https://brown-csci1660.github.io

CS1660: Intro to Computer Systems Security
Spring 2025

Lecture 10: Web Security II

Co-Instructor: **Nikos Triandopoulos**

February 27, 2025

BROWN

CS1660: Announcements

- Course updates
  - Project 2 is out
  - Homework 1 is now due this Sunday, March 2
  - Where we are
    - **Part I: Crypto**
    - **Part II: Web**
    - Part III: OS
    - Part IV: Network
    - Part V: Extras

Today

- Web security

Cookies

- HTTP is a stateless protocol; cookies used to emulate state
- Servers can store cookies (name-value pairs) into browser
  - user preferences (e.g., language and page layout), user tracking, authentication
  - expiration date can be set
  - may contain sensitive information (e.g., for user authentication)

```
POST /login.php HTTP/1.1
Set-Cookie: Name: sessionid
            Value: 19daj3kdop8gx
            Domain: cs.brown.edu
            Expires: Wed, 21 Feb 2024 …
```

- Browser sends back cookies to server on the next connection

Cookies scope

Each cookie has a scope

- base domain, which is a given host
  - e.g., brown.edu
- plus, optionally, all its subdomains
  - cs.brown.edu, math.brown.edu, www.cs.brown.edu , etc.
  - for ease of notation, included subdomains are denoted as .
    - e.g., .brown.edu
    - in fact, specified in HTTP with the "Domain:" attribute of a cookie

Same Origin Policy: Cookie Reads

Websites can only read cookies within their scope

- Browser has cookies with scope
  - brown.edu
  - .brown.edu
  - .math.brown.edu
  - cs.brown.edu
  - .cs.brown.edu
  - blog.cs.brown.edu
- Browser accesses cs.brown.edu
- Browser sends cookies with scope
  - .brown.edu
  - cs.brown.edu
  - .cs.brown.edu

Same Origin Policy: Cookie Writes

A website can set cookies for (1) its base domain;
or (2) a super domain (except TLDs) and its subdomains

- Browser accesses cs.brown.edu
- cs.brown.edu can set cookies for
  - .brown.edu
  - cs.brown.edu
- But not for
  - google.com
  - .com
  - .math.brown.edu
  - brown.edu
  - …

Session Management

Session

- keep track of client over a series of requests
- server assigns clients a unique, unguessable ID
- clients send back ID to verify themselves

Session

- necessary in sites with authentication
  - e.g., banking
- useful in most other sites
  - e.g., remembering preferences
- various methods to implement them
  - mainly cookies
  - but also could be in HTTP variables

Session Management (cont.)

- Goal
  - users should not have to authenticate for every single request
- Problem
  - HTTP is stateless
- Solution
  - user logs in once
  - server generates session ID and gives it to browser
    - temporary token that identifies and authenticates user
  - browser returns session ID to server in subsequent requests

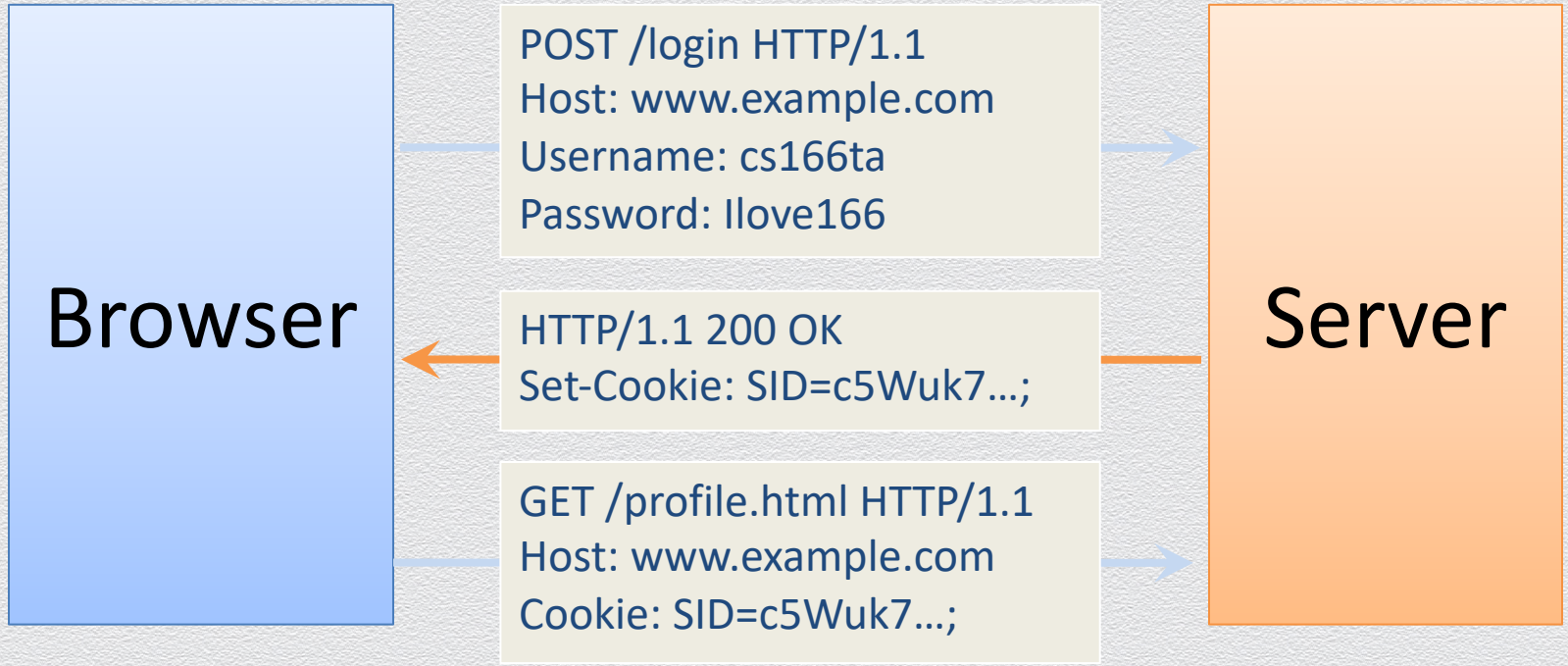
Specifications for a Session ID

- Created by server upon successful user authentication
  - generated as long random string
  - associated with scope (set of domains) and expiration
  - sent to browser
- Kept as secret shared by browser and server
- Transmitted by browser at each subsequent request to server
  - must use secure channel between browser and server
- Session ID becomes invalid after expiration
  - user asked to authenticate again

Implementation of Session ID

- Cookie
  - Transmitted in HTTP headers
  - Set-Cookie: SID=c5Wuk7…
  - Cookie: SID=c5Wuk7…
- GET variable
  - Added to URLs in links
  - https://www.example.com?SID=c5Wuk7…
- POST variable
  - Navigation via POST requests with hidden variable
  - <input type="hidden" name="SID" value="c5Wuk7…">

Session ID in Cookie



Session ID in Cookie (cont.)

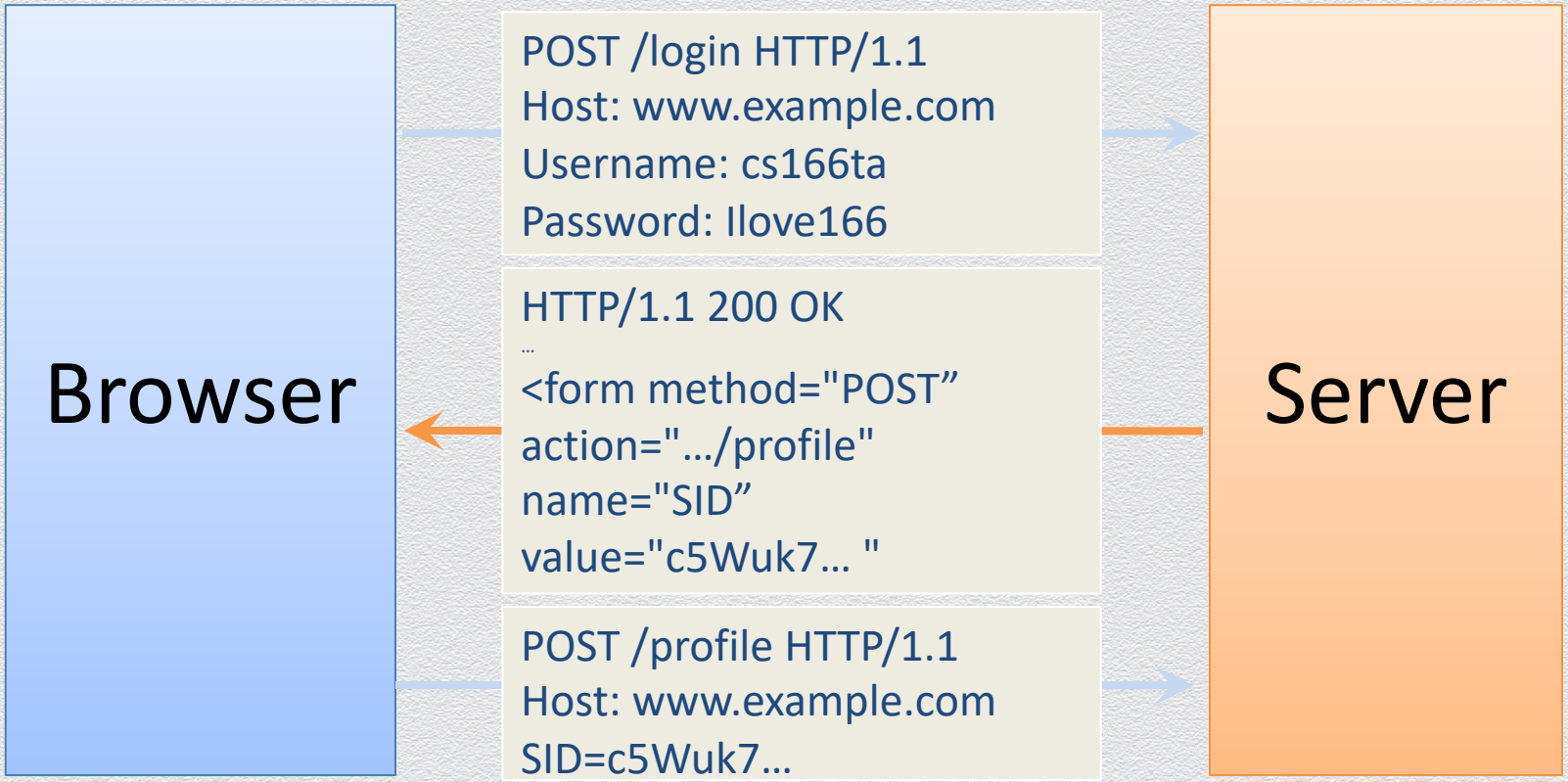
- Advantages
  - Cookies automatically returned by browser
  - Cookie attributes provide support for expiration, restriction to secure transmission (HTTPS), and blocking JavaScript access (httponly)
- Disadvantages
  - Cookies are shared among all browser tabs
    - (**not** other browsers or incognito)
  - Cookies are returned by browser even when request to server is made from element (e.g., image or form) within page from other server
  - This may cause browser to send cookies in context not intended by user

Session ID in GET Variable

Session ID in GET Variable (cont.)

- Advantages
  - Session ID transmitted to server only when intended by user
- Disadvantages
  - Session ID inadvertently transmitted when user shares URL
  - Session ID transmitted to third-party site within referrer
  - Session ID exposed by bookmarking and logging
  - Server needs to dynamically generate pages to customize site navigation links and POST actions for each user
  - Transmission of session ID needs to be restricted to HTTPS on every link and POST action

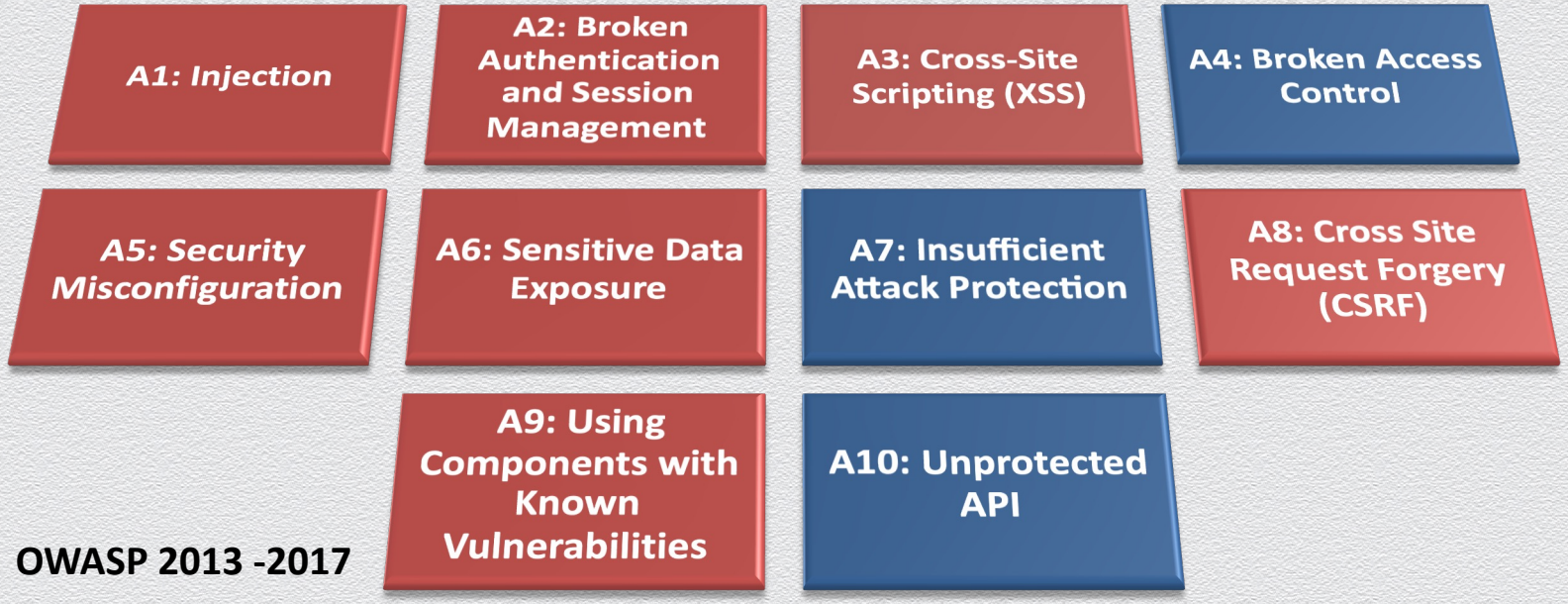
Session ID in POST Variable



Session ID in POST Variable

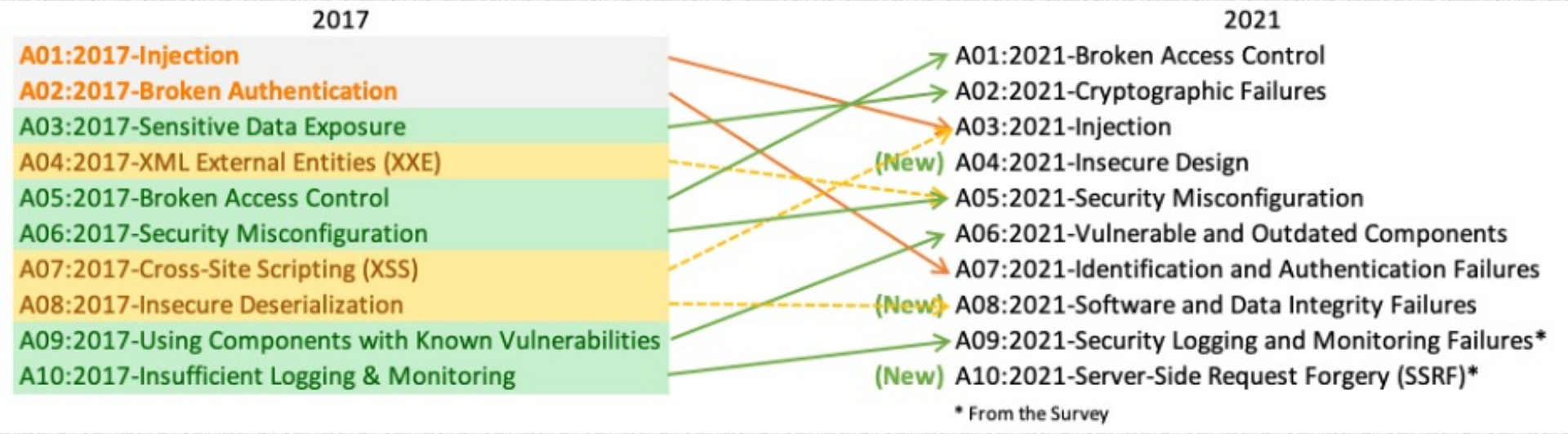
- Advantages
  - Session ID transmitted to server only when intended by user
  - Session ID not present in URL, hence not logged, bookmarked, or transmitted within referrer
- Disadvantages
  - Navigation must be made via POST requests
  - Server needs to dynamically generate pages to customize forms for each user
  - Transmission of session ID needs to be restricted to HTTPS on every link and POST action

OWASP Top Ten (2013-17)



OWASP
The Open Web Application Security Project
http://www.owasp.org

OWASP 2017 - 2021



www.owasp.org/index.php/Top_10

OWASP
The Open Web Application Security Project
http://www.owasp.org

What we know so far

- HTTP and Browsers
- Cookies (and what happens if you steal them)
- “Client-side controls”

Today

- More about requests: cross-origin/same-origin
- CSRF attacks
- Session token entropy

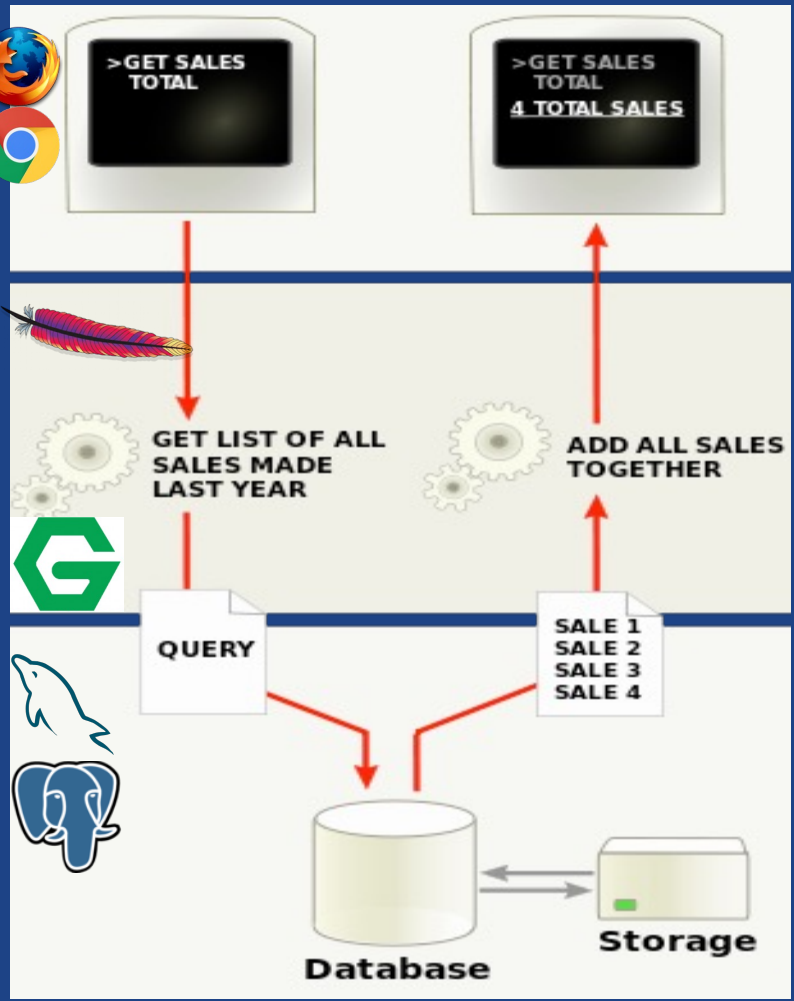
Benefits of the Web

- A web browser is usually sufficient, typically preinstalled and free
- No upgrade procedure, since all new features are implemented on the server and automatically delivered to the users
- Cross-platform compatibility in most cases (i.e., Windows, Mac, Linux, etc.) , everything happens in a web browser window
- Easy to integrate into other server-side web procedures (i.e. email, searching, localization etc.)
- HTML5 allows the creation of richly interactive environments natively within browsers

Web Architecture

A web site usually is a collection of web pages that are:

- Accessed by users over a network through the HTTP or HTTPS protocol
- Coded in a browser-supported programming language (i.e JavaScript, HTML, etc.)
- Used through a common web browser (EDGE, Firefox, Chrome, Safari, Opera, etc.) to render the pages executable, with usually the help of some cookies
- Managed by a web application with a client–server architecture (i.e. 3-tiers) in which Presentation, Logic, and Data tiers are logically separated



Review: Cookies

Key-value pairs (stored in browser) that keep track of certain information

- User preferences, session ID, tracking, ad networks, etc.
- Key attributes (so far):
  - **`Domain`**: eg. cs.brown.edu .brown.edu

When a request is made, all cookies with a matching domain are sent with it
…subject to certain other browser restrictions (today's topic!)

Same origin policy (SOP): so far

- Limits how a site can set cookies*
- Limits which cookies are sent on each request

In general, “origin” must match:

`https://site.example.com[:443]/some/path`

Cookies: examples

- Session ID: cookie used for authentication
- App state: Shopping cart, page views
- Ad networks/tracking
- …

User Tracking

- Done mainly through cookies
- Keeps track of users and information about them
  - Could be their online habits, behaviors, and preferences
  - Could also be demographics — race, gender, age, etc.
- Can be used in a (arguably) benign manner
  - Used for company statistics
  - Personalized content feeds and targeted advertising
- Can also be used malevolently
  - Can be viewed as infringing on privacy rights
  - Ex: Facebook—Cambridge Analytica Scandal in 2018

Web Access Control

- Authentication
  - Username and password, additional factors
- Session management
  - Keep track of authenticated users across sequence of requests
- Authorization
  - Check and enforce permissions of authenticated users

Session Management

- Goal
  - Users should not have to authenticate for every single request
- Problem
  - HTTP is stateless
- Solution
  - User logs in once
  - Server generate session ID and gives it to browser
    - Temporary token that identifies and authenticates user
  - Browser returns session ID to server in subsequent requests

SOP: JavaScript and iframes

JavaScript

- Programming language interpreted by the browser
- Code embedded within `<script> … </script>` tags
- Defining functions:

```
<script
  type="text/javascript">
  function hello() {
  alert("Hello world!");}
</script>
```

- Examples:
  - Read / modify elements of the DOM
    - "Look for all <p> tags and return the content"
    - "Change the content within all <img> tags to _____"
  - Open another window

```
window.open("http://brown.edu")
```

  - Read cookies

```
alert(document.cookie);
```

Javascript

- Scripting language interpreted by browser
- Fetched as part of a page (just like HTML, images)

Capabilities

- Read/modify most page elements
  - DOM: Document Object Model
- Make requests (often asynchronously)
- Powers essentially all modern webapps

Same Origin Policy: JavaScript

- Scripts loaded from a website have restrictions on accessing content from another website (e.g., in another tab)
- All code within `<script> … </script>` tags is restricted to the context of the embedding website
  - However, this includes embedded, external scripts
  - `<script src="http://mal.com/library.js"></script>`
  - The code from mal.com can access HTML elements and cookies on our website
  - **Notice**: Different from the SOP for third-party cookies

Question

Say our website is example.com, and we've embedded the script from mal.com in our website. If the script from mal.com sets a cookie, under which origin can it / will it be set?

A. example.com
B. mal.com
C. All of the above
D. None of the above

Answer

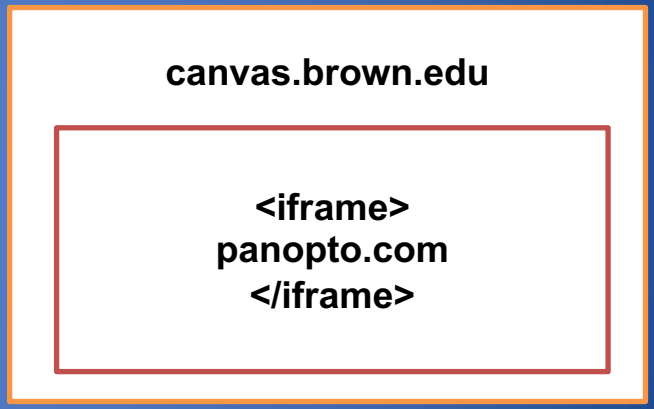
Say our website is example.com, and we've embedded the script from mal.com in our website. If the script from mal.com sets a cookie, under which origin will it be set?

A. example.com

Scripts run within the context of the embedding website, so the script from mal.com can set a cookie for example.com (but not for mal.com).

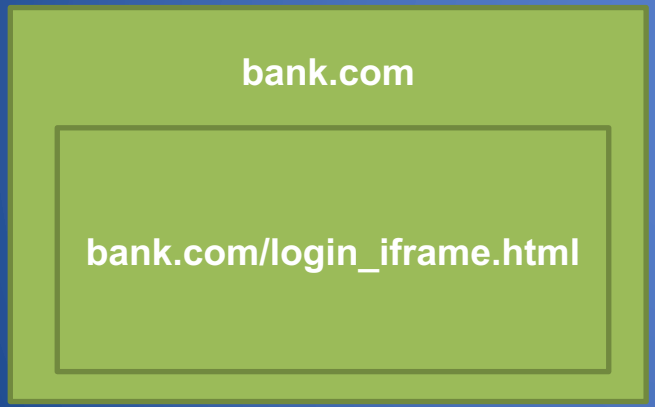
iframes

- Allows a website to "embed" another website's content
- Examples:
  - YouTube video embeds
  - Embedded Panopto lectures on Canvas
- Same origin policy?

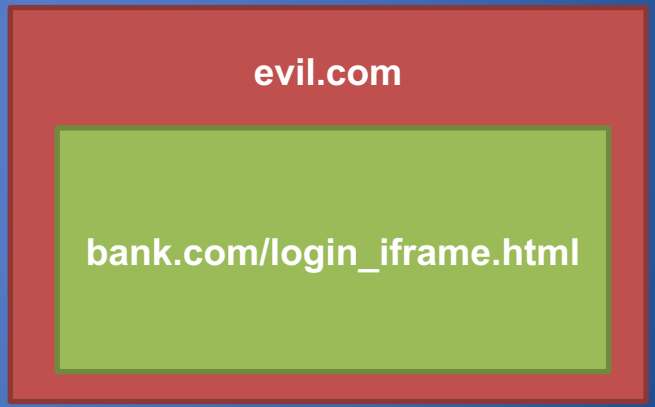


SOP: iframes

Only code from the same origin can access HTML elements on another site (or in an iframe).



bank.com can access HTML elements in the iframe (and vice versa)



evil.com cannot access HTML elements in the iframe (and vice versa).

SOP: Requests

Websites can submit requests to another site (e.g., sending a GET / POST request, image embedding, `Javascript requests (XMLHttpRequest)`)

- Can generally embed (display in browser) cross-origin response
  - Embedding an image
  - Opening content / opening the response to a request in an iframe

- Usually can't read (cross-origin response (i.e. via a script))
  - Sometimes websites always allow cross-origin reads
  - Why might this be bad?

SOP: Foreshadowing

- To reiterate: Websites can submit requests to another site
  - …and can display the responses on their own site (via iframe, img, etc.)
  - …but can't read the responses themselves (via a script)
- Attacker can still accomplish a lot with just sending out requests …

Bringing Everything Together…

- Cookies often contain an authentication token
  - Stealing a cookie == accessing account
- Perhaps your web application uses JavaScript to validate client-side input…
  - i.e. “You can only make Piazza posts with alphanumeric characters”
- What if I disable JavaScript on my browser?
  - No more client-side check
  - Can potentially inject HTML code; links; JavaScript into the web application…
- What happens if someone clicks on this link?
  - Click here!
- *More to come …*

Cross-Site Request Forgery (CSRF)

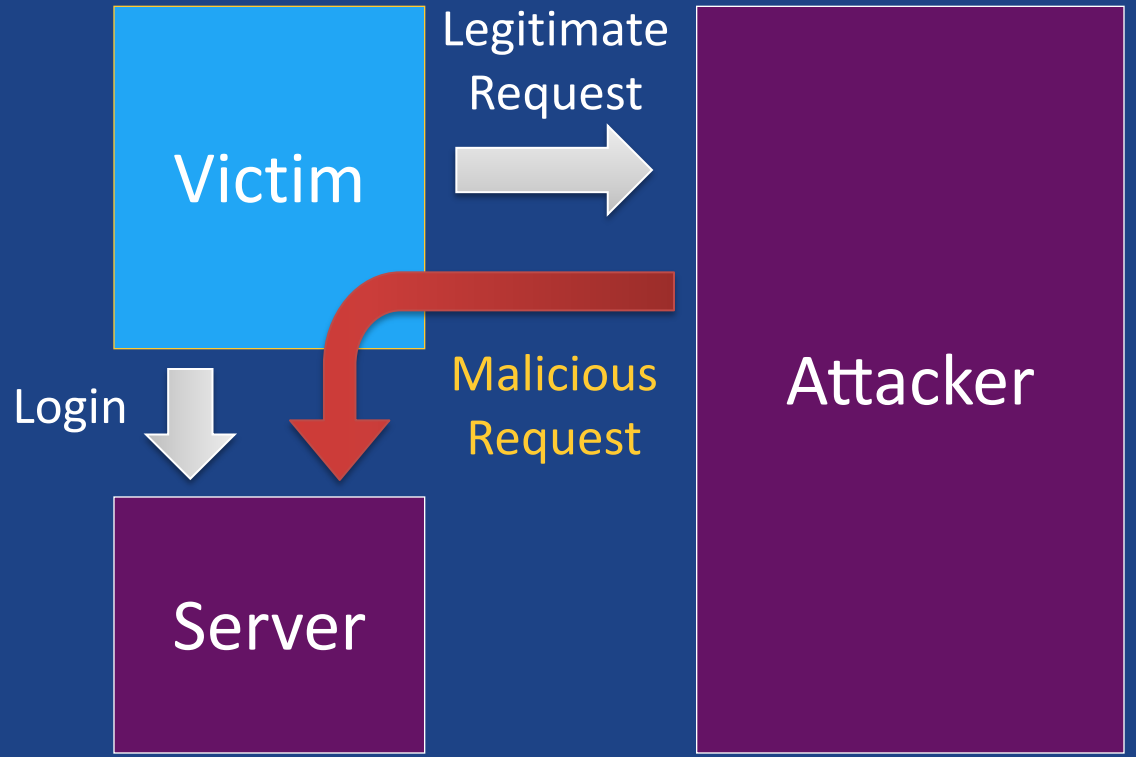
- Attacker's site has script that issues a request on target site
- Example

```
<form action="https://bank.com/wiretransfer" method="POST" id="rob">
<input type="hidden" name="recipient" value="Attacker">
<input type="hidden" name="account" value="2567">
<input type="hidden" name="amount" value="$1000.00">
…
document.getElementById("rob").submit();
```

- If user is already logged in on target site …
- Request is executed by target site on behalf of user
  - E.g., funds are transferred from the user to the attacker

CSRF Trust Relationships

- Server trusts victim (login)
- Victim trusts attacker enough to click link/visit site
- Attacker could be a hacked legitimate site



Question

Cross-Site Request Forgery relies primarily on which of the following trust relationships?

A. Server trusting victim
B. Victim trusting attacker
C. Server trusting attacker
D. Both A and B
E. All of the above

Answer

Cross-Site Request Forgery relies primarily on which of the following trust relationships?

A. Server trusting victim
B. Victim trusting attacker
C. Server trusting attacker
D. Both A and B
E. All of the above

CSRF Mitigation

- To protect against CSRF attacks, we can use a cookie in combination with a POST variable, called CSRF token
- POST variables are not available to attacker
- Server validates both cookie and CSRF token

CSRF: How to defend?

One way: CSRF token: server sends unguessable value to client, include as hidden variable in POST

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken" value="ABBE294xF. . .">
[...]
</form>
```

- On POST, server compares against expected value
- Could be random value stored on server, or signed/MAC'd by key on server

CSRF Token

- Token included as hidden parameter in POST
- Server-side validation
  - Action rejected if token is incorrect or missing
- Per-session tokens:
  - One token generated for current session and used for all requests
- Per-request tokens:
  - Randomize parameter name and/or value
  - Higher security but some usability concerns (e.g., back button functionality)

Another way: Verifying Source Origin

- Check that source origin matches target origin
  - "Referer" header: entire URL of page from which request is sent
  - "Referer" used by some websites for logging and analytics
  - "Origin" header: hostname of page from which request is sent
- Potential issue: Referer/Origin headers not always present for all requests

Another way

- Hardened session cookies: SameSite attribute
- SameSite=Strict: cookie can only be sent if domain matches URL bar
- SameSite=Lax: allows some top-level mitigations

Some recent changes to how browsers enforce this…

Token Patterns

Synchronizer Token

- Stateful
- Value randomly generated with large entropy
- Mapped to user's current session
- Server validates that token exists and is associated to user's session ID

Encrypted Token

- Stateless
- Token generated from user ID and timestamp
- Encrypted with server’s secret key
- Server validates token by decrypting it and checking that it corresponds to current user and acceptable timestamp

Custom Request Headers

- Check presence of some custom header, block request if absent
- Only way to set custom headers is through JavaScript
  - JavaScript unable to make cross-site requests due to Same-Origin-Policy
- Scenario
  - Alice is logged into bob.com
  - bob.com requires all incoming requests to contain header Bobs-Header
  - Bobs-Header set by JavaScript code present on each page of bob.com
  - Eve tricks Alice into visiting eve.com, which sends malicious request to bob.com on behalf of Alice
  - bob.com blocks Eve's request because Eve is unable to construct the request to include Bobs-Header

Other CSRF Mitigation Techniques

- Identifying source origin
  - Verify that the Referer header's hostname matches the target origin
- Custom request header
  - Generated by JavaScript
  - Subject to same origin policy
  - Verify presence of header on every request
- SameSite cookie attribute
  - "Strict" value prevents cookie from being sent in cross-site requests
  - Recent standard may not be supported by browser
- User-interaction
  - Re-authentication, one-time token, CAPTCHA, etc.
  - Strong defense but negatively impacts user experience

Strict SameSite Cookie Attribute

- Browser will only send cookie if the site for the stored cookie matches the URL of the page making the request
- Scenario
  - Alice logs in to bob.com, which sets cookie:
    Set-Cookie: sessionid=12345; Domain=bob.com; SameSite=Strict
  - Eve tricks Alice into visiting her page eve.com, which sends a malicious request to bob.com on behalf of Alice
  - Since the cookie has SameSite set to Strict, Alice's browser does not send sessionid to bob.com from eve.com
- Potential issue: Not all browsers have adopted default policy for websites that do not set SameSite

User Interaction

- Make a user reauthenticate, submit a one-time token, or do a CAPTCHA before performing any user-specific or privileged action on a website
- Scenario
  - Alice is logged into bob.com
  - Eve tricks Alice into visiting her page eve.com in another tab, which automatically redirects to send a malicious request to bob.com
  - Alice sees a login page for bob.com, but she thought she was visiting eve.com
- Potential issue: negatively impacts user experience

Question

Which of the following measures can help a user defending against CSRF attacks?

A. Accessing potentially malicious sites only with an incognito window
B. Accessing trusted sites only via HTTPS
C. All of the above
D. None of the above

Answer

Which of the following measures can help a user defending against CSRF attacks?

A. Accessing potentially malicious sites only with an incognito window

B. Accessing trusted sites only via HTTPS

C. All of the above

D. None of the above

What We Have Learned

- Motivation and specifications for session management
- Session ID implementations
  - Cookie
  - GET variable
  - POST variable
- Cross-Site Request Forgery (CSRF) attack
- CSRF mitigation techniques