6.858 Lecture 8
Web Security

What is the web? In the old days, it was a simple client/server architecture (client was your web browser, server was a machine on the network that could deliver static text and images to your browser).

- In the old days, the server-side was much more complex than the client-side: browsers didn't support rich interactivity, but the server might interface with databases,other servers, etc.
- Because the server was so much more complicated, "web security" focused on the server-side. Up to this point, this class has largely focused on the server-side as well (e.g., buffer overflows on web servers, privilege separation in the OKWS server).

The web has changed: now the browser is very complicated.

- JavaScript: Allows a page to execute client-side code.
- DOM model Provides a JavaScript interface to the page's HTML, allowing the page to add/remove tags, change their styling, etc.
- XMLHttpRequests (AJAX): Asynchronous HTTP requests.
- Web sockets: Full-duplex client-server communication over TCP.
- Web workers: Multi-threading support.
- Multimedia support: <video>, web cams, screen-sharing.
- Geolocation: Browser can determine your location by examining GPS units. Firefox can also locate you by passing your WiFi information to the Google Location Service.
- <canvas> and WebGL: Bitmap manipulation and interactive 2D/3D graphics.
- Nacl: Allows browsers to run native code!

The web is now a complex platform for distributed computation! But what does this mean for security?

- The threat surface is huge!
- A single web application now spans multiple programming languages, OSes, hardware platforms. I might be running Firefox on Windows interacting with a Linux server running Apache and interfacing with memcached and MySQL).
- All of this composition makes it difficult to verify end-to-end correctness, or even understand what the system is doing. Ex: Parsing contexts and content sanitization.

```
<script> var x = 'UNTRUSTED'; </script>
     //Single quote breaks out of JS string
     //context into JS context
     //
     //"</script>" breaks out of JS context
     //into HTML context
```

- The web specs are incredibly long, very complex, occasionally contradictory, and constantly evolving.
    - So, browser vendors do something that roughly resembles the specs and then laugh about it with their friends.
    - If you want to understand the horror, go to quirksmode.org.

In this lecture, we're going to focus on the client-side of a web application. In particular, we're going to focus on how to isolate content from different providers that has to reside within the same browser.

- Big difference between a web application and a traditional desktop application: the bits in a desktop application typically come from a single vendor (e.g., Microsoft or Apple or TurboTax), but a single web application contains content from a bunch of different principals!

```
+---------------------------------------------+
|  +---------------------------------------+  |
|  |        ad.gif from ads.com            |  |
|  +---------------------------------------+  |
|  +-----------------+ +-------------------+  |
|  | Analytics .js   | | jQuery.js from    |  |
|  | from google.com | | from cdn.foo.com  |  |
|  +-----------------+ +-------------------+  |
|                                             |
|        HTML (text inputs, buttons)          |
|                                             |
|  +---------------------------------------+  |
|  | Inline .js from foo.com (defines      |  |
|  | event handlers for HTML GUI inputs)   |  |
|  +---------------------------------------+  |
|+-------------------------------------------+|
|| frame: https://facebook.com/likeThis.html||
||                                           ||
|| +----------------------+ +---------------+||
|| | Inline .js from      | | f.jpg from https://
|| | https://facebook.com | | facebook.com |||
|| +----------------------+ +---------------+||
||                                           ||
|+-------------------------------------------+|
|                                             |
```

Question: Which pieces of JavaScript code can access which pieces of state? For example…

- Can the analytics code from google.com access state in the jQuery code from cdn.foo.com? [Seems maybe bad since different principals wrote the code, but they are included in the same frame . . .]
- Can the jQuery code from cdn.foo.com access state in the inline JavaScript code defined by foo.com? [They're *almost* from the same place . . .]
- Can the analytics code or jQuery access the HTML text inputs? [We've got to make that content interactive somehow.]
- Can JavaScript in the Facebook frame touch any state in the foo.com frame? Does it matter that the Facebook frame is https://, but the foo.com frame is regular http://?

To answer these questions, browsers use a security model called the same-origin policy.

- Vague goal: Two different websites should not be able to tamper with each other's content.
- Easy to state, but tricky to implement.
    - Obviously bad: If I have two different web sites open, the first site should not be able to overwrite the visual display of the second site.
    - Obviously good: Developers should be able to create mash-up sites that combine content from mutually cooperative web sites.
        - -Ex: A site that combines Google Map data with real estate data.
        - -Ex: Advertistements.
        - -Ex: Social media widgets (e.g., the Facebook "like" button).
    - Hard to say: If a page from web server X downloads a JavaScript library from a different server Y, what capabilities should that script have?
- Basic strategy of same-origin policy: The browser assigns an origin to every resource in a page, including JavaScript libraries. JavaScript code can only access resources that belong to its origin.
- Definition of an origin: scheme + hostname + port
- For example:
    - http://foo.com/index.html (http, foo.com, 80 [implicit])
    - https://foo.com/index.html (https, foo.com, 443 [implicit])
    - http://bar.com:8181/index.html (http, bar.com, 8181)
- Schemes can be http, https, ftp, file, etc.
- Four main ideas:
    1. Each origin is associated with client-side resources (e.g., cookies, DOM storage, a JavaScript namespace, a DOM tree, windows, a visual display area, network addresses).
        - An origin is the moral equivalent of a UID in the Unix world.
    2. Each frame gets the origin of its URL. A frame is the moral equivalent of a process in Unix.
    3. Scripts included by a frame execute with the authority of that HTML file's origin. This is true for both inline scripts *and* ones that are pulled from external domains! [Unix analogy: Running a binary that's stored in somebody else's home directory.]

4. Passive content (e.g., images and CSS) can't execute code, so this content is given zero authority.

- Returning to our example:
  - The Google analytics script and the jQuery script can access all the resources belonging to foo.com (e.g., they can read and write cookies, attach event handlers to buttons, manipulate the DOM tree, access JavaScript variables, etc.).
  - JavaScript code in the Facebook frame has no access to resources in the foo.com frame, because the two frames have different origins. The two frames can only talk via postMessage(), a JavaScript API that allows domains to exchange immutable strings.
    - If the two frames *were* in the same origin, they could use window.parent and window.frames[] to directly interact with each other's JavaScript state!
  - JavaScript code in the Facebook frame cannot issue an XMLHttpRequest to foo.com's server [the network is a resource with an origin!] . . .
  - However, the Facebook frame *can* import scripts, CSS, or images from foo.com (although that content can only update the Facebook frame, since the content inherits the authority of the Facebook origin, not foo.com origin).
  - The browser checks the type of ad.gif, determines that ad.gif is a image, and concludes that the image should receive no authority at all.

What happens if the browser mistakenly identifies the MIME type of an object?

- Old versions of IE used to do MIME sniffing.
  - Goal: Detect when a web server has given an incorrect file extension to an object (e.g., foo.jpg should actually be foo.html).
  - Mechanism: IE looks at the first 256 bytesof the file and looks for magic values which indicate a file type. If there's a disagreement between the magic values and the file extension, IE trusts the file extension.
  - Problem: Suppose that a page includes some passive content (e.g., an image) from an attacker-controlled domain. The victim page thinks that it's safe to import passive content, but the attacker can intentionally put HTML+JavaScript in the image and execute code in the victim page!
- Moral: Browsers are complex---adding a well-intentioned feature may cause subtle and unexpected security bugs.

Let's take a deeper look at how the browser secures various resources.

Frame/window objects

- Note: A frame object is a DOM node of type HTMLIFrameElement, whereas the window object is the alias for the global JavaScript namespace. Both objects have references to each other.
- Get the origin of their frame's URLs

-OR-

- Get the origin of the adjusted document.domain
    - A frame's document.domain is originally derived from the URL in the normal way.
    - A frame can set document.domain to be a suffix of the full domain. Ex:
        - x.y.z.com //Original value
        - y.z.com //Allowable new value
        - z.com //Allowable new value
        - a.y.z.com //Disallowed
        - .com //Disallowed
    - Browsers distinguish between a document.domain that has been written, and one that has not, even if both have the same value! Two frames can access each other if:
        - They have both set document.domain to the same value, or
        - Neither has changed document.domain (and those values are equal in both frames)
    - These rules help protect a site from being attacked by a buggy/malicious subdomain, e.g., x.y.z.com trying to attack y.z.com by shortening its document.domain.

DOM nodes

- Get the origin of their surrounding frame

Cookies

- A cookie has a domain AND a path. Ex: *.mit.edu/6.858/
    - Domain can only be a (possibly full) suffix of a page's current domain.
    - Path can be "/" to indicate that all paths in the domain should have access to the cookie.
- Whoever sets cookie gets to specify the domain and path.
    - Can be set by the server using a header, or by JavaScript code that writes to document.cookie.
    - There's also a "secure" flag to indicate HTTPS-only cookies.
- Browser keeps cookies on client-side disk (modulo cookie expiration, ephemeral cookies, etc.).

- When generating an HTTP request, the browser sends all matching cookies in the request.
    - Secure cookies only sent for HTTPS requests.
- JavaScript code can access any cookie that match the code's origin, but note that the cookie's path and the origin's port are ignored!
    - The protocol matters, because HTTP JavaScript cannot access HTTPS cookies (although HTTPS JavaScript can access both kinds of cookies).

- Q: Why is it important to protect cookies from arbitrary overwriting?

- A: If an attacker controls a cookie, the attacker can force the user to use an account that's controlled by an attacker!
    - Ex: By controlling a Gmail cookie, an attacker can redirect a user to an attacker controlled account and read any emails that are sent from that account.
- Q: Is it valid for foo.co.uk to set a cookie's domain to co.uk?
- A: This is valid according to the rules that we've discussed so far, but in practice, we should disallow such a thing, since ".co.uk" is semantically a single, "atomic" domain like ".com". Mozilla maintains a public list which allows browsers to determine the appropriate suffix rules for top-level domains. [https://publicsuffix.org]

HTTP responses: Many exceptions and half-exceptions to same-origin policy.

- XMLHttpRequests: By default, JavaScript can only send XMLHttpRequests to its origin server… unless the remote server has enabled Cross-origin Resource Sharing (CORS). The scheme defines some new HTTP response headers:
    - Access-Control-Allow-Origin specifies which origins can see HTTP response.
    - Access-Control-Allow-Credentials specifies if browser should accept cookies in HTTP request from the foreign origin.
- Images: A frame can load an image from any origin… but it can't look at the image pixels… but it can determine the image's size.
- CSS: Similar story to images--a frame can't directly read the content of external CSS files, but can infer some of its properties.
- JavaScript: A frame can load JavaScript from any origin . . . but it can't directly examine the source code in a <script> tag/XMLHttpRequest response body . . . but all JavaScript functions have a public toString() method which reveals source code… and a page's home server can always fetch the source code directly and then pass it to the page!
    - To prevent reverse-engineering, many sites minify and obfuscate their JavaScript.
- Plugins: A frame can run a plugin from any origin.
    - <embed src=...> // Requires plugin-specific elaborations.

Remember that, when the browser generates an HTTP request, it automatically includes the relevant cookies.

- What happens if the browser creates a frame with a URL like this?
    - http://bank.com/xfer?amount=500&to=attacker
- This attack is called a cross-site request forgery (CSRF).
- Solution: Include some random data in URLs that is difficult for the attacker to guess. Ex:

```
<form action="/transfer.cgi" ...>
         <input type="hidden"
                name="csrfToken"
```

```
value="a6dbe323..."/>
```

- Each time a user requests the page, the server generates HTML with new random tokens. When the user submits a request, the server validates the token before actually processing the request.
- Drawback: If each URL to the same object is unique, it's difficult to cache that object!

Network addresses almost have an origin.

- A frame can send HTTP *and* HTTPS requests to a host+port that match its origin.
- Note that the security of the same-origin policy depends on the integrity of the DNS infrastructure!
- DNS rebinding attack
    - Goal: Attacker wants to run attacker-controlled JavaScript code with the authority of an origin that he does not control (victim.com).
    - Approach:
        1) Attacker registers a domain name (e.g., attacker.com) and creates a DNS server to respond to the relevant queries.
        2) User visits the attacker.com website, e.g., by clicking on an advertisement.
        3) The attacker website wants to downloads a single object, but first, the browser must issue a DNS request for attacker.com. The attacker's DNS server responds with a DNS record to the attacker's IP address. However, the record has a short time-to-live.
        4) The attacker rebinds attacker.com to the IP address of victim.com.
        5) A bit later, the attacker website creates an XMLHttpRequest that connects to attacker.com. That request will actually be sent to the IP address of victim.com! The browser won't complain because it will revalidate the DNS record and see the new binding.
        6) Attacker page can now exfiltrate data, e.g., using CORS XMLHttpRequest to the attacker domain.
    - Solutions:
        - Modify DNS resolvers so that external hostnames can never resolve to internal IP addreses.
        - Browsers can pin DNS bindings, regardless of their TTL settings. However, this may break web applications that use dynamic DNS (e.g., for load-balancing).

What about the pixels on a screen?

- They don't have an origin! A frame can draw anywhere within its bounding box.
- Problem: A parent frame can overlay content atop the pixels of its child frames.
    - Ex: At attacker creates a page which has an enticing button like "Click here for a free iPad!" Atop that button, the page creates a child frame that contains the Facebook "Like" button. The attacker places that button atop

the "free iPad" button, but makes it transparent! So, if the user clicks on the "free iPad" button, he'll actually "Like" the attackers page on Facebook.

- Solutions
    1) Frame-busting code: Include JavaScript that prevents your page from being included as a frame. Ex: if(top != self)
    2) Have your web server send the X-Frame-Options HTTP response header. This will instruct the browser not to put your content in a child frame.

What about frame URLs that don't have an origin?
Ex: file://foo.txt
about:blank
javascript:document.cookie="x"

- Sometimes the frame is only accessible to other frames with that protocol (e.g., file://). [This can be irritating if you're debugging a site and you want to mix file:// and http:// content].
- Sometimes the frame is just inaccessible to all other origins (e.g., "about:").
- Sometimes the origin is inherited from whoever created the URL (e.g., "javascript:"). This prevents attacks in which a attacker.com creates a frame belonging to victim.com, and then navigates the victim frame to a javascript: URL--we don't want the JavaScript to execute in the context of victim.com!

Names can be used as an attack vector!
- IDN: internationalized domain names (non-latin letters).
- Supporting more languages is good, but now, it can be difficult for users to distinguish two domain names from each other.
- *Ex: The Cyrillic "C" character looks like the Latin "C" character! So, an attacker can buy a domain like "cats.com" (with a Cyrillic "C") and trick users who thought that they were going to "cats.com" (Latin "C").
- Good example of how new features can undermine security assumptions.
- Browser vendors thought registrars will prohibit ambiguous names.
- Registrars thought browser vendors will change browser to do something

Plugins often have subtly-different security policies
- Java: Sort of uses the same-origin policy, but Java code can set HTTP headers (bad! see "Content-Length" discussion), and in some cases, different hostnames with the same IP address are considered to share the same origin.
- Flash: Developers place a file called crossdomain.xml on their web servers. That file specifies which origins can talk to the server via Flash.

HTML5 introduces a new screen-sharing API: Once the user gives permission, a site can capture the entire visible screen area and transmit it back
to the site's origin.

- So, if an attacker page can convince the user to grant screen-sharing permission, the attacker page can open an iframe to a sensitive site (e.g., banking, Facebook, email), and capture the screen contents!
- The iframe will send cookies, so the user will automatically be logged in, allowing the attacker to see "real" information, not boring login page stuff.
- Attacker can make the iframe flash only briefly to prevent the user from noticing the mischief.
- Possible defenses:
    - Allow users to only screen-share part of the DOM tree? It seems like this will be tedious and error-prone.
    - Only allow an origin to screen-capture content from its own origin? Seems like a more reasonable approach, although it prevents

Since "The Tangled Web," there have been various modifications and additions to the aggregate web stack.

- In general, things have gotten more complicated, which is typically bad for security.
- For reference, here are some of the new features:
    - http://en.wikipedia.org/wiki/Content_Security_Policy
    - http://en.wikipedia.org/wiki/Strict_Transport_Security
    - http://en.wikipedia.org/wiki/Cross origin_resource_sharing
    - HTML5 iframe sandbox attribute [http://msdn.microsoft.com/enn us/hh563496.aspx]

The browser security model is obviously a mess. It's very complex and contains a lot of subtleties and inconsistencies.

- Q: Why not rewrite the security model from scratch?
- A1: Backwards compatibility! There's a huge amount of preexisting web infrastructure that people rely on.
- A2: How do we know that a new security model would be expressive enough? Users typically do not accept a reduction of features in exchange for an increase in security.
- A3: Any security model may be intrinsically doomed---perhaps all popular systems are destined to accumulate a ton of features as time progresses. [Ex: Word processing programs, smartphones.]
- What might a better design look like?
    - Strict isolation Embassies---everything is a network message, even locally
        - https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final85.pdf
    - Don't make policy extraction and enforcement dependent on complex parsing rules (remember our sanitization example)

    - Only add features in small, clearlyn   defined quanta with minimal room for implementation error or interpretation mistakes---remove ambiguity and the need for guessing.