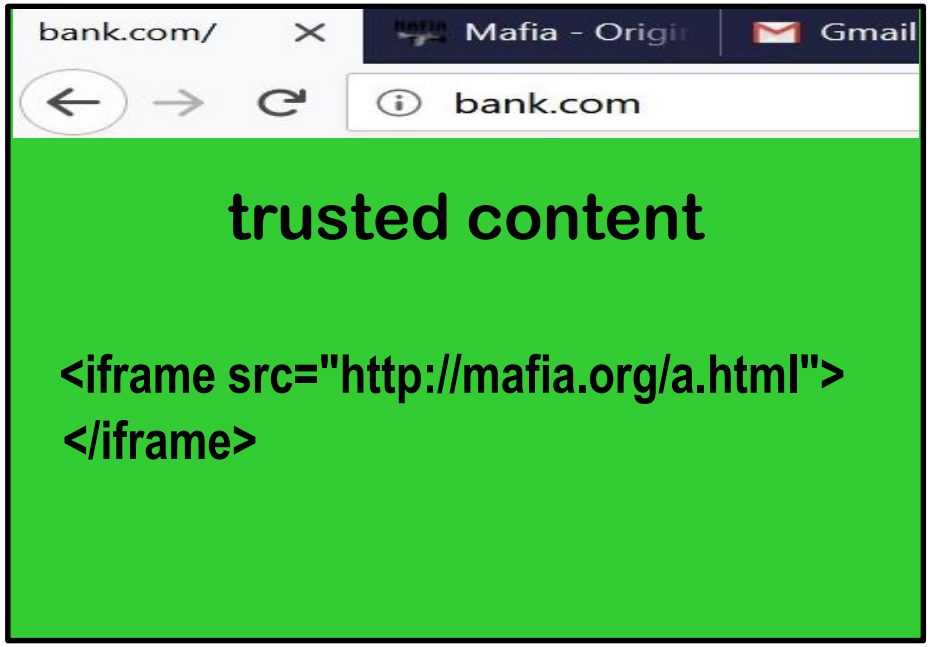
Client-side attacks continued

Last week: security provided by SOP



SOP protects against malicious content (eg advertisement) from another origin

Last week: security provided by SOP

SOP protects against malicious content (eg advertisement) from another origin

Last week: security provided by SOP

(JavaScript in) a.html cannot observe or interfere with surrounding webpage, thanks to SOP

SOP examples

For example of the SOP in action, experiment with

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

and look at the HTML code

Last week: by-passing SOP with HTML-injection

Contents included with HTML injection (incl. XSS)
(reflected, stored, or via DOM)
is counted as coming from the same origin

SOP & XSS

Can SOP prevent or mitigate XSS?

- eg a malicious Brightspace forum post with XSS

***NO*, as XSS scripts come from the same origin**

  - e.g. an attack script stored in Brightspace forum is 1st party content, and comes from the same origin as legitimate scripts from Brightspace

YES, if you design your website to use multiple origins

- if uploaded content is hosted on a different domain

  say `untrusted_student_content.ru.nl`

  instead of `brightspace.ru.nl`

  uploaded scripts cannot access `brightspace.ru.nl`
- Eg gmail uses `googleusercontent.com` for this purpose
- Brightspace could also use this trick, for Defense in Depth

CORS (Cross-Origin Resource Sharing)

- SOP is too strict in many settings

- Using CORS, a website can relax the SOP policy to allow some cross-origin requests

  For example

  `Access-Control-Allow-Origin: *`

  allows any cross-origin requests

  `Access-Control-Allow-Origin: https://trusted.com`

  allows cross-origin requests from a specific origin

We won't go into the gory details of CORS in this course

SOP problems

Modern browsers are very complex, and SOP is complex

Hence: some implementations screw things up

See CVEs about this

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Same%20Origin%20Policy

Bug: SOP bypass in Internet Explorer 6 & 7

The DOM provides the `.domain` property for the domain part of a document's origin.

A bug in Internet Explorer allowed any JavaScript to *set* this property

So a malicious script could include

```
<script>
  var document;
  document = {};
  document.domain = 'bank.com';
  // now we can access bank.com content
  ...

</script>
```

Bug: SOP bypass in Android WebView [CVE 2014-6041]

WebView is a web rendering engine for Android

- **i.e. it renders (aka displays) a piece of HTML**

A null character before JavaScript would by-pass the SOP

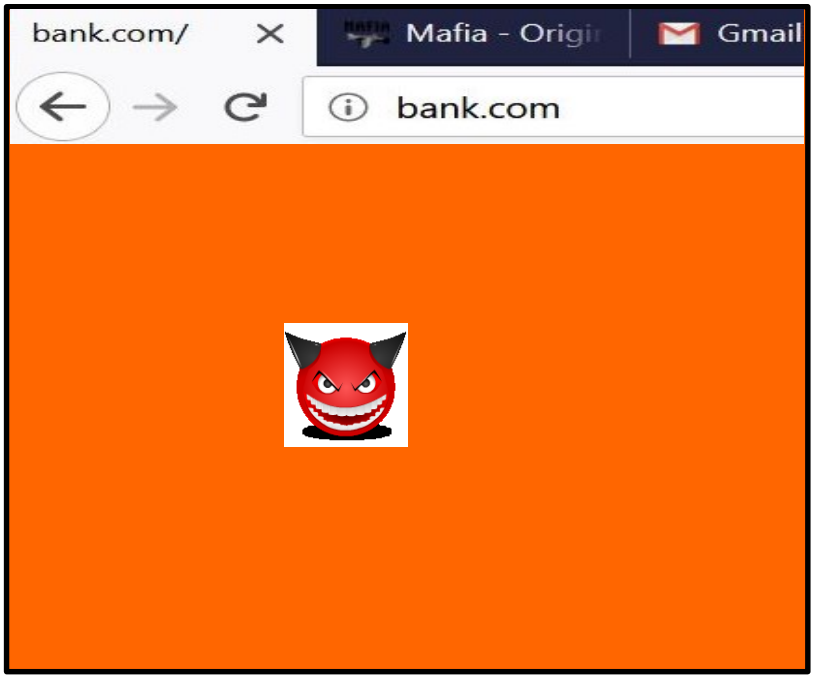
```
... onclick="window.open('\u0000 javascript:alert(..))
```

This bug affected 42 out of the top 100 apps in the Google Playstore with 'Browser' in their name

Alternative: supply chain attack

Confusingly, 3rd party JavaScript included in 1st party HTML source is counted as same origin, so SOP does not impose access restrictions on lib.js

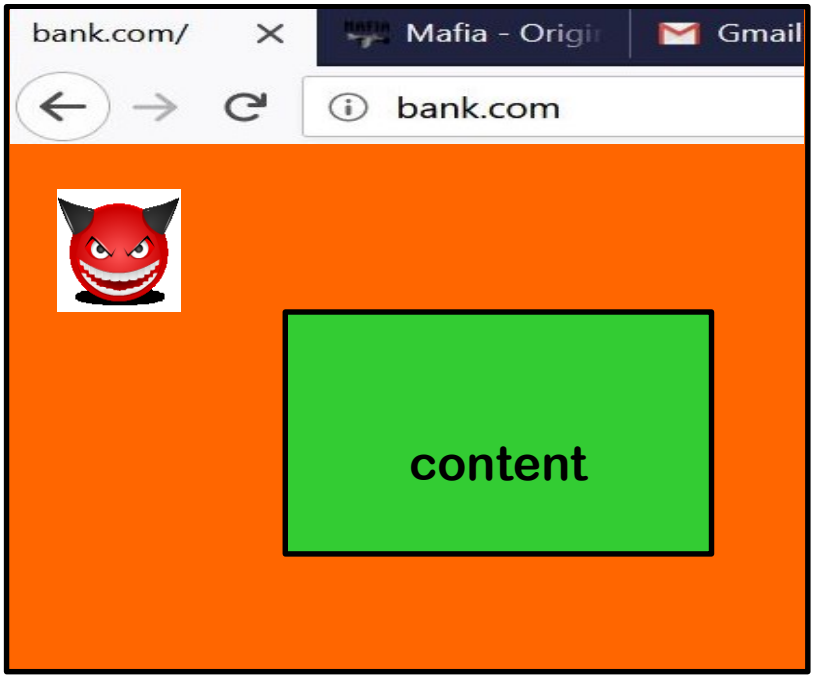
Alternative: use a malicious website



Malicious site could phish for logins & passwords.

It could also include malicious links to the attacked website,
eg abusing CSRF

Or: malicious website with genuine iframe



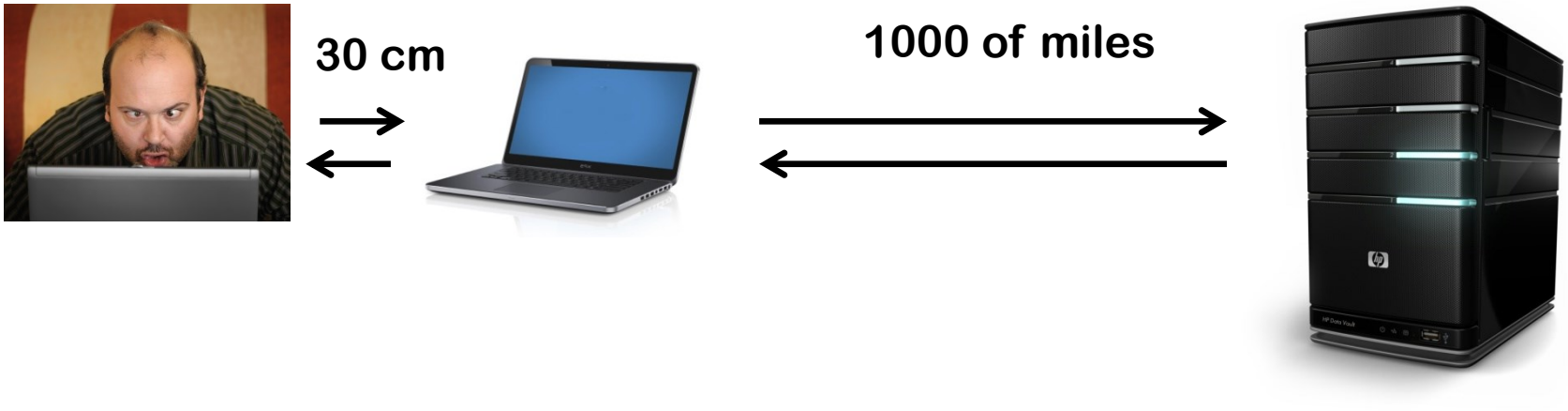
SOP protects against malicious site from observing or messing with trusted content

- but, as we will see, user can still be misled

More attacks on clients, esp. the user

URL obfuscation,
Click-jacking/UI redressing,
CSRF

Securing the last 30 centimeter...



We can secure connections between computers 1000s of miles apart, eg using TLS,

but the remaining 30 cm between user and laptop remain a problem

Would you trust these URLs?

- `https://www.paypal.com:get_request%2Eupdate&id=234782&`

  Recall that a URL has the form

  `https://username:password@host/....`

  So what is the domain we are accessing?

- `https://www.paypal.com`

  How do you know that the first `p` *is not a Cyrillic character?*

URL obfuscation

Attacker tries to confuse the user (in e.g. phishing attack) by

- including a username before the domain name

  `https://www.visa:com@%32%32%30%2E%36%38%2E%32%31%34%2E...`

  which translates to the IP address `220.68.214.213`
- using strange Unicode characters in a homograph attacks

  `https://paypal.com` with a Cyrillic `p`

Browser bugs may offer more opportunities to confuse the user.
In a famous Internet Explorer bug, a URL with a null character, e.g.
`http://paypal.com%00@mafia.com,` would not display properly...

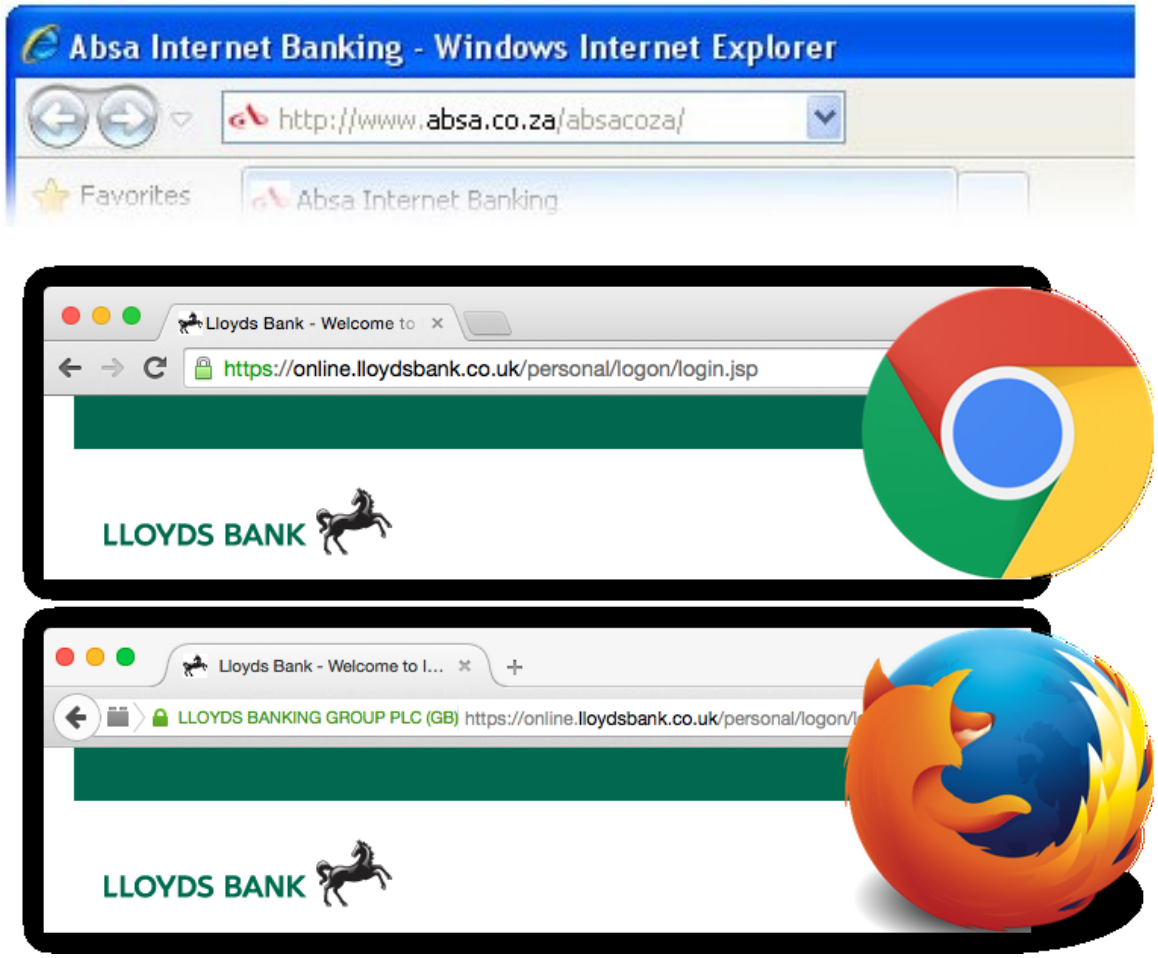
Countermeasures:

1. Punycode which encodes Unicode as ASCII to reveal funny characters

   `www.xn-pypal-4ve.com`
2. Domain highlighting to make it clear which part of URL is the domain name

Browser warnings – use of strange character sets

Highlighting domain name in address bar

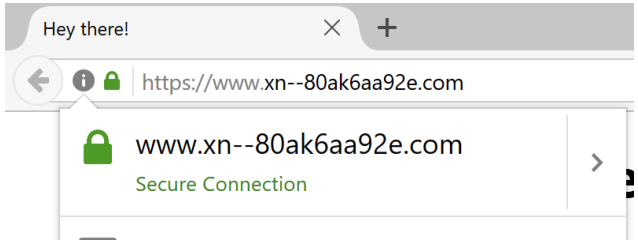


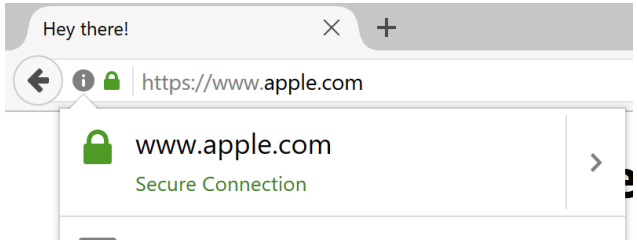
Newer homograph attack [2017]

Some browsers display

https://xn--80ak6aa92e.com

as apple.com



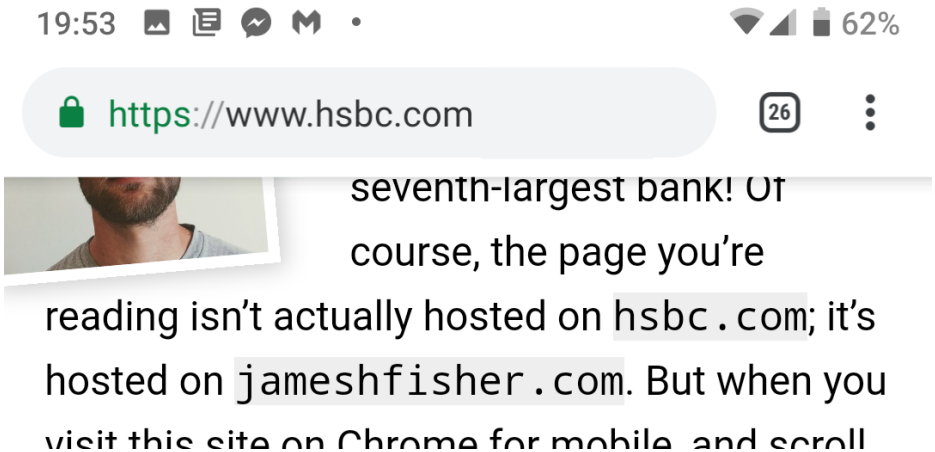


Problem: some browsers only use puny encoding if URL mixes several characters sets, not if *all* characters are from *one* (misleading) character set

See https://www.xudongz.com/blog/2017/idn-phishing/

Attack still works in Firefox, not In Chrome & Edge?

Latest UI confusion on mobile phones [2019]



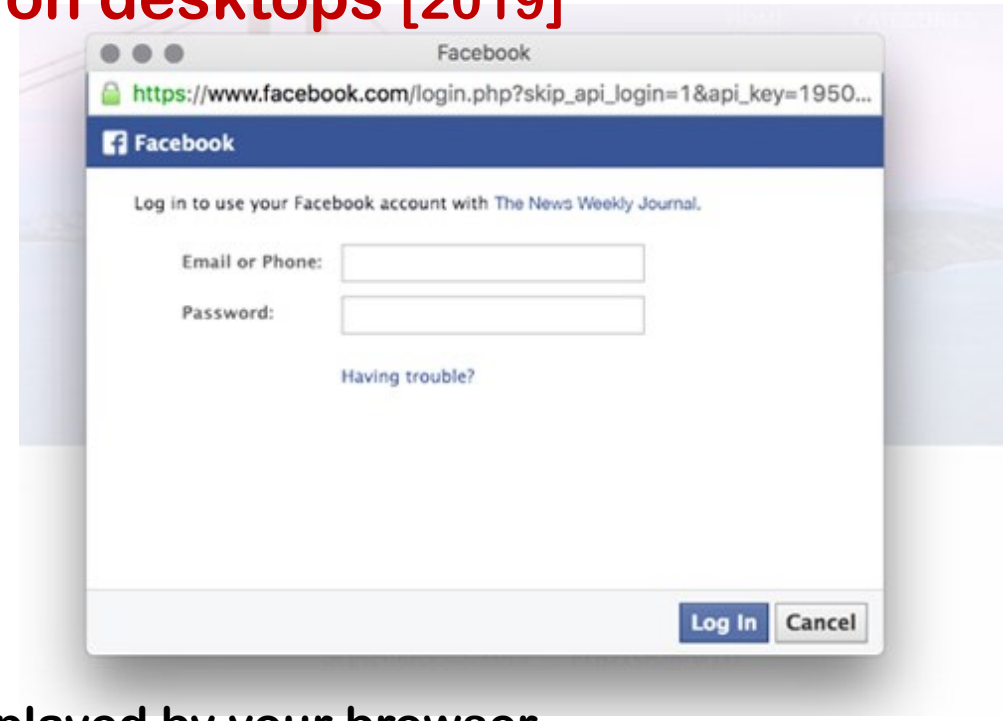
Chrome on mobile phone hides URL bar when you scroll down. Attacker can abuse this feature to display a fake URL bar.

See https://jameshfisher.com/2019/04/27/the-inception-bar-a-new-phishing-method/

UI confusion on desktops [2019]

Is this pop-up window legit?

It has an https-link
to facebook.com



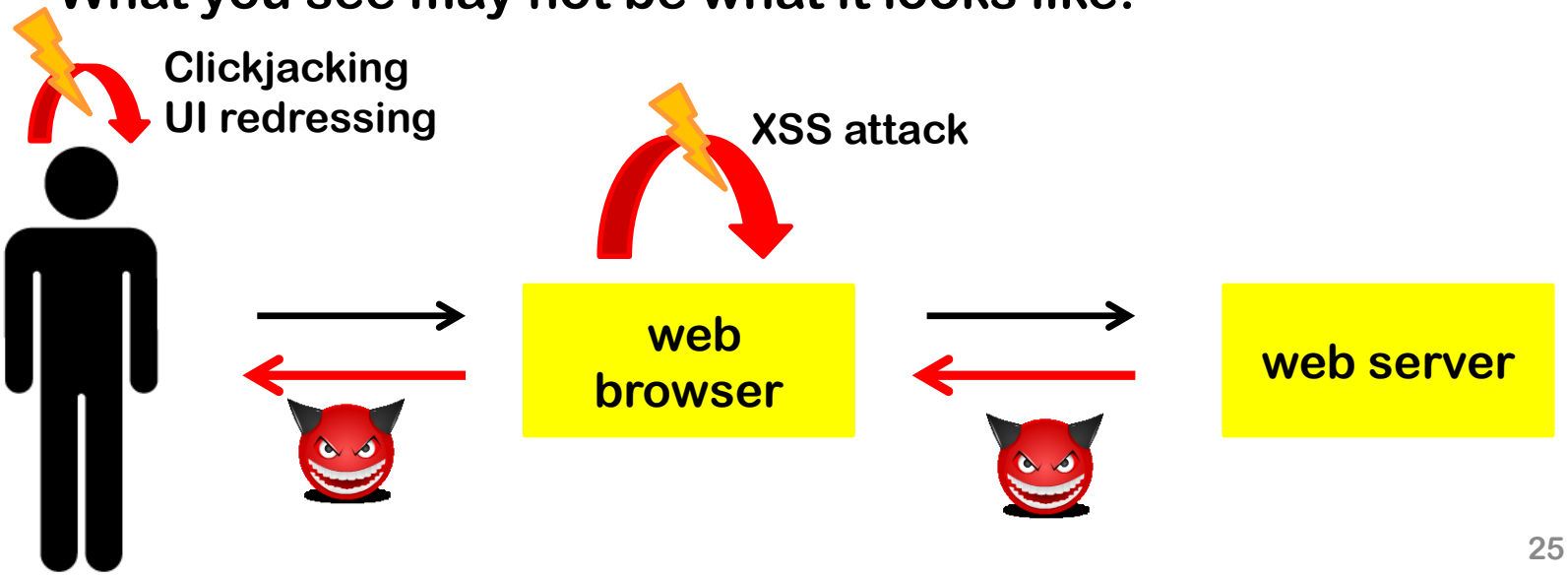
- This is not pop-up window displayed by your browser,
  but a fake pop-up rendered inside a malicious phishing webpage
- *How can you tell?*
  - You can move this 'pop-up window'
    but you cannot drag it outside of the confines of the webpage

See https://myki.com/blog/facebook-login-phishing-campaign
and check the video there https://youtu.be/nq1gnvYC144

Click-jacking & UI redressing

Click-jacking & UI redressing

- 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 *trust that users have in a webpage and their browser*
  - ie. the trust that users have in what they see
  - What you see may not be what it looks like!



Click-jacking & UI redressing

Terminology is very messy

- Click-jacking and UI redressing sometimes regarded as synonyms;
  Some people regard click-jacking as an ingredient for UI redressing

- To add to the confusion, these attacks often come 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.police.nl">Trust me, it is safe to
 click here, you will simply go to police.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 (thanks to cookie)
- Click fraud

Business model for click jacking: 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

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)

- Click bots (hijacked computers in botnet, running software to automate clicking)

Example: website with age confirmation check

S3.AMAZONAWS.COM

Votre vie avant et après le mariage, en images

Pour accéder à ce site, vous devez être âgé de 16 ans ou plus.

Avez-vous plus de 16 ans?

OUI.

Example: website with age confirmation check

Inspecting HTML source to see what you are actually clicking

```
<div class="popup-copy">

    <h4>Pour accéder à ce site, vous devez être âgé de 16 ans ou
    plus.</h4>
    <h4>Avez-vous plus de 16 ans?</h4>

        <button type="submit" name="submit" class="btn
        btn-newsletter" onclick="top.location.href = '
            https://s3.amazonaws.com/q93tz5838rkh7kgmn6borad/
            s730aI5Vxa9Uejre.html'">Oui.</button>
        <iframe class="d8485i63ikjasdiu73h" id="
        d8485i63ikjasdiu73h"  onload="" scrolling="no" src="
        https://pejzbugpedau.s3.amazonaws.com/iframe.html"></
        iframe>
    </form>
</div>
```

Inspecting content of these Amazon S3 buckets leads to

https://mobile.facebook.com/v2.6/dialog/share?app_id=283197842324324&href=https://example.com&in_iframe=1&locale=en_US&mobile_iframe=1

Example: website with age confirmation check

Clicking age confirmation bucket shares a post of Facebook

Such clickjacking can get you many likes or shares!

Attack only worked in the Facebook mobile app, not in a normal browser

- **NB the Facebook app is/contains a web-browser**

Read the description at

https://malfind.com/index.php/2018/12/21/how-i-accidentaly-found-clickjacking-in-facebook/

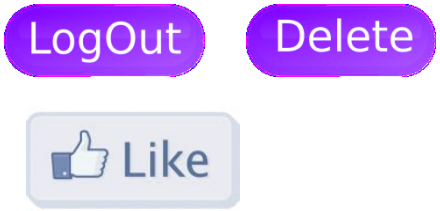
UI (User Interface) redressing

Attacker creates a malicious web page that includes elements of a target website, esp. links victims can click.

- **With iframe (inline frame) with content from attacked website**
  - **iframes allow flexible nesting, cropping, and overlapping**

Two approaches

1. **"steal" a button with non-specific text**



2. **make a iframe transparent**

NB esp. 1 looks a lot like CSRF, as we'll discuss later

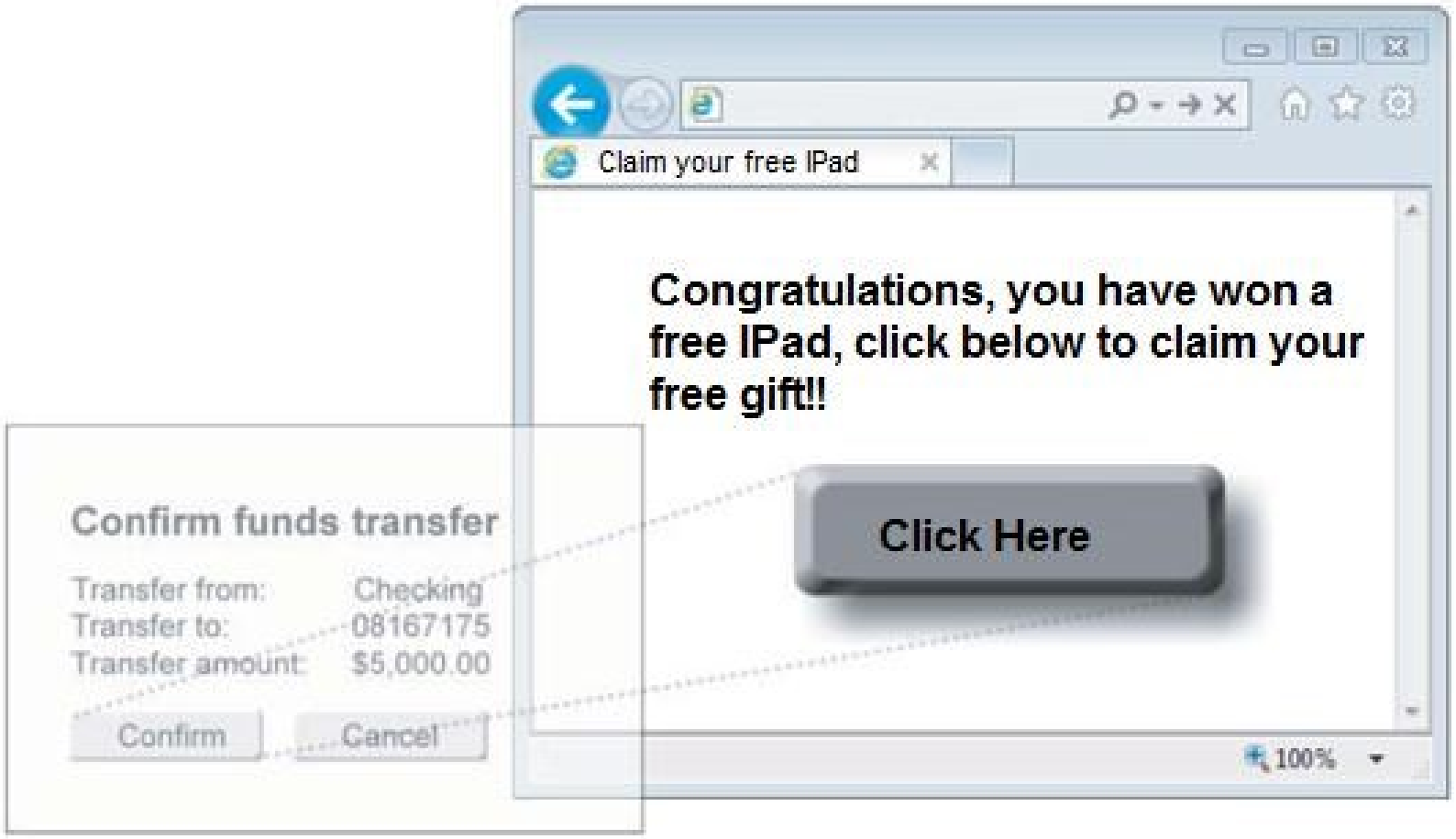
UI redressing example

Tricking users into altering security settings of Flash

- Load Adobe Flash player settings into an invisible iframe
- Click will give permission for any Flash animation to use the computer's microphone and camera

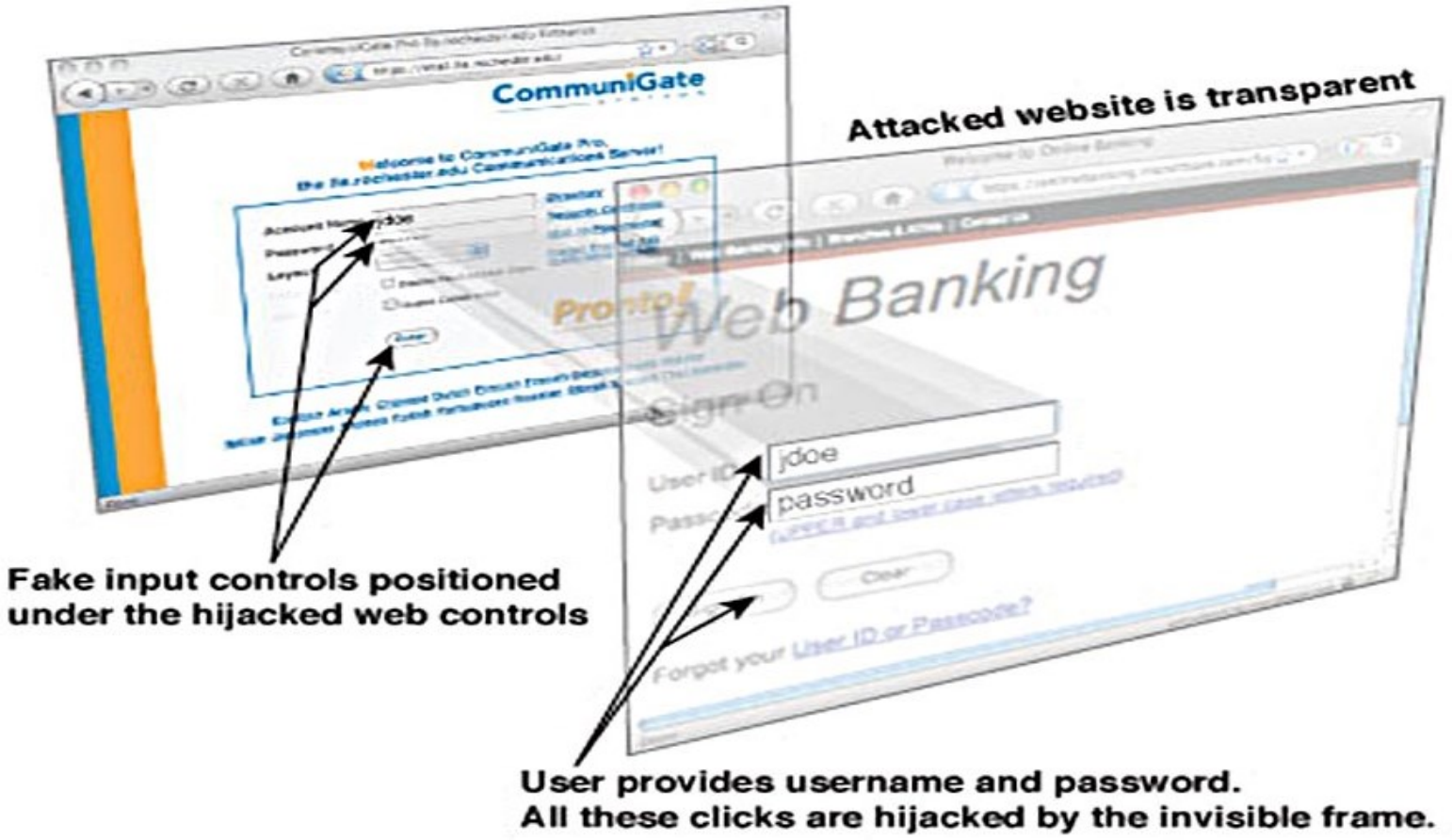
UI redressing example

Trick users into confirming a financial transaction



UI redressing example

Trick users to login to a banking website



Click-jacking and UI redressing: abusing trust

- These attacks abuse trust users have in a webpage
  - in what they *see* in their browser

- These attacks also abuse trust the web server has in browsers
  - Web server trusts that all actions from the browser performed *willingly* & *intentionally by the user*

- Some browser will prevent users from interacting with transparent content

  Check if your browsers does at
  http://www.cs.ru.nl/~erikpoll/websec/demo/clickjack_some_button.html
  http://www.cs.ru.nl/~erikpoll/websec/demo/clickjack_some_button_transparent.html

Variations of click-jacking

- like-jacking and share-jacking  

- cursor-jacking
  (See http://www.cs.ru.nl/~erikpoll/websec/demo/cursor-jacking.html)
- file-jacking (unintentional uploads in Google Chrome)
- event-jacking
- 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

- Countermeasure to prevent being included as iframe: webpage tries to bust any frames it is included in
- Example JavaScript code for frame busting

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

  - `top` is the top or outer window in the DOM; `self` is the current window
  - If an iframe executes this code, it will make itself the top window.
  - For a demo, see http://www.cs.ru.nl/~erikpoll/websec/demo/framebusting1.html which includes a frame-busting iframe http://www.cs.ru.nl/~erikpoll/websec/demo/framebuster.html

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

Busting the frame busting

HTML5 sandbox feature for iframes (discussed last week) can restrict capabilities of a victim iframe

- eg. it can be disallowed to change `top.location`

This can block the framebusting

- Example HTML code for sandboxing:

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

  - `allow-scripts`: allow scripts
  - `allow-forms`: allow forms
  - there is no `allow-top-navigation`, so the iframe is not allowed to change of top.location

For a demo, see
http://www.cs.ru.nl/~erikpoll/websec/demo/framebusting2.html

Better solution: X-Frame options

`X-Frame-Options` in HTTP response header introduced to indicate if webpage can be loaded as iframe

- Possible values

  `DENY` never allowed

  `SAMEORIGIN` only allowed if other page has same origin

  `ALLOW-FROM` *`<url>`* only allowed for specific URL (Only ?)

- Simpler than using JavaScript to do frame busting, and cannot be disable with sandbox-feature

- CSP (Content Server Policy) also provides ways to do this, but given the complexity of CSP, many sites continue to use `X-Frame-Options`

Example: website with age confirmation check

Why doesn't Facebook use `X-Frame-Options` to prevent malicious inclusion of share or like buttons?

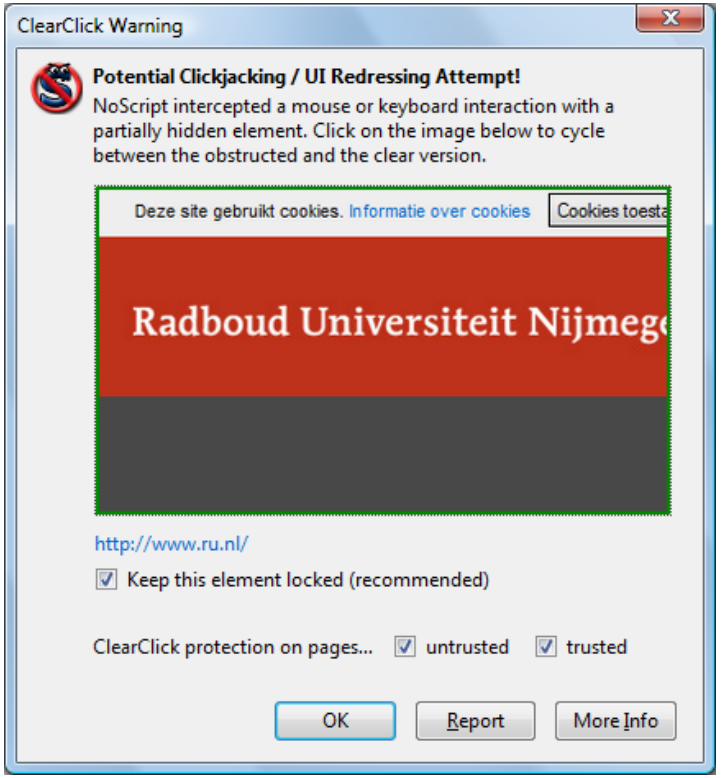
Facebook does set `X-Frame-Options` to `DENY`, but only for content served to a normal web browser, not for content sent to their mobile facebook app

See also

https://malfind.com/index.php/2018/12/21/how-i-accidentaly-found-clickjacking-in-facebook/

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 in an iframe
  - Takes screenshot of the iframe, its own 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 (showing screenshots so user can evaluate by himself)



CSRF
(Cross Site Request Forgeries)
revisited

Recall from 3 weeks ago: Abusing cookies without stealing them (CSRF)

Attacker sets up a malicious website mafia.com with a link on it to bank.com

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

- What happens if victim visits mafia.com and click this link?
- If the victim is logged in to the back, this request will be sent with the victim's cookies for bank.com
- This is called a Cross-Site Request Forgery (CSRF)

CSRF

- Ingredients
  - malicious link or JavaScript on attacker's website
  - abuses automatic authentication by cookie at targeted website
- Attacker 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
  - it does not involve UI redressing
  - if JavaScript is used, it is more than just clicking a link

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!

Countermeasures against CSRF
-
which might also help against clickjacking?

Recall: Countermeasures against CSRF [week 2]

1. Let client re-authenticate before important actions
2. Anti-CSRF token [aka Tokenization, discussed in Surviving the Web §4.4.3]
   - an unpredictable CSRF token in all webpages sent as hidden parameter in requests & checked server-side for freshness
     - effectively a second session ID
   - Only links from a legitimate webpage will have the right value for this token
   - Cross-site requests may have the right cookie but not the right token value
3. `SameSite` flag of the cookie
   - `strict` cookie never attached to cross-site requests
   - `lax` cookie only attached to top-level GET requests
     i.e. GET requests which change the address bar to `bank.com`
     (so not for loading an iframe on `mafia.com`)

Which of these help against click-jacking?

More CSRF prevention

At the server side:

- 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 in page from a.com, then referer/origin is a.com
  - The referrer is `Null` if there is none, eg. if you type in the URL in the address bar in a new browser window or tab
  - 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; then origin-header is sent

  But may be spoofed by attacker or suppressed by victim's browser (for privacy); because of latter reason, not a nice option to use

Which of these help against click-jacking?

More CSRF prevention: ARLs

- **Proposal for Allowed Referrer Lists (ARLs)**

  ARL is allow-list that specifies which origins are entitled to send authenticated requests to a given website

  - Allow-list compiled by web developers & sent by web site to browser
  - Enforcement done by the browser

- More precise than the `SameSite` cookie flag
- But: Compiling ARL requires effort or be infeasible
  - eg Paypal: ARL for e-commerce websites may include Paypal, but ARL for Paypal might have to include any legitimate e-commerce site using Paypal
- And: standardising ARL support for all browsers requires more effort

See Section 4.4.2 of "Surviving the Web" article

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**

Beware of confusion!

XSS
vs
CSRF
vs
Click-jacking & UI redressing

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 is more powerful than CSRF**
  - **`SameSite` can stop cookie being attached for CSRF, but not for UI redressing**
  - **With UI redressing any additional (hidden) parameters, incl. anti CSRF tokens, will be correctly added to the request; for CSRF not.**

CSRF meets HTML injection & XSS

Instead of using own site or emails with malicious links for CSRF, attacker could insert also malicious link as content stored on a vulnerable target 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
- If the CSRF attack uses JavaScript (eg for a POST), then it is *also* a XSS attack

CSRF vs XSS

Easy to confuse! Some differences:

- CSRF does not require JavaScript (for GET actions), XSS always does
- For any JavaScript used:
  - XSS: script is in webpage of the attacked website
  - CSRF: script can be anywhere, also the attacker's website
    - You can use XSS to do CSRF, as explained in previous slide, and then the CSRF code will be in the attacked site
- Server-side validation
  - Victim server cannot prevent CSRF, as the content reaching the target web site is not malicious or strange in any way
  - Victim server can prevent (reflected & stored) XSS, by trying to filter out malicious JavaScript (as discussed last week)

Trust: CSRF vs XSS

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

- XSS abuses trust of user & browser in the webserver
  - The user & browser trusts that all content of a webpage is really coming from that webserver
    - even though it may include HTML and scripts that are really coming from an attacker

- Clickjacking/UI redressing abuses both types of trust

Example: CSRF in Instagram

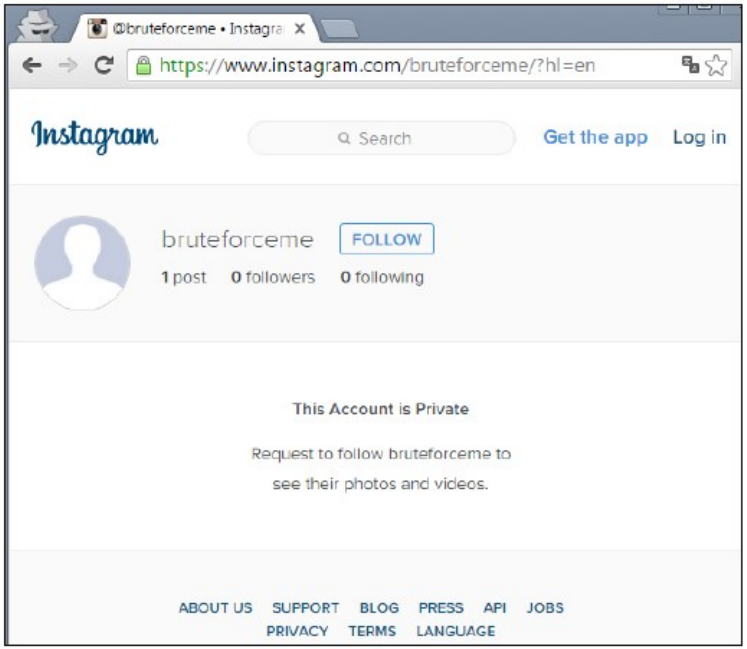
- **Thanks to Arne Swinnen** (https://www.arneswinnen.net)

- These and other attacks are presented by Arne at
  - https://www.youtube.com/watch?v=dsekKYNLBbc
  - https://www.arneswinnen.net/2016/02/the-tales-of-a-bug-bounty-hunter-10-interesting-vulnerabilities-in-instagram

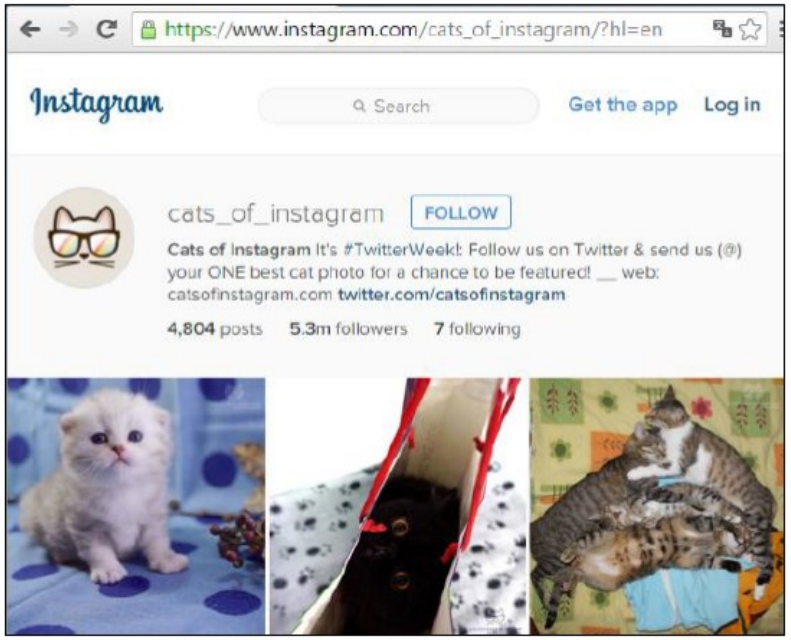
Example: CSRF in Instagram

Upload your pictures either in private or public account

Private account

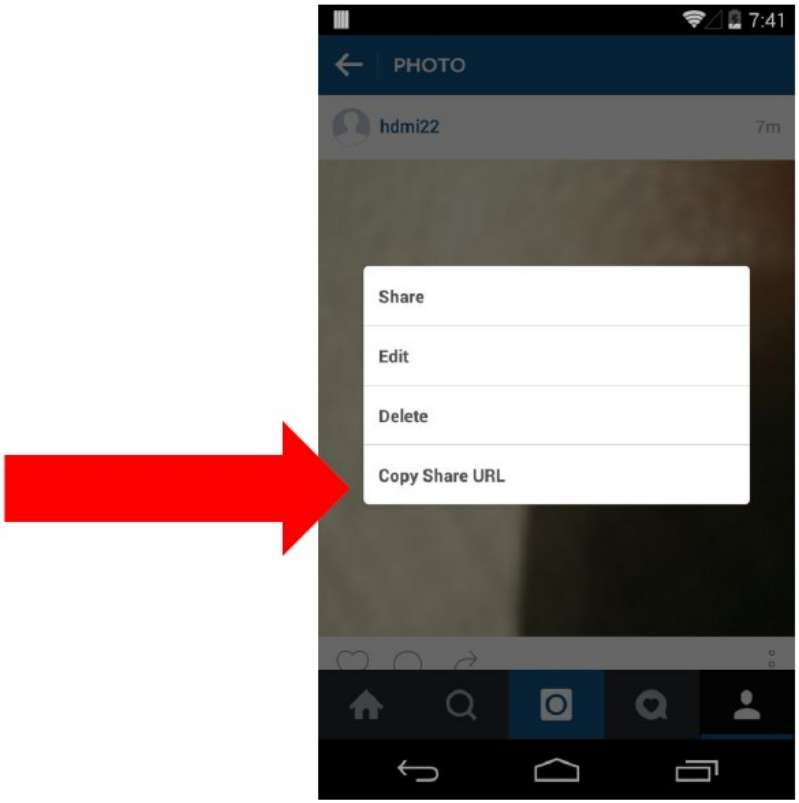


Public account



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 and modifies access rights so that picture is publicly accessable**

```
HTTP/1.1 200 OK
(…SNIP…)

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

- **Vulnerability: the GET-request that can be repeated and modified (using different picture-id)**
- **So if attacker can make the victim issue these GET-requests that can expose private pictures**

Example: CSRF in Instagram

How to obtain picture-IDs needed for this CSRF?

- **Messing with the usertag allows an authorisation bypass: attacker can retrieve picture-IDs of victim**

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
djlhCDPCKyr3mKSz5ey%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 bug bounty