# Cookies and CSRF

# Last Time: URLs



- URL: A string that uniquely identifies one piece of data on the web
- Parts of a URL:
  - Protocol: Defines which Internet protocol to use to retrieve the data (e.g. HTTP or HTTPS)
  - Location: Defines which web server to contact
  - Path: Defines which file on the web server to fetch
  - Query (optional): Sends arguments in name-value pairs to the web server
  - Fragment (optional): Not sent to the web server, but used by the browser for processing
- Special characters should be URL escaped

# Last Time: Parts of a Webpage



- HTML: A markup language to create structured documents
  - Create a link
  - Create a form
  - Embed an image
  - Embed another webpage (iframe or frame)
- CSS: A style sheet language for defining the appearance of webpages
  - As powerful as JavaScript if used maliciously!
- JavaScript: A programming language for running code in the web browser
  - JavaScript code runs in the web browser
  - Modify any part of the webpage (e.g. HTML or CSS)
  - Create pop-up messages
  - Make HTTP requests

# Last Time: Same-Origin Policy



- Rule enforced by the browser: Two websites with different origins cannot interact with each other
- Two webpages have the same origin *if and only if* the protocol, domain, and port of the URL all match exactly (string matching)
- Exceptions
  - JavaScript runs with the origin of the page that loads it
  - Websites can fetch and display images from other origins
  - Websites can agree to allow some limited sharing
- Q: Why do we need Same-Origin Policy?

# Last Time: HTTP



- HTTP: A protocol used to request and retrieve data from a web server
  - HTTPS: A secure version of HTTP
  - HTTP is a request-response protocol
- HTTP request
  - Method (GET or POST)
  - URL path and query parameters
  - Protocol
  - Data (only for POST requests)
- HTTP response
  - Protocol
  - Status code: A number indicating what happened with the request
  - Headers: Metadata about the response
  - Data

# Today: Cookies and CSRF



- Cookies
  - Parts of a cookie
- Cookie Policy
  - Setting cookies
  - Sending cookies
- Session Authentication
- Cross-Site Request Forgery (CSRF)
- CSRF Defenses

# Cookies

# 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
- Creating cookies
  - The server can create a cookie by including a `Set-Cookie` header in its response
  - JavaScript in the browser can create a cookie
  - Users can manually create cookies in their browser
- Storing cookies
  - Cookies are stored in the web browser (not the web server)
- Sending cookies
  - The browser ***automatically*** attaches relevant cookies in 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**
- The name and value can be any string
  - Some special characters can't be used (e.g. semicolons)

| | |
|---|---|
| Name | `Theme` |
| Value | `Dark` |
| Domain | `cci.charlotte.edu` |
| Path | `/sis-faculty` |
| Secure | `True` |
| HttpOnly | `False` |
| Expires | `12 Aug 2023 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 | `Theme` |
| Value | `Dark` |
| Domain | `cci.charlotte.edu` |
| Path | `/sis-faculty` |
| Secure | `True` |
| HttpOnly | `False` |
| Expires | `12 Aug 2023 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
- Each attribute is either True or False
- If the **Secure attribute** is True, then the browser only sends the cookie if the request is made over HTTPS (not HTTP)
- If the **HttpOnly attribute** is True, then JavaScript in the browser is not allowed to access the cookie

| | |
|---|---|
| Name | `Theme` |
| Value | `Dark` |
| Domain | `cci.charlotte.edu` |
| Path | `/sis-faculty` |
| Secure | `True` |
| HttpOnly | `False` |
| Expires | `12 Aug 2023 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

| | |
|---|---|
| Name | `Theme` |
| Value | `Dark` |
| Domain | `cci.charlotte.edu` |
| Path | `/sis-faculty` |
| Secure | `True` |
| HttpOnly | `False` |
| Expires | `12 Aug 2023 20:00:00` |
| (*other fields omitted*) | |

# Cookie Policy

# 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

# Cookie Policy

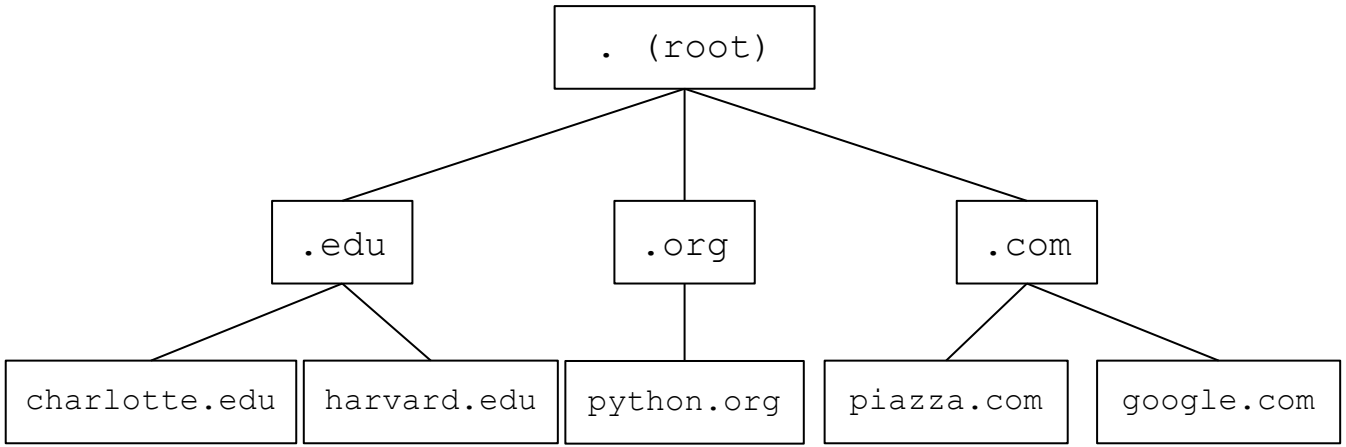


- **Cookie policy**: A set of rules enforced by the browser
  - When the browser receives a cookie from a server, should the cookie be accepted?
  - When the browser makes a request to a server, should the cookie be attached?
- Cookie policy is **not** the same as same-origin policy

# Domain Hierarchy

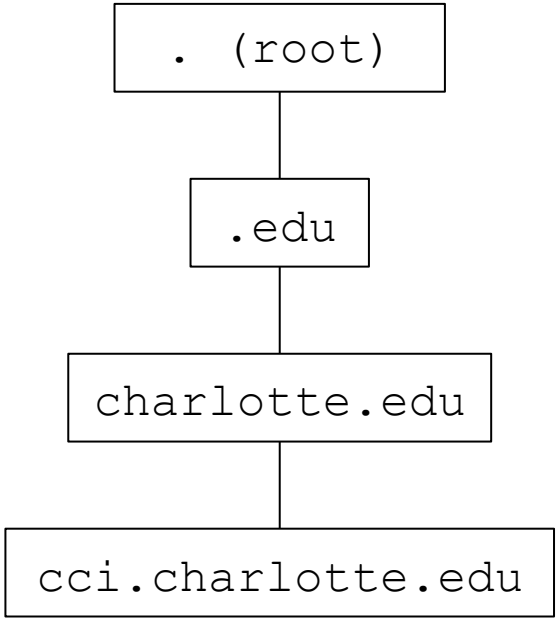


- Domains can be sorted into a hierarchy
  - The hierarchy is separated by dots



# Domain Hierarchy






`.edu` is a **top-level domain** (TLD), because it is directly below the root of the tree.

`cci.charlotte.edu` is a **subdomain** of `charlotte.edu`.

# 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 with domain attribute Y if
  - The domain attribute is a **domain suffix** of the server's domain
    - X ends in Y
    - X is below or equal to Y on the hierarchy
    - X is more specific or equal to Y
  - The domain attribute Y is not a top-level domain (TLD)
  - No restrictions for the Path attribute (the browser will accept any path)
- Examples:
  - mail.**google.com** can set cookies for 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

# Cookie Policy: Sending Cookies



(server URL)

https://cci.charlotte.edu/academics/software-and-information-systems

charlotte.edu/academics

(cookie domain) (cookie path)

Quick method to check cookie sending: Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

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

# Cookie Policy: Sending Cookies



(server URL)

https://cci.charlotte.edu/academics/software-and-information-systems

charlotte.edu/about-us/

(cookie domain) (cookie path)

Quick method to check cookie sending: Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

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

# Session Authentication

# 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?

# 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)

# Review: Cookies and Session Tokens



- Session token cookies are used to associate a request with a user
- The browser automatically attaches relevant cookies in every request

# 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

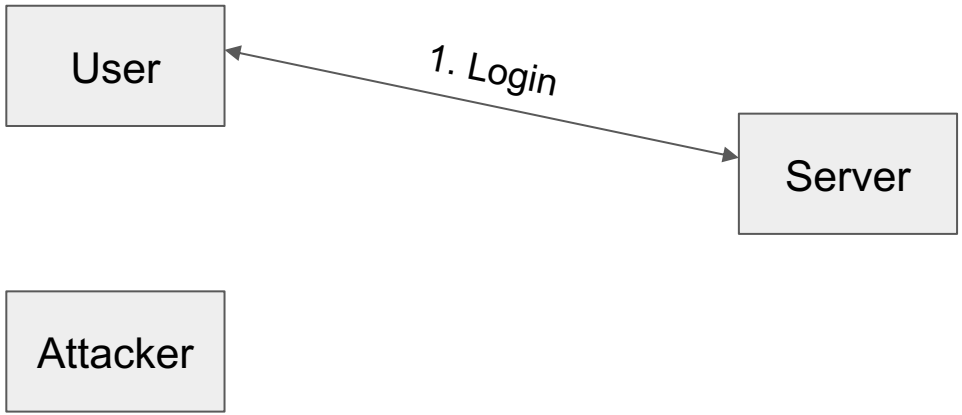
# Steps of a CSRF Attack



# Steps of a CSRF Attack



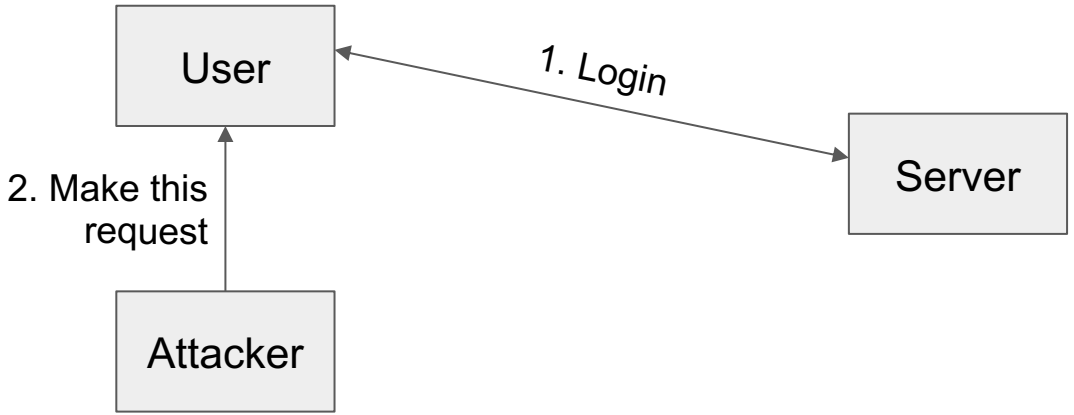
1. User authenticates to the server
    - User receives a cookie with a valid session token



# Steps of a CSRF Attack



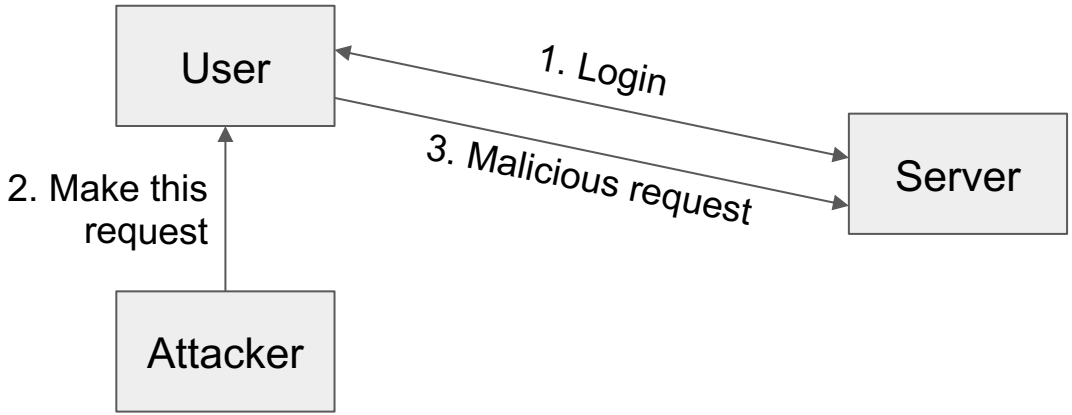
1. User authenticates to the server
    - User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server



# 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
  - 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

# Can Same-Origin Policy help?



- Cross-Site Request Forgery (CSRF)
  - 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

- Same-Origin Policy (SOP)
  - Brower-level security control
  - Prevents scripts from one origin from **reading** data from another origin

- Can SOP help with defending against CSRF?
  - With SOP, reading data from another origin is not permitted, but
  - form submissions are still permitted, which results in some server side state change

# 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 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

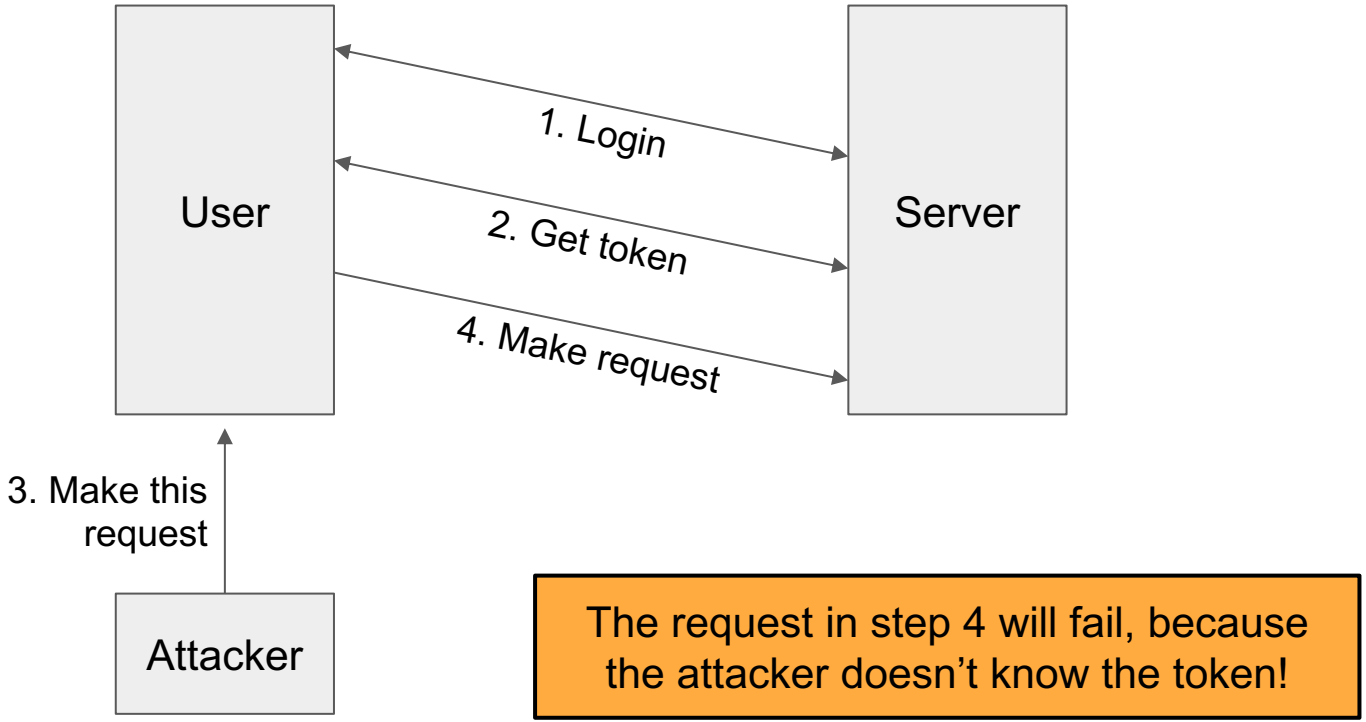
# CSRF Tokens: Usage



- 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

# Attacks on Cookies

# Cookie Ambiguity



- If two cookies should both be sent to a page with the same name, they are sent in an **undefined** order!
  - Consider two cookies:
    - `name=value1; Domain=bank.com; Path=/`
    - `name=value2; Domain=bank.com; Path=/page`
  - The browser would send both cookies to `bank.com/page` in an undefined order!
    - The server doesn't receive the Domain and Path attributes
    - The server might not be able to tell the cookies apart

# Spectre Attack: Vulnerability



- Original browser design: Chrome isolated **each tab** in its own Unix process
  - Security sandboxing: The operating system (OS) makes sure that one process cannot access other processes
  - Makes attacks harder: To compromise another tab, you have to exploit the browser code and escape the Unix sandbox
- Issues with this design
  - There are many scenarios where a program wants to protect data from other parts of the same program
  - Notable example: If one tab includes multiple origins (e.g. from an iframe embed), the browser must enforce same-origin policy: JavaScript from one origin cannot read cookies related to the other origin

# Spectre Attack: Exploiting browser design



- Spectre: An attack exploiting this browser design
  - The victim visits **`evil.com`** in a browser tab
  - **`evil.com`** opens an iframe with **`victim.com`**
  - Recall: JavaScript in **`evil.com`** should not be able to read any cookies from **`victim.com`**
  - **`evil.com`** and **`victim.com`** are now running in the same operating system process
  - No operating system sandboxing is active! The only memory protection is enforced by the JavaScript compiler
  - If we can break the JavaScript compiler, we can read memory from **`victim.com`**

# Spectre Attack: Exploiting the processor



- Quick review: Modern processors
  - Designed to be very fast: High instructions per cycle (IPC)
  - Uses aggressive behavior to achieve high IPC
    - Aggressive caching
    - Branch prediction: Guess the outcome of a branch and start executing that branch before the outcome is known
    - Speculative execution: Execute some code if the processor thinks it'll be executed later
  - Note: Predictions are not always correct
- Spectre: Exploits a hardware side-channel attack
  - Use a side channel (e.g. timing, cache state) to detect the results of failed speculative execution
  - Use a side channel to see what the input to the speculative execution was
  - Idea: Force speculative execution by forcing the processor to make wrong predictions
  - Idea: Read the side channel to see the results of the speculative execution

# Spectre Attack: Exploiting the processor



- Using the side-channel to break the JavaScript compiler's memory isolation
  - Recall: **`evil.com`** has loaded an iframe with **`victim.com`**
  - **`evil.com`** executes a repeated loop with legitimate instructions
  - The branch predictor is trained to think this loop will keep going (it will keep guessing the **`while`** condition is true)
  - On the last run of the loop, the processor incorrectly guesses it will keep going and starts running the loop one more time (speculative execution)
  - In the speculative run of the loop, do computation on illegal memory (e.g. **`victim.com`**'s cookies)
  - Use the side-channel to leak some information about the illegal memory being read
  - Repeat this process to leak all the desired information

```
i = 0
while i <= 1000:
  if i <= 1000:
    [legal things]
  else:
    [illegal things]
  i += 1
```

Speculative execution: The else case never runs, but the predictor will try to execute it after the last run of the loop

# Spectre Attack: Defenses



- Chrome and Firefox now run each *origin*, not tab, in its own process
  - Known as "Site Isolation"
  - Recall: The operating system (OS) makes sure that one process cannot access other processes
- Security: Spectre attack is defeated
  - When **`evil.com`** loads an iframe with **`victim.com`**, the two frames are run in different processes
  - Speculative execution no longer works: the OS prevents the **`evil.com`** process from accessing memory of the **`victim.com`** process
  - The attack now requires breaking the OS isolation (much harder)
- Cost: Processes are expensive
  - Lots of memory overhead
  - Switching between processes is expensive: optimizations (e.g. caches) must be wiped

# Spectre Attack: Takeaways



- **Takeaway**: Enforcing isolation between websites (same-origin policy, cookie policy) requires the browser to be securely designed. Getting this right can be very tricky!
- **Takeaway**: The web was not designed in security in mind. Many defenses (e.g. same-origin policy) were added afterwards, leading to awkward design and expensive performance costs