IERG 4130 – 2026 Fall

香港中文大學
CUHK

Cookies and CSRF

Yajin ZHOU

https://yajin.org

Credits: the slides are from CS161 of UC Berkeley

Customizing HTTP Responses



- **HTTP is a request-response protocol**
  - The web server processes each request independently of other requests
- **What if we want our responses to be customized?**
  - Example: If I enable dark mode on a website, I want future responses from the website to be in dark mode
  - Example: If I log in to a website, I want future responses from the website to be related to my account

Cookies: Definition



- **Cookie: a piece of data used to maintain state across multiple requests**
- This data is stored and remembered by the browser, and sent to server when needed
- **Creating cookies**
  - The server can create a cookie by including a **Set-Cookie** header in its response
  - JavaScript running in the browser can create a cookie
- **Storing cookies**
  - Cookies are stored by the web browser (not the web server)
  - The browser's cookie storage is sometimes called a **cookie jar**
- **Sending cookies**
  - The browser *automatically* attaches relevant cookies to every request
  - The server uses received cookies to customize responses and connect related requests

Parts of a Cookie: Name and Value



- **The actual data in the cookie is stored as a name-value pair**

Name	SESSIONID
Value	298zf09hf012fh2
Cookie: SESSIONID=298zf09hf012fh2;	
Domain	toon.cs161.org
Path	/xorcist
Secure	True
HttpOnly	False
Expires	11 Jul 2024 20:00:00
(other fields omitted)	

Parts of a Cookie: Domain and Path



- The **domain attribute** and **path attribute** define which requests the browser should attach this cookie for
- **The domain attribute usually looks like the domain in a URL**
- **The path attribute usually looks like a path in a URL**

Name	SESSIONID
Value	298zf09hf012fh2
Cookie: SESSIONID=298zf09hf012fh2;	
Domain	toon.cs161.org
Path	/xorcist
Secure	True
HttpOnly	False
Expires	11 Jul 2024 20:00:00
(other fields omitted)	

Parts of a Cookie: Secure and HttpOnly



- The Secure attribute and HttpOnly attribute restrict the cookie for security purposes
- If the **Secure attribute** is True, then the browser only sends the cookie if the request is made over HTTPS (and doesn't send the cookie with HTTP requests)
- If the **HttpOnly attribute** is True, then JavaScript in the browser is not allowed to access the cookie

Name	SESSIONID
Value	298zf09hf012fh2
Cookie: SESSIONID=298zf09hf012fh2;	
Domain	toon.cs161.org
Path	/xorcist
Secure	True
HttpOnly	False
Expires	11 Jul 2024 20:00:00
(other fields omitted)	

Parts of a Cookie: Expires



- **The Expires attribute defines when the cookie is no longer valid**
- **The expires attribute is usually a timestamp**
- If the timestamp is in the past, then the cookie has expired, and the browser deletes it from the cookie jar

Name	SESSIONID
Value	298zf09hf012fh2
Cookie: SESSIONID=298zf09hf012fh2;	
Domain	toon.cs161.org
Path	/xorcist
Secure	True
HttpOnly	False
Expires	11 Jul 2024 20:00:00
(other fields omitted)	

Cookies: Issues



- **Recall:**
  - The server can create a cookie by including a **Set-Cookie** header in its response
  - The browser automatically attaches relevant cookies in every request
- **Security issues:**
  - A server should not be able to set cookies for unrelated websites
    - Example: **evil.com** should not be able to set a cookie that gets sent to **google.com**
  - Cookies shouldn’t be sent to the wrong websites
    - Example: A cookie used for authenticating a user to Google should not be sent to **evil.com**
    - We’ll see how cookies are used for logins later

Cookie Policy

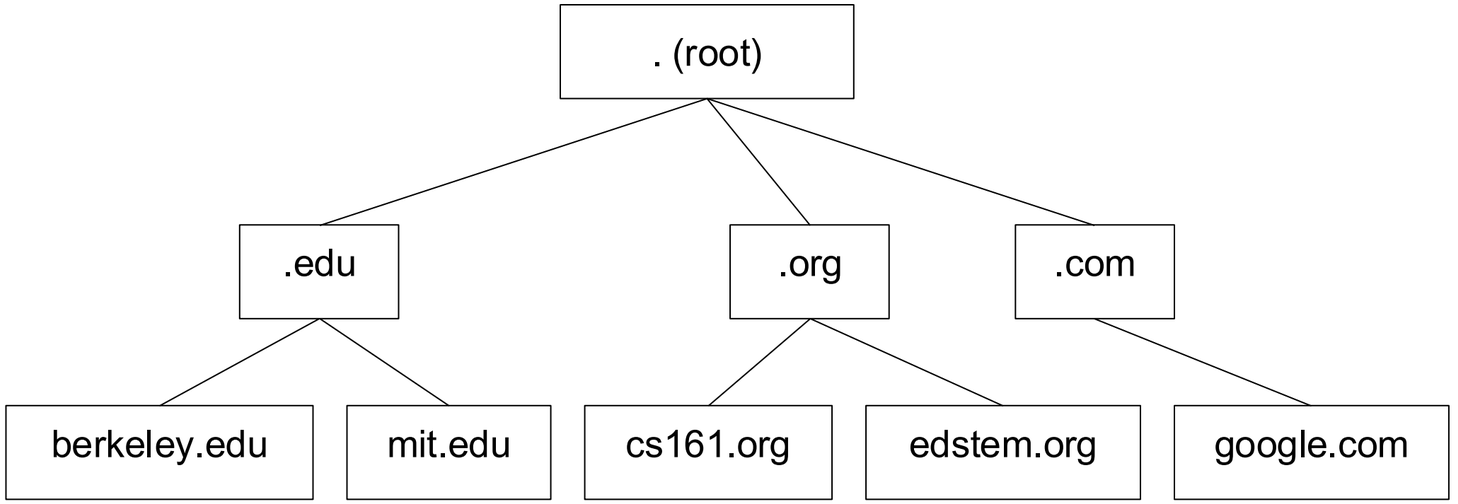


- **Cookie policy: A set of rules enforced by the browser**
  - When the browser receives a cookie from a server, should the cookie be accepted?
  - When the browser makes a request to a server, should the cookie be attached?
- **Cookie policy is (confusingly) different from the same-origin policy**

Domain Hierarchy



- **Recall: Domains**
  - Located after the double slashes, but before the next single slash
  - Written as several phrases separated by dots
- **Domains can be sorted into a hierarchy**
  - The hierarchy is separated by dots



Cookie Policy: Setting Cookies



- **When the browser receives a cookie from a server, should the cookie be accepted?**
- **Server with** domain X **can set a cookie whose** domain is Y **if**
  - (1) Y equals with X or Y is the parent of X
  - ... and, (2) the domain attribute Y is not a top-level domain (TLD)
  - **No restrictions for the Path attribute (the browser will accept any value)**
- **Examples:**
  - mail.**google.com** can set cookies for Domain=mail.google.com or Domain=google.com
  - **google.com** can set cookies for Domain=google.com
  - google.**com cannot** set cookies for Domain=com, because com is a top-level domain

Cookie Policy: Sending Cookies



- **When the browser makes a request to a server, should the cookie be attached?**
- **The browser sends the cookie if both of these are true:**
  - The domain attribute is a **domain suffix** of the server's domain
  - The path attribute is a **prefix** of the server's path

Intuition: Domain attributes



- If the domain attribute is set to **google.com**, think of this as representing

  google.com or ***.google.com** or ***.*.google.com** or …

- A server can only set a cookie if the server matches the cookie's domain attribute
- **A cookie is only sent to a server if the server matches the domain attribute**

Cookie Policy: Sending Cookies

(server URL)

https://toon.cs161.org/cryptoverse/oneshots/subway.html

cs161.org/cryptoverse

(cookie domain) (cookie path)

Quick method to check cookie sending:

- Right-align the domains. The cookie domain should be a suffix of the server URL.

Cookie Policy: Sending Cookies



(server URL)

https://toon.cs161.org/cryptoverse/oneshots/subway.html

cs161.org/cryptoverse

(cookie domain) (cookie path)

Quick method to check cookie sending:

- Right-align the domains. The cookie domain should be a suffix of the server URL.
- Left-align the paths. The cookie domain should be a prefix of the server path.

Cookie Policy: Sending Cookies

(server URL)

https://toon.cs161.org/cryptoverse/oneshots/subway.html

cs161.org/cryptoverse

(cookie domain) (cookie path)

Quick method to check cookie sending:

- Right-align the domains. The cookie domain should be a suffix of the server URL.
- Left-align the paths. The cookie domain should be a prefix of the server path.

If the domains and paths all match, then the cookie is sent.

Cookie Policy: Sending Cookies



(server URL)

https://toon.cs161.org/cryptoverse/oneshots/subway.html

cs161.org/xorcist

(cookie domain) (cookie path)

Quick method to check cookie sending:

- Right-align the domains. The cookie domain should be a suffix of the server URL.
- Left-align the paths. The cookie domain should be a prefix of the server path.

If the domain or path doesn't match, then the cookie is not sent.

Cookie Policy: Sending Cookies



(server URL)

https://su24.cs161.org/proj2/story

www.cs161.org/proj2

(cookie domain) (cookie path)

Will a cookie with this domain and path be sent to this server?

Cookie Policy: Sending Cookies



(server URL)

https://su24.cs161.org/proj2/story
www.cs161.org/proj2

(cookie domain) (cookie path)

Will a cookie with this domain and path be sent to this server?

No! The domain does not match!

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 (common) 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

The Basic Idea: Intuition (A Close Approximation)



- **Server sends the browser a cookie**
- The cookie is attached to every subsequent request to the same server that originally set the cookie

- The cookie contains a random, unique value that identifies the user: e.g., sessionid=09c3ad..9f
- The browser remembers the domain of the server that set the cookie, and sends this value along with every request to that domain
- **The server remembers the association sessionid=09c3ad..9f ⇄ userid=daw**
- **This way, the server can associate each request to the user who is logged in**

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 server 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)



- **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

Analogy: Passing Notes to My Banker



- **I have a bank account with a local bank, with $1000 in my checking account.**
- I can pay someone at any time by writing out a check (a note that is verified to come from me that says, e.g., "Please pay $50 to Alice from my account."), and giving this check to my bank
- I'm at a party, and a stranger Genevieve confides to me that she has a crush on a guy there and convinces me to pass a romantic note to him without reading it.
- Unbeknownst to me, the note actually says "Please pay $1000 to Genevieve from my account" ... and the guy is my banker. When he receives the note, he assumes I want to pay Genevieve and transfers money from my account to her.
- The flaw: My banker assumed that if a request came from me, I must have authorized it.

Steps of a CSRF Attack

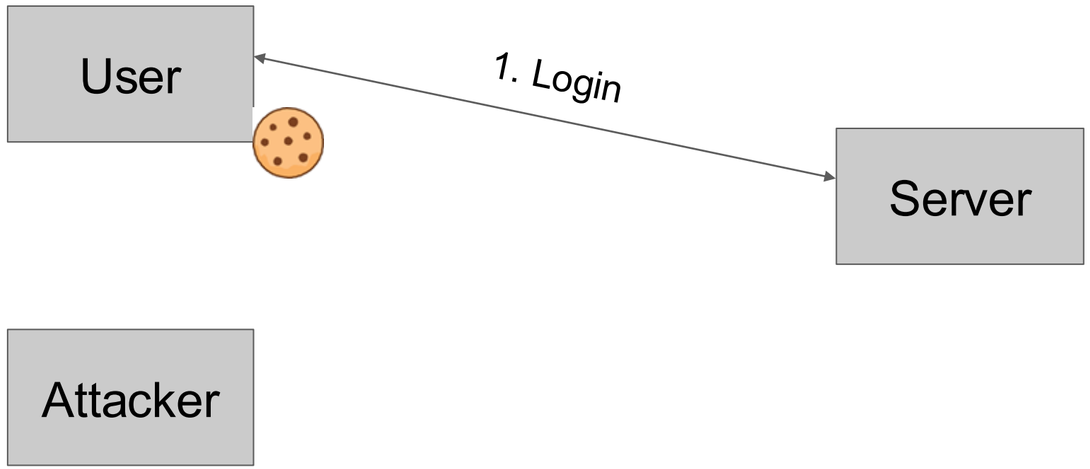


1. **User authenticates to the server**
   - User receives a cookie with a valid session token



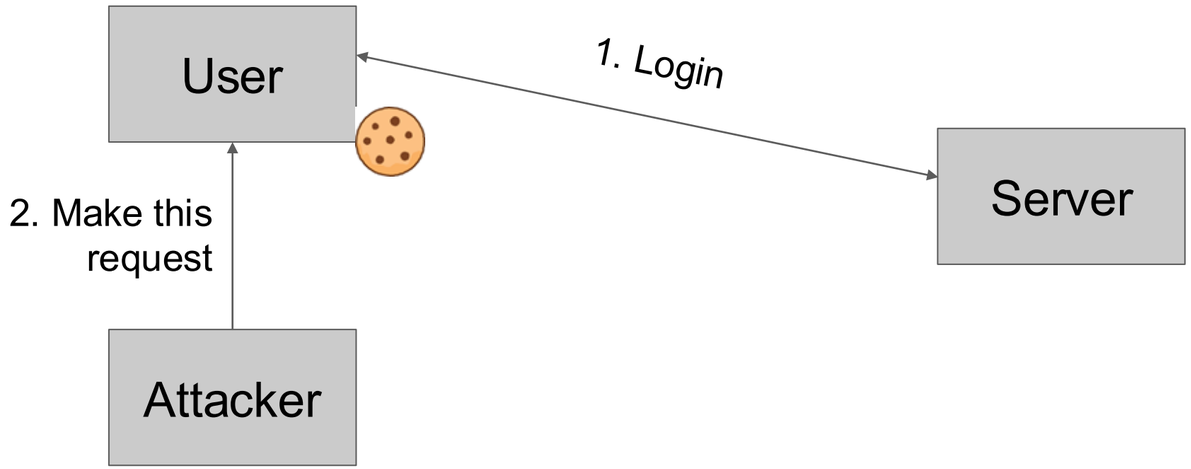
Steps of a CSRF Attack

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



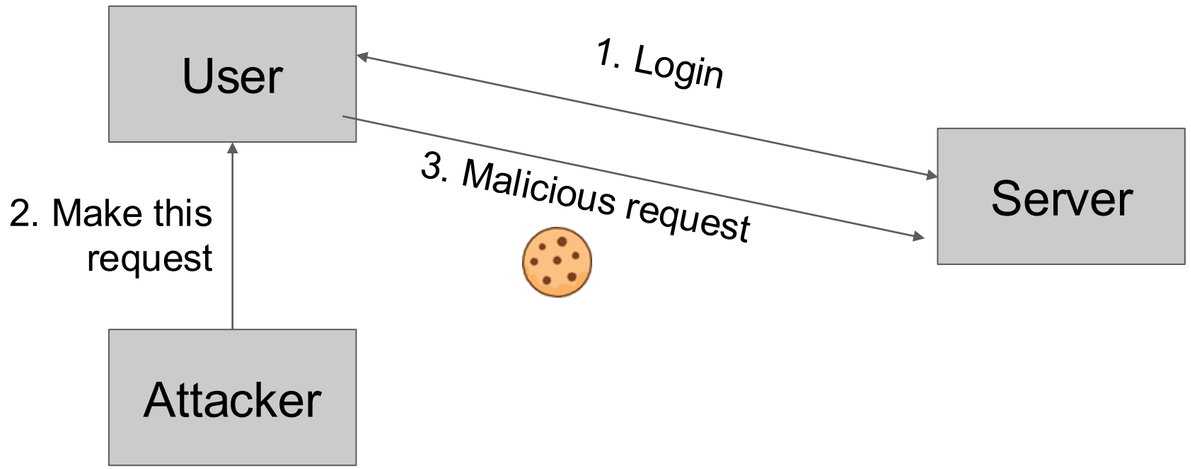
Steps of a CSRF Attack

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**
  - For example, send the victim an email with an enticing-looking 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

CSRF Example: Internet of Things (IoT)



- **IoT devices often use default passwords**
  - Example login URL: **http://10.1.1.1/login?user=admin&password=admin**
  - The attacker tricks the victim into making a request to this link, which gives the victim a session token cookie
- **IoT devices often process commands with web requests**
  - Example request URL: **http://10.1.1.1/set-dns-server?server=8.8.8.8**
  - These requests can perform actions that help the attacker mount other attacks
- **IoT devices don't implement CSRF defenses**
  - An attacker can publish an advertisement that makes these two requests
  - Any victim who opens a webpage with the advertisement will get their IoT device compromised!

CSRF Example: Facebook



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

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.

CSRF Defenses



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

CSRF Tokens



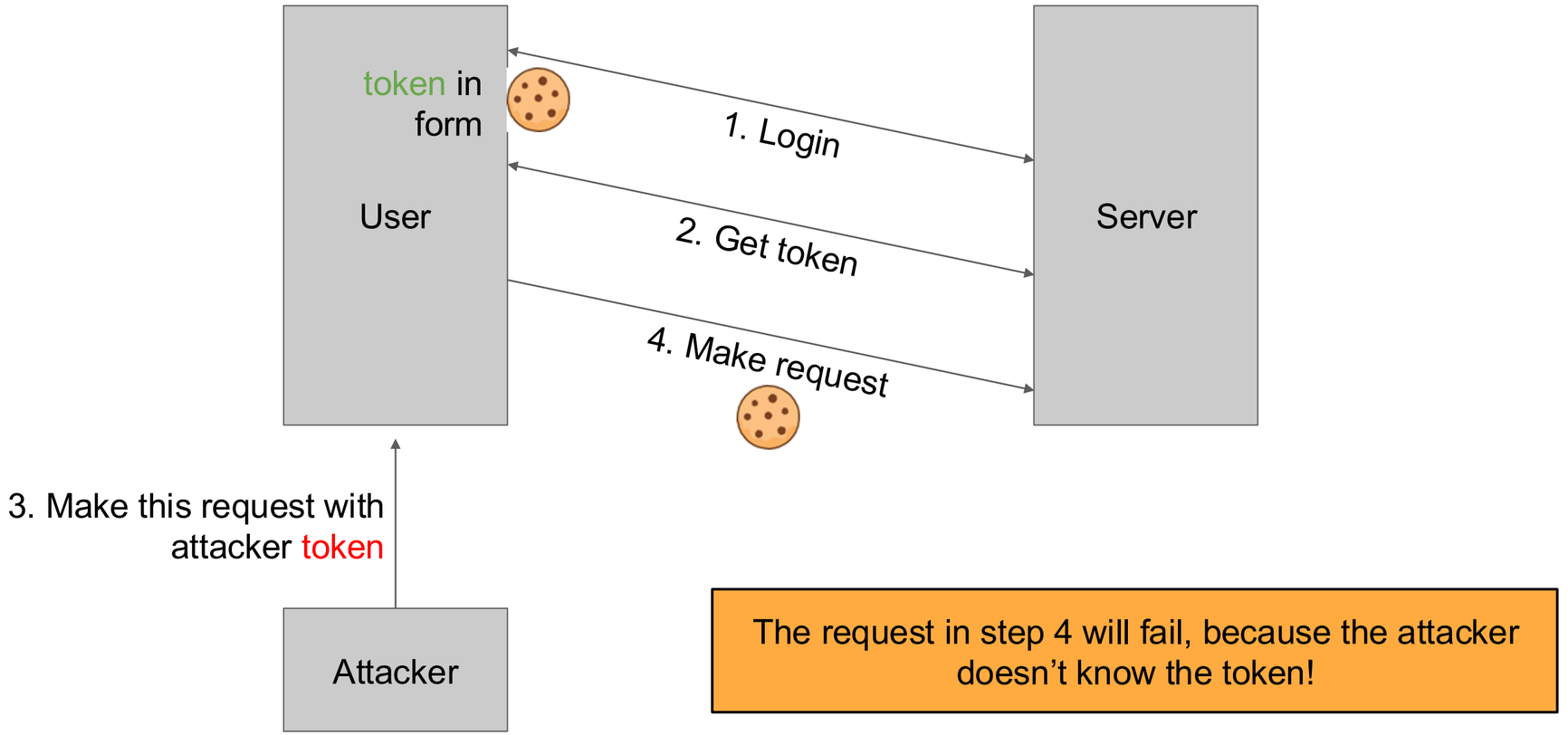
- **Idea: Add a secret value in the request that the attacker doesn't know**
  - The server only accepts requests if it has a valid secret
  - Now, the attacker can't create a malicious request without knowing the secret
- **CSRF token**: A secret value provided by the server to the user. The server only accepts the request is this value is attached.
  - 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: 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 request contains all form fields – including 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**

- **Using this header has limitations! – many ways to bypass it**

Referer Header: Issues



- **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=Strict flag**: An attribute of a cookie that specifies it should be sent only when the domain of the cookie **exactly** matches the domain of the origin of the webpage that triggered the request
  - 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 some less-common 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