Same-origin (cont'd) and Cookies

CS 161 Spring 2023 – March 8

Some content adapted from materials by David Wagner or Dan Boneh

Announcements

Midterm March 13th, 7-9pm

- Scope: Lectures 1-11, Homework 1-4, and Project 1
- Review sessions:
  - Memory Safety Review on Friday, March 10th from 1 to 2:30 PM PT at Wozniak Lounge
  - Cryptography Review on Thursday, March 9th from 5 to 7 PM PT at Soda 320

Same-origin policy

One origin should not be able to access the resources of another origin

Javascript on one page cannot read or modify pages from different origins

Origin

- Granularity of protection for same origin policy
- Origin = (protocol, hostname, port)

http://wikipedia.com:81/tools/info.html

protocol hostname port

- It is **string matching**! If these match, it is same origin, else it is not. Even though in some cases, it is logically the same origin, if there is no match, it is not

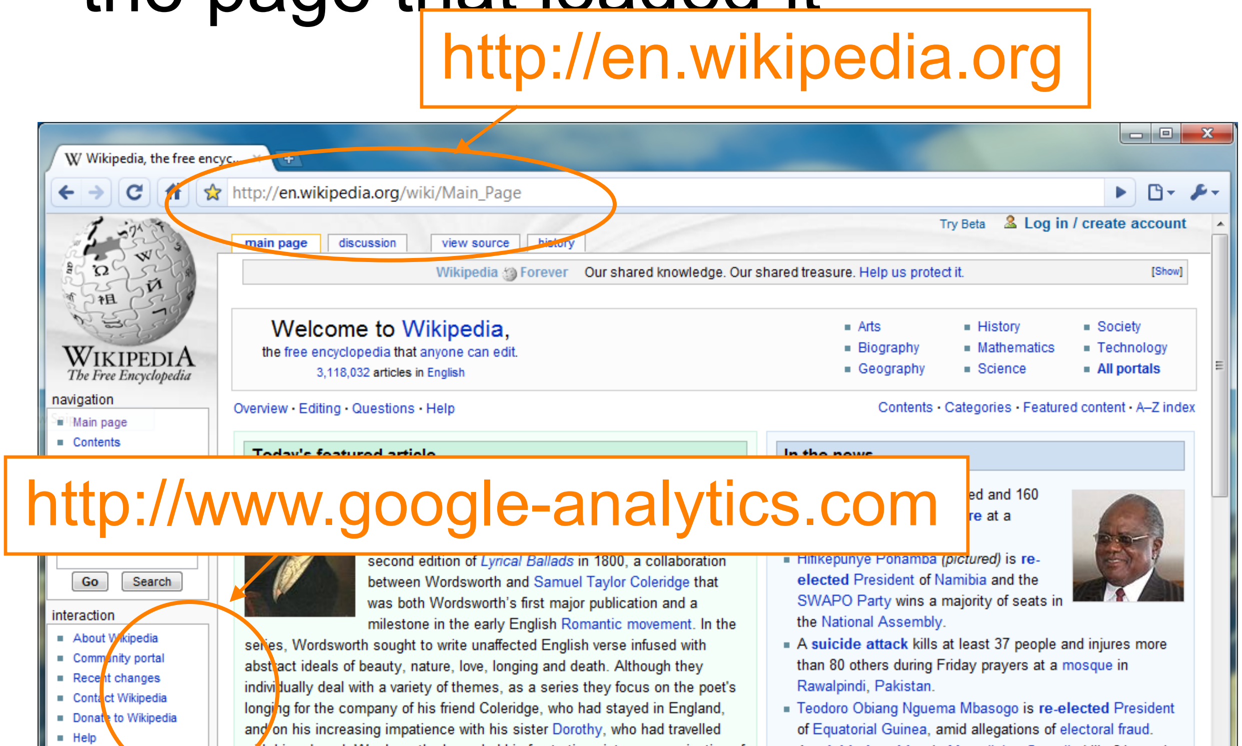
Same-origin policy

- The origin of a page is derived from the URL it was loaded from

http://en.wikipedia.org

Same-origin policy

- The origin of a page is derived from the URL it was loaded from
- Special case: Javascript runs with the origin of the page that loaded it

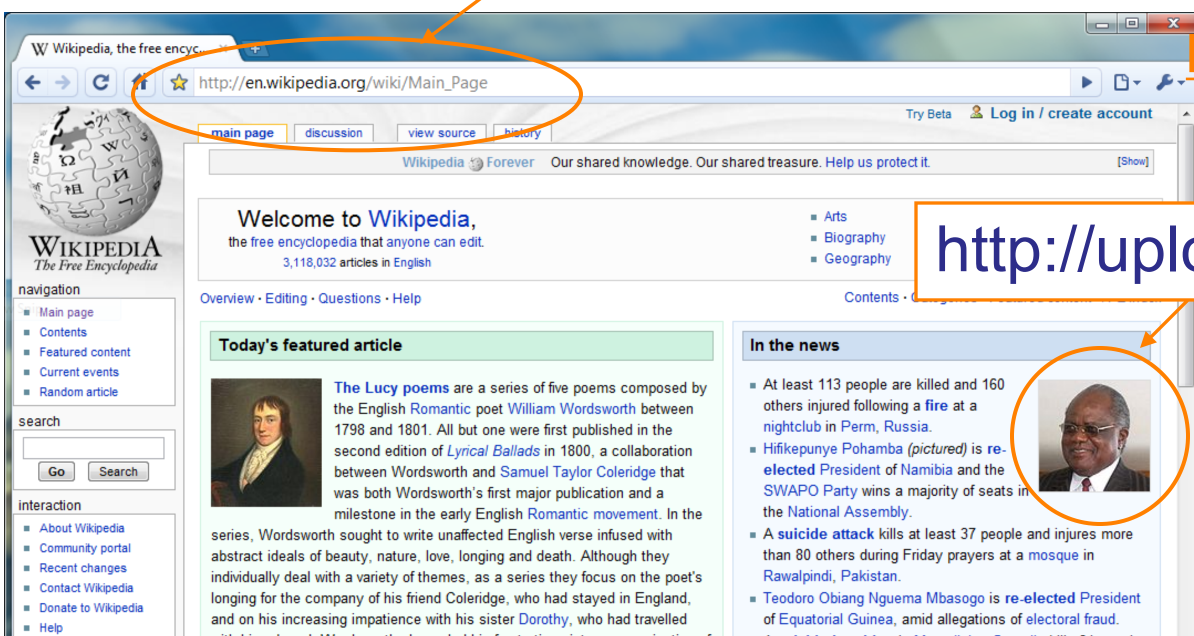


Origins of other components

- **<img src=“”>** the image is “copied” from the remote server into the new page so it has the origin of the embedding page (like JS) and not of the remote origin

http://en.wikipedia.org

Image has http://en.wikipedia.org origin



http://upload.wikimedia.org

Origins of other components

- iframe: origin of the URL from which the iframe is served, and not the loading website.

Origins of other components

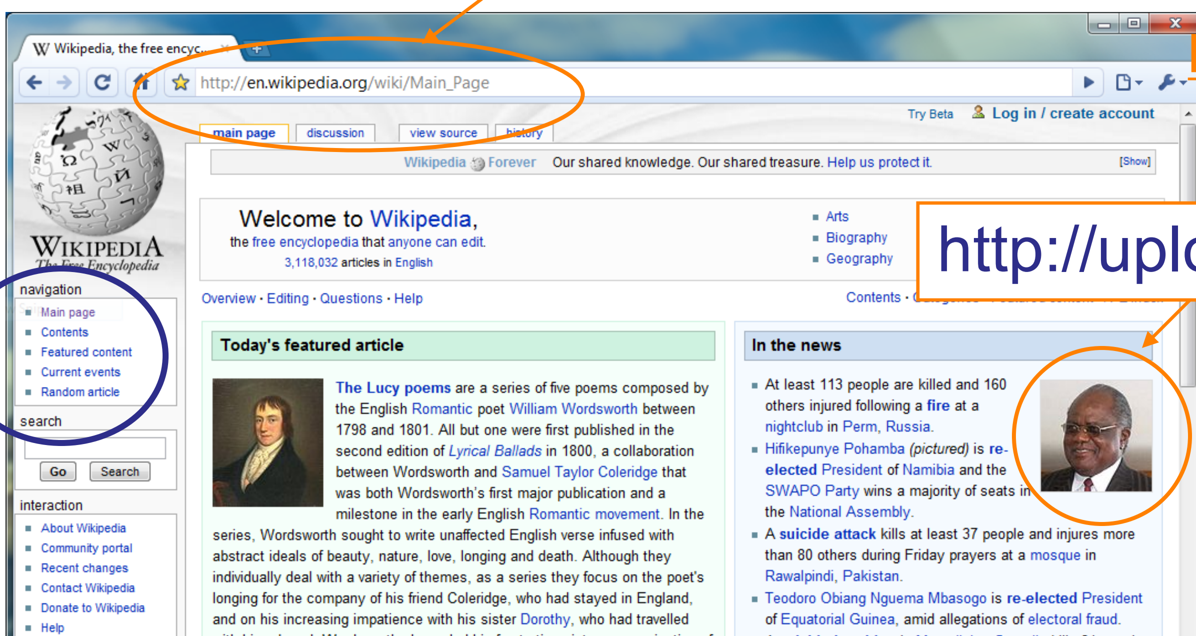
Assume this is an iframe to http://upload.wikimedia.org containing Javascript. Can the javascript on this page modify the photo?

No, because:

http://en.wikipedia.org

Image has http://en.wikipedia.org origin

http://upload.wikimedia.org



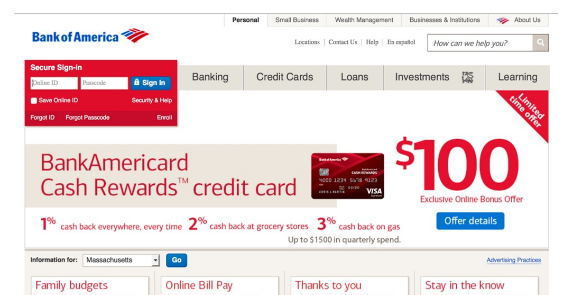
Exercises: Same origin?

Originating document	**Accessed document**
http://wikipedia.org/**a**/	http://wikipedia.org/**b**/
http://wikipedia.org/	http://**www.**wikipedia.org/
http://wikipedia.org/	**https**://wikipedia.org/
http://wikipedia.org**:81**/	http://wikipedia.org**:82**/

Cross-origin communication

- Allowed through a narrow API: **postMessage**
- Receiving origin decides if to accept the message based on origin (whose correctness is enforced by browser)


```
postMessage
("run this
script",
script)
```



Check origin, and request!

Cookies

CS 161 Spring 2023

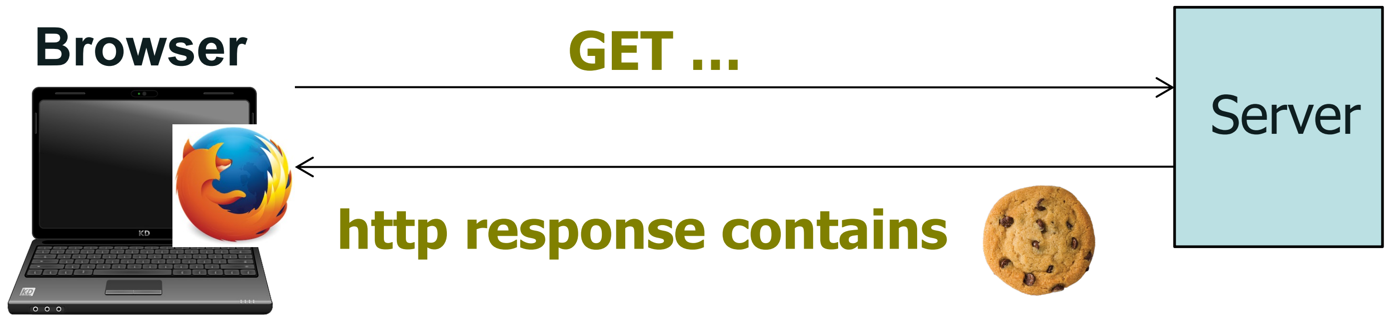
Some content adapted from materials by David Wagner or Dan Boneh

Cookies

- HTTP is largely stateless
- Cookies are a way to add state. This state helps link the same user's requests and helps customize websites for the user

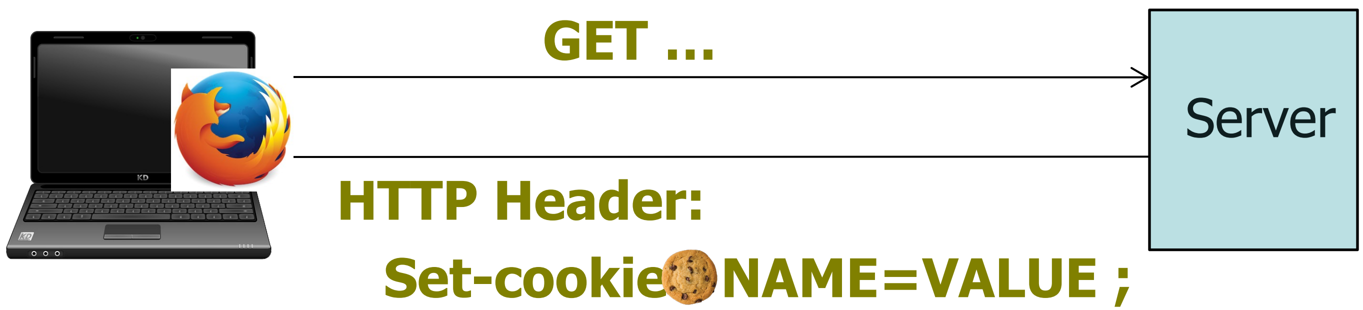
Cookies

A way of maintaining state in the browser



Browser maintains cookie jar with all cookies it receives

Setting/deleting cookies by server



- The first time a browser connects to a particular web server, it has no cookies for that web server
- When the web server responds, it includes a **Set-Cookie:** header that defines a cookie
- Each cookie is just a name-value pair (with some extra metadata)

View a cookie

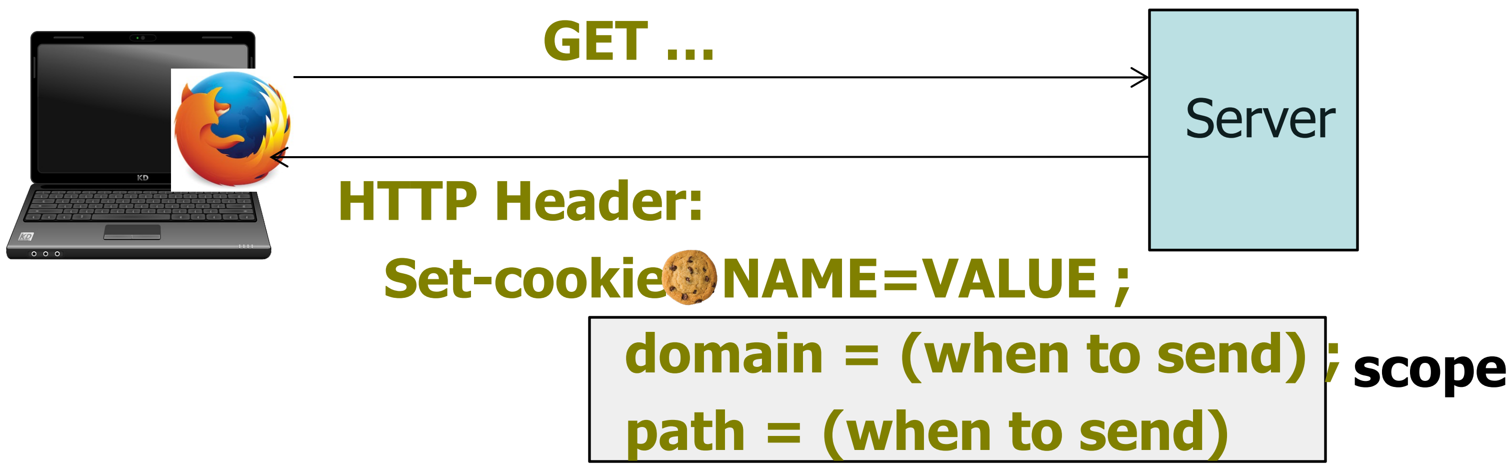
In a web console (chrome, view->developer->developer tools), type

document.cookie

to see the cookie for that site

Each name=value is one cookie.
document.cookie lists all cookies in scope for document

Cookie scope



- When the browser connects to the same server later, it automatically attaches the cookies in scope: header containing the name and value, which the server can use to connect related requests.
- Domain and path inform the browser about which sites to send this cookie to

Cookie scope

GET …

Server

HTTP Header:

Set-cookie: NAME=VALUE ;

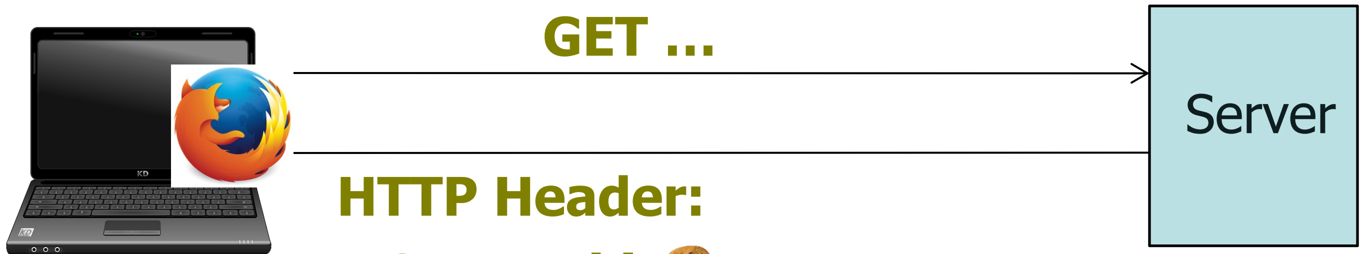
domain = (when to send) ;

path = (when to send)

secure = (only send over HTTPS);

- **Secure: sent over https only**
  - **https provides secure communication using TLS (encryption and authentication)**

Cookie scope



Set-cookie: NAME=VALUE ;

domain = (when to send) ; **scope**

path = (when to send)

secure = (only send over SSL);

expires = (when expires) ;

HttpOnly

- **Expires is expiration date**
  - Delete cookie by setting "expires" to date in past
- **HttpOnly:** cookie cannot be accessed by Javascript, but only sent by browser

Cookie policy

The cookie policy has two parts:

1. What scopes a URL-host name web server is allowed to set on a cookie
2. When the browser sends a cookie to a URL

Cookie scope

- Scope of cookie might not be the same as the URL-host name of the web server setting it

What scope a server may set for a cookie

The browser checks if the web server may set the cookie, and if not, it will not accept the cookie.

domain: any domain-suffix of URL-hostname, except TLD

[top-level domains, e.g. '.com']

example: host = "login.site.com"

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

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

Problematic for sites like .berkeley.edu

path: can be set to anything

Examples

Web server at **foo.example.com** wants to set cookie with domain:

domain	Whether it will be set
(value omitted)	*foo.example.com* (exact)
bar.foo.example.com	
foo.example.com	
baz.example.com	
example.com	yes
ample.com	
.com	

Credits: The Tangled Web: A Guide to Securing Modern Web Applications, by Michał Zalewski

When browser sends cookie

Goal: server only sees cookies in its scope

Browser sends all cookies <u>in URL scope</u>:

- cookie-domain is domain-suffix of URL-domain, and
- cookie-path is prefix of URL-path, and
- [protocol=HTTPS if cookie is “secure”]

When browser sends cookie

A cookie with

domain = example.com, and

path = /some/path/

will be included on a request to

http://foo.example.com/some/path/subdirectory/hello.txt

Examples: Which cookie will be sent?

cookie 1
name = userid
value = u1
domain = login.site.com
path = /
non-secure

cookie 2
name = userid
value = u2
domain = .site.com
path = /
non-secure

http://checkout.site.com/	cookie: userid=u2
http://login.site.com/	cookie: userid=u1, userid=u2
http://othersite.com/	cookie: none

Examples

Web server at foo.example.com wants to set cookie with domain:

domain	Whether it will be set, and if so, where it will be sent to
(value omitted)	*foo.example.com* (exact) ?
bar.foo.example.com	Cookie not set: domain more specific than origin
foo.example.com	?
baz.example.com	Cookie not set: domain mismatch
example.com	?
ample.com	Cookie not set: domain mismatch
.com	Cookie not set: domain too broad, security risk

Credits: The Tangled Web: A Guide to Securing Modern Web Applications, by Michał Zalewski

Examples

Web server at foo.example.com wants to set cookie with domain:

domain	Whether it will be set, and if so, where it will be sent to	
(value omitted)	*foo.example.com* (exact)	**.foo.example.com*
bar.foo.example.com	Cookie not set: domain more specific than origin	
foo.example.com	**.foo.example.com*	
baz.example.com	Cookie not set: domain mismatch	
example.com	**.example.com*	
ample.com	Cookie not set: domain mismatch	
.com	Cookie not set: domain too broad, security risk	

Credits: The Tangled Web: A Guide to Securing Modern Web Applications, by Michał Zalewski

Examples

<u>cookie 1</u> **name = userid** **value = u1** **domain = login.site.com** **path = /** **secure**	**<u>cookie 2</u>** **name = userid** **value = u2** **domain = .site.com** **path = /** **non-secure**

http://checkout.site.com/ **cookie: userid=u2**

http://login.site.com/ **cookie: userid=u2**

http**s**://login.site.com/ **cookie: userid=u1; userid=u2**

(arbitrary order)

Client side read/write: document.cookie

- Setting a cookie in Javascript:

  document.cookie = “name=value; expires=…; ”

- Reading a cookie: alert(document.cookie)

  prints string containing all cookies available for document (based on [protocol], domain, path)

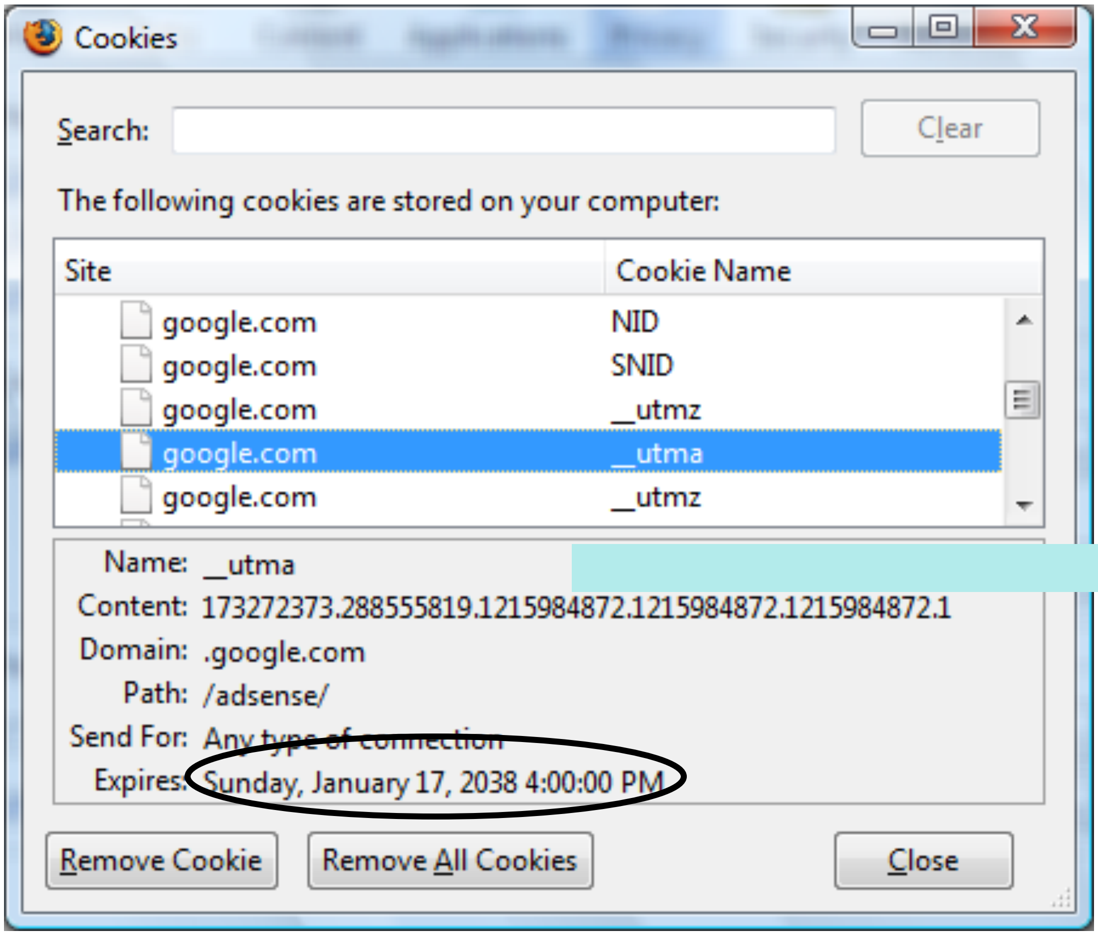
- Deleting a cookie:

  document.cookie = “name=; expires= Thu, 01-Jan-00”

document.cookie often used to customize page in Javascript

Viewing/deleting cookies in Browser UI

Firefox: Tools -> page info -> security -> view cookies



Cookie policy versus same-origin policy

Cookie policy versus same-origin policy

- Consider Javascript on a page loaded from a URL U
- If a cookie is in scope for a URL U, it can be accessed by Javascript loaded on the page with URL U,

  unless the cookie has the httpOnly flag set.

So there isn't exact domain match as in same-origin policy, but the cookie policy is invoked instead.

Examples

cookie 1
name = userid
value = u1
domain = login.site.com
path = /
non-secure

cookie 2
name = userid
value = u2
domain = .site.com
path = /
non-secure

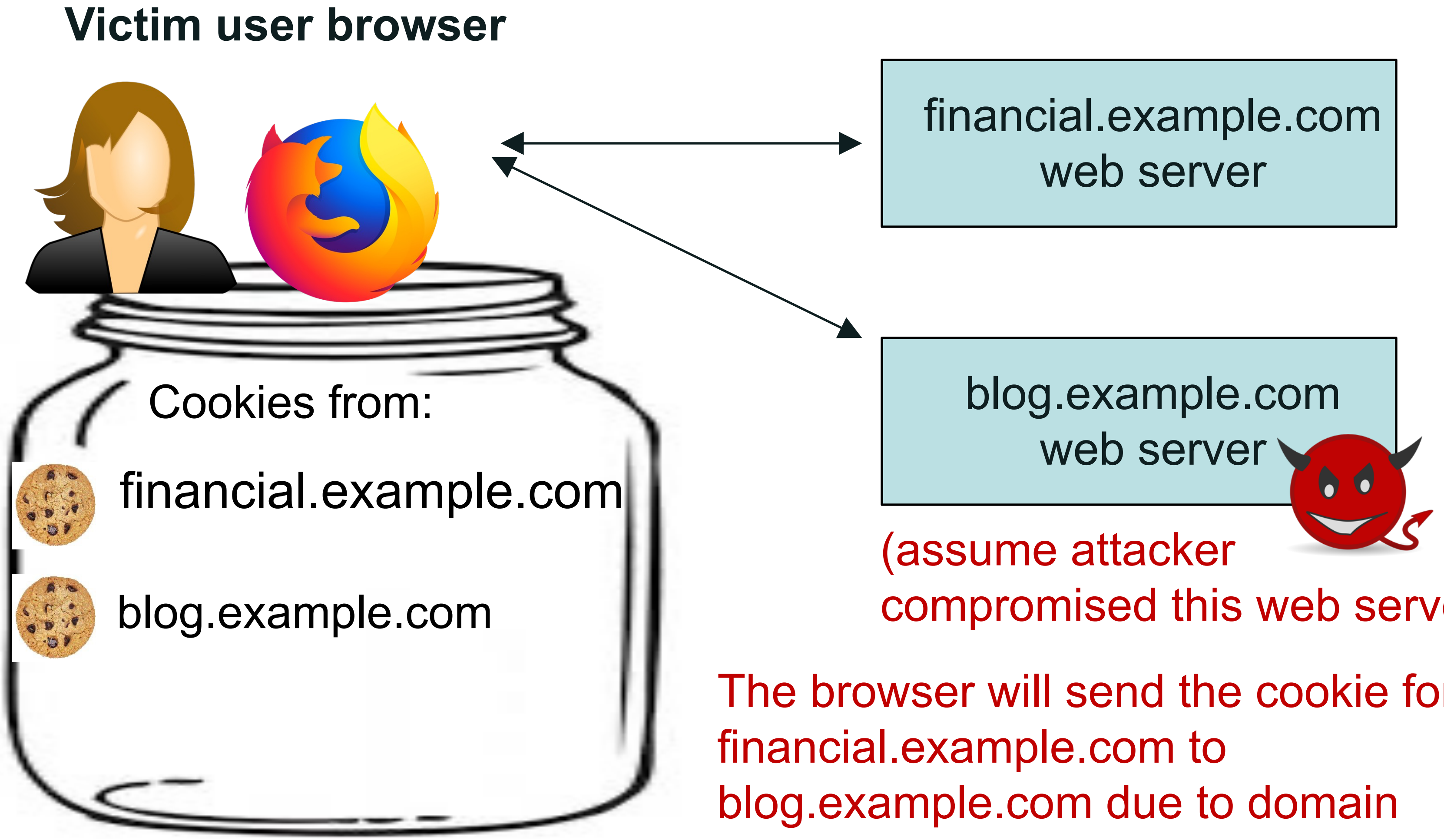
http://checkout.site.com/	**cookie: userid=u2**
http://login.site.com/	**cookie: userid=u1, userid=u2**
http://othersite.com/	**cookie: none**

JS on each of these URLs can access the corresponding cookies even if the domains are not the same

Indirectly bypassing same-origin policy using cookie policy

- Since the cookie policy and the same-origin policy are different, there are corner cases when one can use cookie policy to bypass same-origin policy
- Ideas how?

Example



cookies in jar with domain example.com

The browser will send the cookie for financial.example.com to blog.example.com due to domain

RFC6265

- For further details on cookies, checkout the standard RFC6265 “HTTP State Management Mechanism”

https://tools.ietf.org/html/rfc6265

- Browsers are expected to implement this reference, and any differences are browser specific

Web security attacks

What can go bad if a web server is compromised?

- Steal sensitive data (e.g., data from many users)
- Change server data (e.g., affect users)
- Gateway to enabling attacks on clients
- Impersonation (of users to servers, or vice versa)
- Others

A set of common attacks

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- XSS – Cross-site scripting
  - Attacker inserts client-side script into pages viewed by other users, script runs in the users' browsers
- CSRF – Cross-site request forgery
  - Bad web site sends request to good web site, using credentials of an innocent victim who "visits" site

Session Authentication

- **Session**: A sequence of requests and responses associated with the same authenticated user
    - Example: When you check all your unread emails, you make many requests to Gmail. The Gmail server needs a way to know all these requests are from you
    - When the session is over (you log out, or the session expires), future requests are not associated with you
- Naïve solution: Type your username and password before each request
    - Problem: Very inconvenient for the user!
- Better solution: Is there a way the browser can automatically send some information in a request for us?
    - Yes: Cookies!

Session Authentication: Intuition

- Imagine you're attending a concert
- The first time you enter the venue:
  - Present your ticket and ID
  - The doorperson checks your ticket and ID
  - If they're valid, you receive a wristband
- If you leave and want to re-enter later
  - Just show your wristband!
  - No need to present your ticket and ID again

Session Tokens

- **Session token**: A secret value used to associate requests with an authenticated user
- The first time you visit the website:
  - Present your username and password
  - If they're valid, you receive a session token
  - The server associates you with the session token
- When you make future requests to the website:
  - Attach the session token in your request
  - The server checks the session token to figure out that the request is from you
  - No need to re-enter your username and password!

Session Tokens with Cookies

- Session tokens can be implemented with cookies
  - Cookies can be used to save *any* state across requests (e.g. dark mode)
  - Session tokens are just one way to use cookies
- The first time you visit a website:
  - Make a request with your username and password
  - If they're valid, the server sends you a cookie with the session token
  - The server associates you with the session token
- When you make future requests to the website:
  - The browser attaches the session token cookie in your request
  - The server checks the session token to figure out that the request is from you
  - No need to re-enter your username and password!
- When you log out (or when the session times out):
  - The browser and server delete the session token

Session Tokens: Security

- If an attacker steals your session token, they can log in as you!
  - The attacker can make requests and attach your session token
  - The browser will think the attacker's requests come from you
- Servers need to generate session tokens *randomly* and *securely*
- Browsers need to make sure malicious websites cannot steal session tokens
  - Enforce isolation with cookie policy and same-origin policy
- Browsers should not send session tokens to the wrong websites
  - Enforced by cookie policy

Session Token Cookie Attributes

- What attributes should the server set for the session token?
    - Domain and Path: Set so that the cookie is only sent on requests that require authentication
    - Secure: Can set to True to so the cookie is only sent over secure HTTPS connections
    - HttpOnly: Can set to True so JavaScript can’t access session tokens
    - Expires: Set so that the cookie expires when the session times out

Name	`token`
Value	`{random value}`
Domain	`mail.google.com`
Path	`/`
Secure	`True`
HttpOnly	`True`
Expires	`{15 minutes later}`
(*other fields omitted*)	

Cross-Site Request Forgery (CSRF)

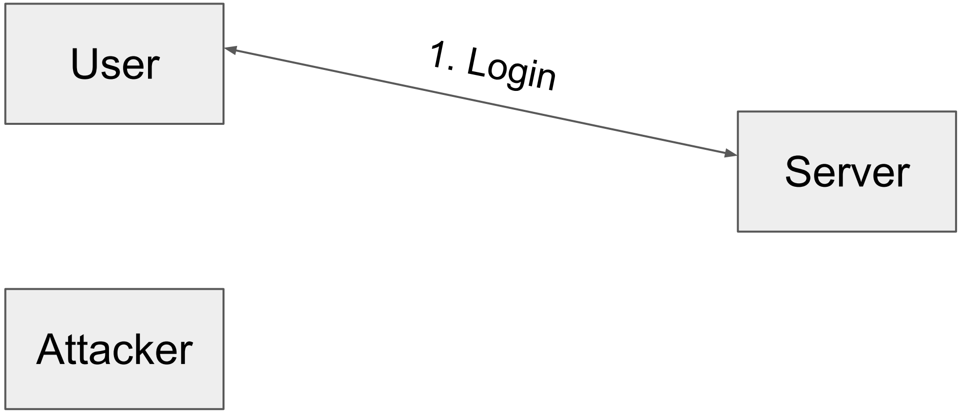
Cross-Site Request Forgery (CSRF)

- Idea: What if the attacker tricks the victim into making an unintended request?
  - The victim’s browser will automatically attach relevant cookies
  - The server will think the request came from the victim!
- **Cross-site request forgery (CSRF or XSRF)**: An attack that exploits cookie-based authentication to perform an action as the victim

Steps of a CSRF Attack

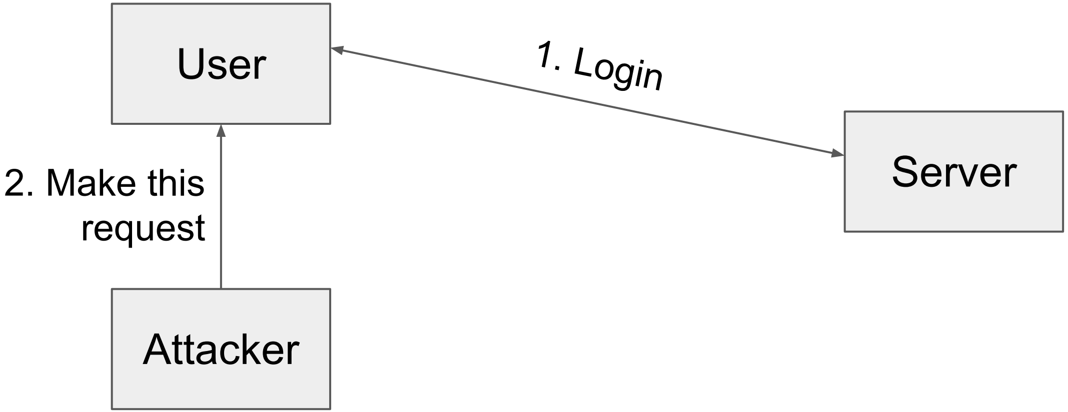
Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token



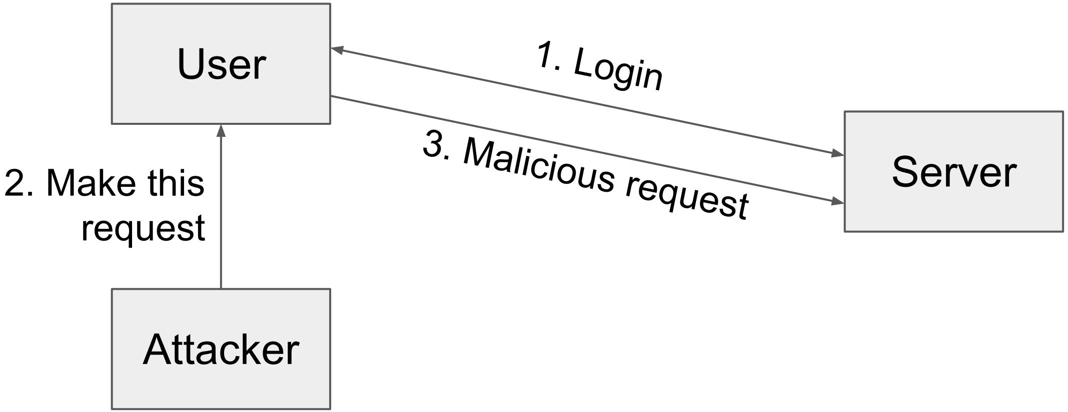
Steps of a CSRF Attack

1. User authenticates to the server
   - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server



Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server
3. The server accepts the malicious request from the victim
    - Recall: The cookie is automatically attached in the request



Steps of a CSRF Attack

1. User authenticates to the server
    - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server
3. The server accepts the malicious request from the victim
    - Recall: The cookie is automatically attached in the request

Executing a CSRF Attack

- How might we trick the victim into making a GET request?
- Strategy #1: Trick the victim into clicking a link
  - Later we'll see how to trick a victim into clicking a link
  - The link can directly make a GET request:
    `https://www.bank.com/transfer?amount=100&to=Mallory`
  - The link can open an attacker's website, which contains some JavaScript that makes the actual malicious request
- Strategy #2: Put some HTML on a website the victim will visit
  - Example: The victim will visit a forum. Make a post with some HTML on the forum
  - HTML to automatically make a GET request to a URL:
    `<img src="https://www.bank.com/transfer?amount=100&to=Mallory">`
    - This HTML will probably return an error or a blank 1 pixel by 1 pixel image, but the GET request will still be sent...with the relevant cookies!

Executing a CSRF Attack

- How might we trick the victim into making a POST request?
  - Example POST request: Submitting a form
- Strategy #1: Trick the victim into clicking a link
  - Note: Clicking a link in your browser makes a GET request, not a POST request, so the link cannot directly make the malicious POST request
  - The link can open an attacker's website, which contains some JavaScript that makes the actual malicious POST request
- Strategy #2: Put some JavaScript on a website the victim will visit
  - Example: Pay for an advertisement on the website, and put JavaScript in the ad
  - Recall: JavaScript can make a POST request

Top 25 Most Dangerous Software Weaknesses (2020)

Rank	ID	Name	Score
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
[2]	CWE-787	Out-of-bounds Write	46.17
[3]	CWE-20	Improper Input Validation	33.47
[4]	CWE-125	Out-of-bounds Read	26.50
[5]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
[7]	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
[8]	CWE-416	Use After Free	18.87
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
[10]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
[11]	CWE-190	Integer Overflow or Wraparound	15.81
[12]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
[13]	CWE-476	NULL Pointer Dereference	8.35
[14]	CWE-287	Improper Authentication	8.17
[15]	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
[16]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
[17]	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53

CSRF Example: YouTube

- 2008: Attackers exploit a CSRF vulnerability on YouTube
- By forcing the victim to make a request, the attacker could:
  - Add any videos to the victim's "Favorites"
  - Add any user to the victim's "Friend" or "Family" list
  - Send arbitrary messages as the victim
  - Make the victim flag any videos as inappropriate
  - Make the victim share a video with their contacts
  - Make the victim subscribe to any channel
  - Add any videos to the user's watchlist
- **Takeaway**: With a CSRF attack, the attacker can force the victim to perform a wide variety of actions!

CSRF Example: Facebook

InternetNews.com

Link

Facebook Hit by Cross-Site Request Forgery Attack

Sean Michael Kerner *August 21, 2009*

Nevertheless, that Facebook accounts were compromised in the wild is noteworthy because the attack used a legitimate HTML tag to violate users' privacy.

According to Zilberman's disclosure, the attack simply involved the malicious HTML image tag residing on any site, including any blog or forum that permits the use of image tags even in the comments section.

"The attack elegantly ends with a valid image so the page renders normally, and the attacked user does not notice that anything peculiar has happened," Zilberman said.

Takeaway: The HTML image tag can be used to execute a CSRF attack

CSRF Defenses

CSRF Defenses

- CSRF defenses are implemented by the server (not the browser)
- Defense: CSRF tokens
- Defense: Referer validation
- Defense: SameSite cookie attribute

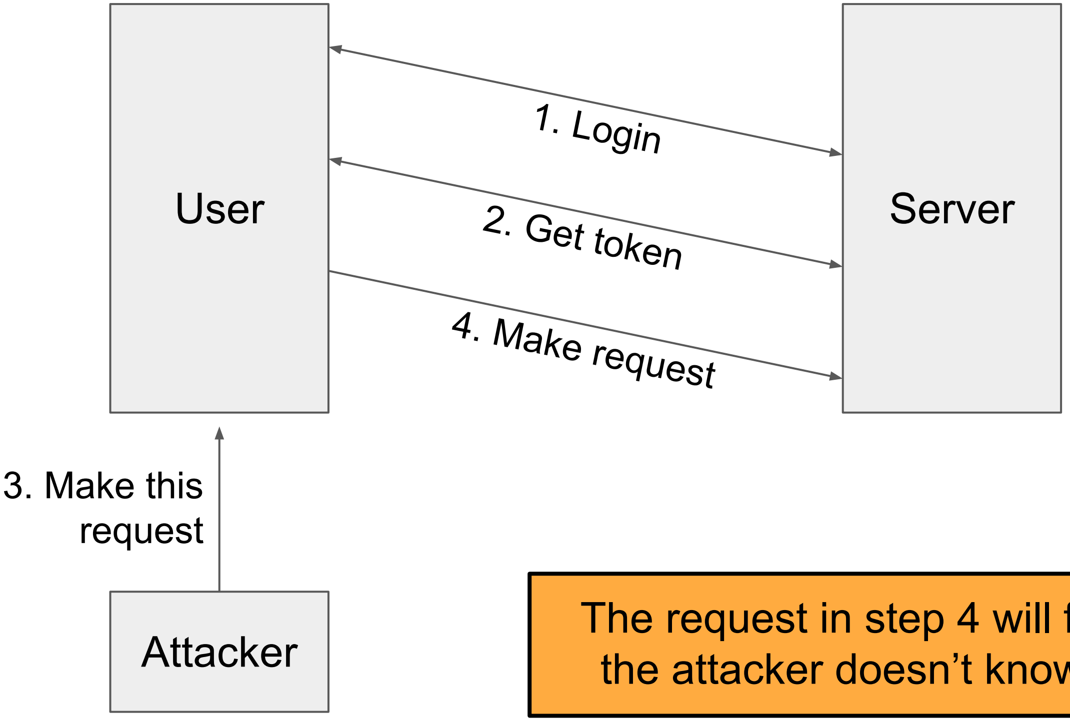
CSRF Tokens

- Idea: Add a secret value in the request that the attacker doesn't know
  - The server only accepts requests if it has a valid secret
  - Now, the attacker can't create a malicious request without knowing the secret
- **CSRF token**: A secret value provided by the server to the user. The user must attach the same value in the request for the server to accept the request.
  - CSRF tokens cannot be sent to the server in a cookie!
    - The token must be sent somewhere else (e.g. a header, GET parameter, or POST content)
  - CSRF tokens are usually valid for only one or two requests

CSRF Tokens: Usage

- Example: HTML forms
  - Forms are vulnerable to CSRF
    - If the victim visits the attacker's page, the attacker's JavaScript can make a POST request with a filled-out form
- CSRF tokens are a defense against this attack
  - Every time the user requests a form from the legitimate website, the server attaches a CSRF token as a *hidden form field* (in the HTML, but not visible to the user)
  - When the user submits the form, the form contains the CSRF token
  - The attacker's JavaScript won't be able to create a valid form, because they don't know the CSRF token!
  - The attacker can try to fetch their own CSRF token, but it will only be valid for the attacker, not the victim

CSRF Tokens: Usage



The request in step 4 will fail, because the attacker doesn't know the token!

Referer Header

- Idea: In a CSRF attack, the victim usually makes the malicious request from a different website
- Referer header: A header in an HTTP request that indicates which webpage made the request
  - “Referer” is a 30-year typo in the HTTP standard (supposed to be “Referrer”)!
  - Example: If you type your username and password into the Facebook homepage, the Referer header for that request is **`https://www.facebook.com`**
  - Example: If an **`img`** HTML tag on a forum forces your browser to make a request, the Referer header for that request is the forum’s URL
  - Example: If JavaScript on an attacker’s website forces your browser to make a request, the Referer header for that request is the attacker’s URL

Referer Header

- Checking the Referer header
  - Allow **same-site requests**: The Referer header matches an expected URL
    - Example: For a login request, expect it to come from **`https://bank.com/login`**
  - Disallow **cross-site requests**: The Referer header does not match an expected URL
- If the server sees a cross-site request, reject it

Referer Header: Issues

- The Referer header may leak private information
  - Example: If you made the request on a top-secret website, the Referer header might show you visited **`http://intranet.corp.apple.com/projects/iphone/competitors.html`**
  - Example: If you make a request to an advertiser, the Referer header gives the advertiser information about how you saw the ad
- The Referer header might be removed before the request reaches the server
  - Example: Your company firewall removes the header before sending the request
  - Example: The browser removes the header because of your privacy settings
- The Referer header is optional. What if the request leaves the header blank?
  - Allow requests without a header?
    - Less secure: CSRF attacks might be possible
  - Deny requests without a header?
    - Less usable: Legitimate requests might be denied
  - Need to consider fail-safe defaults: No clear answer

SameSite Cookie Attribute

- Idea: Implement a flag on a cookie that makes it unexploitable by CSRF attacks
  - This flag must specify that **cross-site** requests will not contain the cookie
- **SameSite flag**: A flag on a cookie that specifies it should be sent only when the domain of the cookie **exactly** matches the domain of the origin
  - SameSite=None: No effect
  - SameSite=Strict: The cookie will not be sent if the cookie domain does not match the origin domain
  - Example: If **`https://evil.com/`** causes your browser to make a request to **`https://bank.com/transfer?to=mallory`**, cookies for bank.com will not be sent if SameSite=Strict, because the origin domain (**`evil.com`**) and cookie domain (**`bank.com`**) are different
- Issue: Not yet implemented on all browsers

Cookies: Summary

- Cookie: a piece of data used to maintain state across multiple requests
  - Set by the browser or server
  - Stored by the browser
  - Attributes: Name, value, domain, path, secure, HttpOnly, expires
- Cookie policy
  - Server with domain X can set a cookie with domain attribute Y if the domain attribute is a **domain suffix** of the server's domain, and the domain attribute Y is not a top-level domain (TLD)
  - The browser attaches a cookie on a request if the domain attribute is a **domain suffix** of the server's domain, and the path attribute is a **prefix** of the server's path

Session Authentication: Summary

- Session authentication
  - Use cookies to associate requests with an authenticated user
  - First request: Enter username and password, receive session token (as a cookie)
  - Future requests: Browser automatically attaches the session token cookie
- Session tokens
  - If an attacker steals your session token, they can log in as you
  - Should be randomly and securely generated by the server
  - The browser should not send tokens to the wrong place

CSRF: Summary

- Cross-site request forgery (CSRF or XSRF): An attack that exploits cookie-based authentication to perform an action as the victim
    - User authenticates to the server
        - User receives a cookie with a valid session token
    - Attacker tricks the victim into making a malicious request to the server
    - The server accepts the malicious request from the victim
        - Recall: The cookie is automatically attached in the request
- Attacker must trick the victim into creating a request
    - GET request: click on a link
    - POST request: use JavaScript

CSRF Defenses: Summary

- CSRF token: A secret value provided by the server to the user. The user must attach the same value in the request for the server to accept the request.
  - The attacker does not know the token when tricking the user into making a request
- Referer Header: Allow same-site requests, but disallow cross-site requests
  - Header may be blank or removed for privacy reasons
- Same-site cookie attribute: The cookie is sent only when the domain of the cookie exactly matches the domain of the origin
  - Not implemented on all browsers