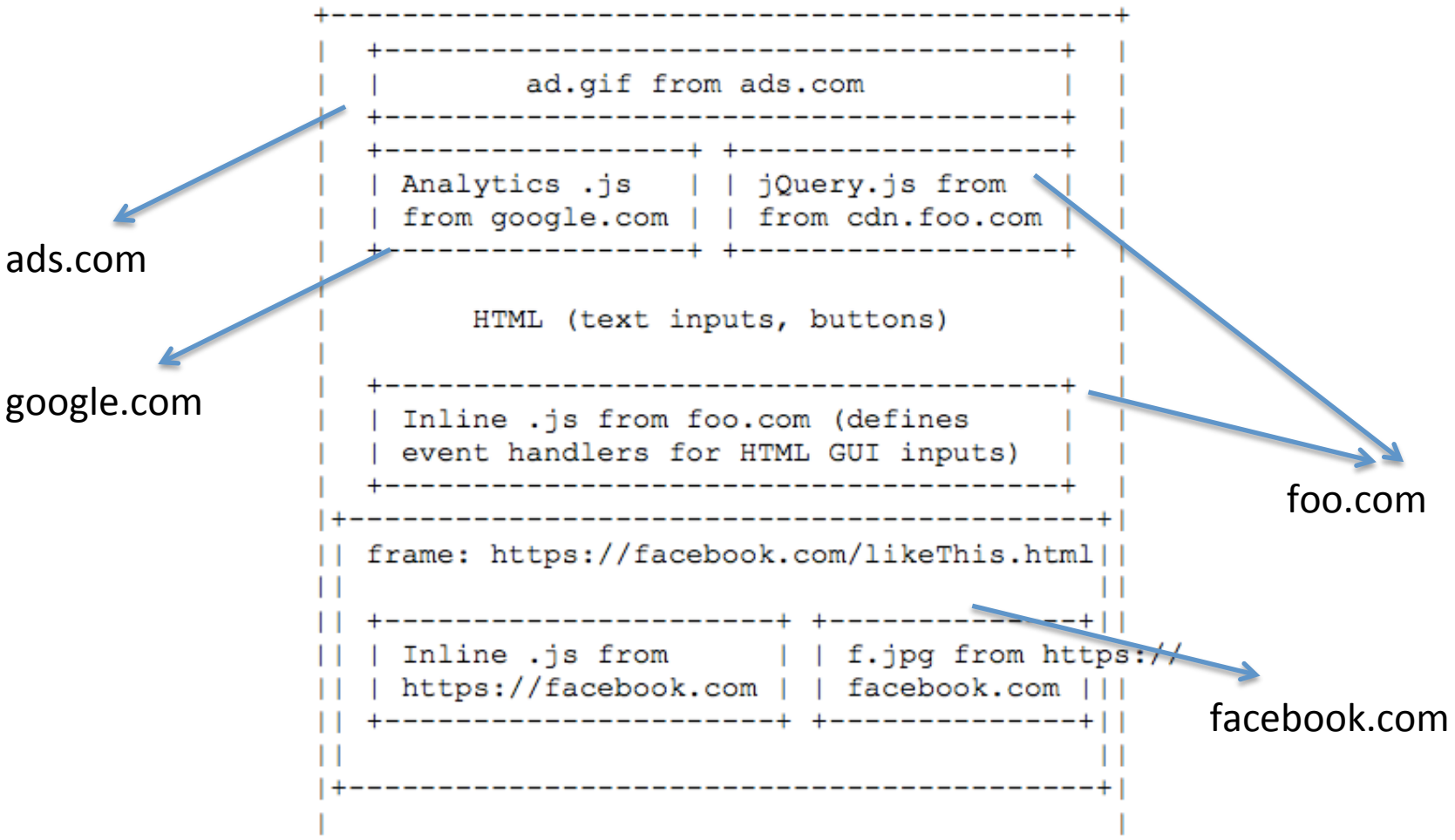
Rules of SOP

- Each frame gets the origin of its URL
- Scripts executed by a frame execute with the authority of the HTML file's origin
  - True for both inline scripts and those pulled from external domains
- Passive content (e.g., CSS, Images) can't run code and is hence given zero authority

Frames and SOP

- A Frame is a self-contained entity in a webpage which has scripts and content
- A frame's origin is set to the domain it comes from, but if and only if it explicitly sets the property domain="xyz.com"
- Subframes can set their "domain=" property to only their parent domain(s) or themselves
  - Example: "ece.ubc.ca" can set its domain property to "ubc.ca", but not "utoronto.ca" (for example)

Example Revisited



Class Activity

- Can the Google analytics and jquery script access foo.com's contents and resources ?
- Can JavaScript code in Facebook's frame access the contents of foo.com's frame ?
- Can JavaScript code in the Facebook.com frame send an AJAX message to foo.com ?
- Can JavaScript code in the Facebook.com frame import the scripts from foo.com ?
- Can JS code in the cdn.foo.com access JS code in the frame from foo.com ?

Solution to the Activity

- Yes, as they are in the same frame and execute with the authority of foo.com
- No, because they have different origins. This is even if facebook.com used http and not https
- No, unless foo.com allows cross-origin resource sharing requests (CORS requests)
- Yes, but it has to add them to the facebook.com frame, not foo.com's frame
- Depends. Both foo.com and cdn.foo.com frames need to set their domain=foo.com to allow this

Outline

- Web security model
- Same Origin Policy (SOP)
- **Web Attacks**
- Web Defences

Web Attacks

- Primarily result from attempts to trick the user or browser into breaking the SOP's restrictions
- Two prominent kinds
  - Cross-site Scripting Attacks (XSS)
  - Cross-site Request Forgery (CSRF)
- NOTE: We don't consider attacks that attempt to break the browser's sandbox model in this course

XSS Attacks

- Somehow get the browser to execute a script with the permissions of the attacked domain
  - Non-persistent (disappears after page reloads)
  - Persistent (persists across page reloads)

- Most common method: somehow inject JavaScript code into a resource of the attacked domain so that the code executes with the authority of the parent and can access it

Non-persistent XSS Attacks

- Occurs when server-side code accepts a query string or form submitted by the user, and sends the string back to the client as a new page or AJAX response without validating it
  - User can inject malicious JavaScript code into the query string or form input (can be hidden)
  - The script when it is sent back now executes with the authority of the server's origin and can access all resources of the same origin at the client

Example of an XSS attack (non-persistent)-1

- Consider a web-page with a search box, that sends the user-specified search string “str” to the server as a parameter of the GET request (“?search=str”). The server looks for the “str”, and if it’s not found, displays “str not found”
  - No checks are done on the server on str (say)

- Malicious user crafts a URL with JS code following the “search=“, for example, “search=?<script type=‘text\javascript’>alert(“bad”);</script>”

Example of an XSS attack (non-persistent)-2

- Let's assume that the malicious user now sends the URL as a embedded hyperlink to other users of the website (say thro' an email)
  - Other user clicks on the malicious URL
- Now, the other user goes to the website and executes the malicious user's injected JS code in their browser. This is bad as the code may for example steal the other user's authentication cookies and send it to the malicious user.

Persistent XSS Attack

- In a persistent XSS attack, the attack string is stored on the server so that future visits to the website (by the same user or different users) would also be subject to the attack
  - Much more devastating than the reflected attacks
  - Result from server not checking the user-specified string before storing it to a database or file (say)

Example of XSS attack (persistent)

- Consider a website with a comment box for users to leave comments. These comments are displayed for all users of the website.
- Assume that the server does not sanitize the comment text in any way and stores it in a DB.
- A malicious user leaves a comment containing malicious JS code, which is stored in the DB, and sent to every user of the website
- The code executes with the permissions of the server on every user's visit to the page, and sends their authentication credentials to malicious user

XSRF Attack

- An attacker attempts to request a URL sent to a user by spoofing it to their benefit
  - Relies on the use of reproducible and guessable URLs (typically as parameters of GET requests)
  - Cookies are automatically sent with every request, and hence the URL can perform malicious actions on behalf of the client
  - Do not require the server to accept/allow JavaScript code (unlike XSS attacks)

Example of an XSRF attack

- Assume that a banking website allows money transfers using the following URL format

http://bank.com/transfer.do?to=me&amt=100

A malicious user can trick another user into clicking the URL (say through an email). If they have logged into the bank's website, then the request will execute with the privileges of the logged in user.

- Relies on social engineering to carry out attack
- Malicious URL can be hidden (e.g., in images)

Outline

- Web security model
- Same Origin Policy (SOP)
- Web Attacks
- **Web Defences**

XSS attack prevention

- **Sanitizing user input by checking for JS**
  - Hard to do as JS code can be concealed in many ways (e.g., by escaping within HTML or CSS tags)
  - Performance overhead on the server for parsing inputs

- **Lighter-weight but incomplete methods**
  - Tying cookies to the IP address of the user logged in (works only for XSS attacks that try to steal cookies)
  - Disabling scripts on the page or in a specific section of the page (may prevent legit. scripts)
  - New method: Content security policy (allow servers to specify approved origins of content for web browsers)

XSRF Attack Prevention

- **Make the URL hard to guess by attaching a random nonce or client-specific key to it**
  - Works only if nonce/key is not leaked, and is complex
- **Things that don't work, but are deployed**
  - Using POST instead of GET requests (pointless)
  - Using multi-step transactions (makes it harder for attacker, but they can still forget the sequence)
  - Using a secret cookie (all cookies will be submitted with every request, even the secret ones)

HTML5 Security and Beyond…

- HTML5 provides two clean and secure workarounds for the Same Origin Policy (SOP)

- To send a message to another frame in the same page, use the *postMessage* API function as follows

  postMessage(msg, origin);

- To send a message to a server from a different origin, use a special CORS request API that uses an XMLHttpRequest2 object (object has a 'withCredentials' property that you can check)

```
var xhr = new XMLHttpRequest();
if ("withCredentials" in xhr) {
        xhr.open(method, url, true);
```

Outline

- Web security model
- Same Origin Policy (SOP)
- Web Attacks
- Web Defences