Software and Web Security 2

Forced/forceful browsing

Forced browsing (not in book!)

- Supplying a URL directly (forcing the URL) rather than by accessing it by following links from other pages

- Modify (numerical) value in known URL
  - September 2011: miljoenennota leaked before Prinsjesdag (miljoenennota.prinsjesdag2010.nl, change 2010 in 2011)
  - December 2012: Christmas speech queen Beatrix leaked by manipulating URL of speech in 2011

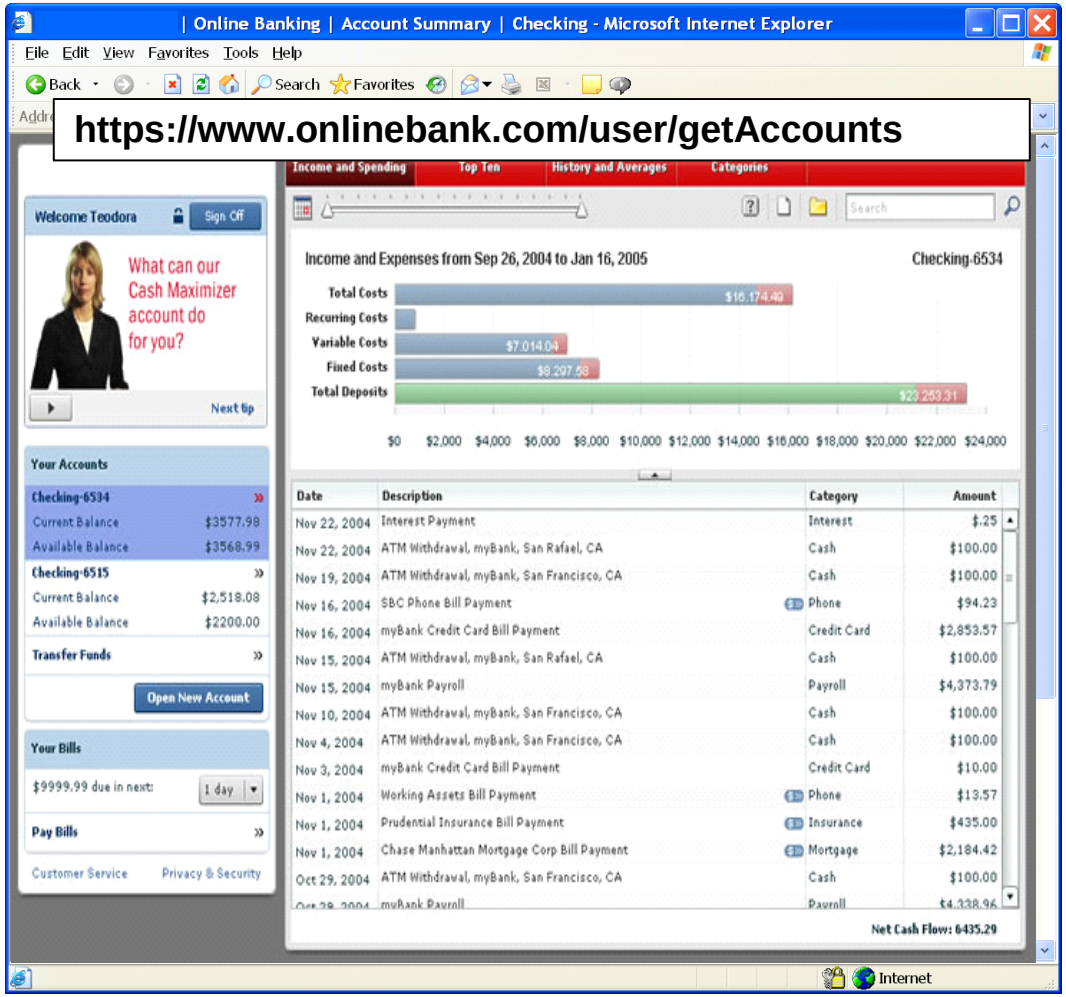
- Modify query parameters in known URL
  - Use brute force search

Forced browsing (not in book!)

- Client ‘attack’ on server
  (with intention to access restricted/hidden resources)

- OWASP top 10:
  - Failure to restrict URL access (2010)
  - Missing function level access control (2013)

Failure to restrict URL access



- Attacker notices the URL indicates his role

  /user/getAccounts

- Attacker modifies role

  /admin/getAccounts, or

  /manager/getAccounts

Defenses against forced browsing

- Avoid ‘sensitive’ information in URL (GET vs POST)
- Access control at server-side
  - Restrict access to authenticated users (if not public) by user-based or role-based permissions
  - Configure server to disallow requests for unauthorized file types (eg., config files, log files, source files, etc.)

Movie on brute-force forceful browsing at http://www.secure-abap.de/wiki/Movies

Software and Web Security 2

More attacks on Clients:

Clickjacking/UI redressing, CSRF

(Section 7.2.3 on Clickjacking;
Section 7.2.7 on CSRF)

Clickjacking & UI redressing

Click jacking & UI redressing

- Click jacking and UI redressing
  - try to confuse the user into unintentionally doing something that the attacker wants (typically clicking some link but sometimes also supplying text input in fields or just moving mouse)
  - 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)
- Some people treat click jacking and UI redressing 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 are 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>
```

Demo: see http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_basic.html

Why?

- click fraud

  Here instead of mafia.org, the link being click jacked would be a link for an advertisement.
- some unwanted side-effect of clicking the link, esp. if the user is automatically authenticated by the target website (eg. with a cookie)

  Here instead of mafia.org, the link being click jacked would be a link to a genuine website the attacker wants to target.

Click fraud

- In online advertising, web sites that publish ads are paid for the number of click-throughs, ie. number of their visitors that click on these ads
- Click fraud: attacker tries to generate lots of clicks on ads that are not from genuinely interesting visitors
- Motivations for attacker
  1. generating revenue for the web site hosting the ad, or
  2. generating 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?)

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 (software to automate clicking)
- Botnets (hijacked computers utilized by click bots)

UI (user interface) redressing (not in book!)

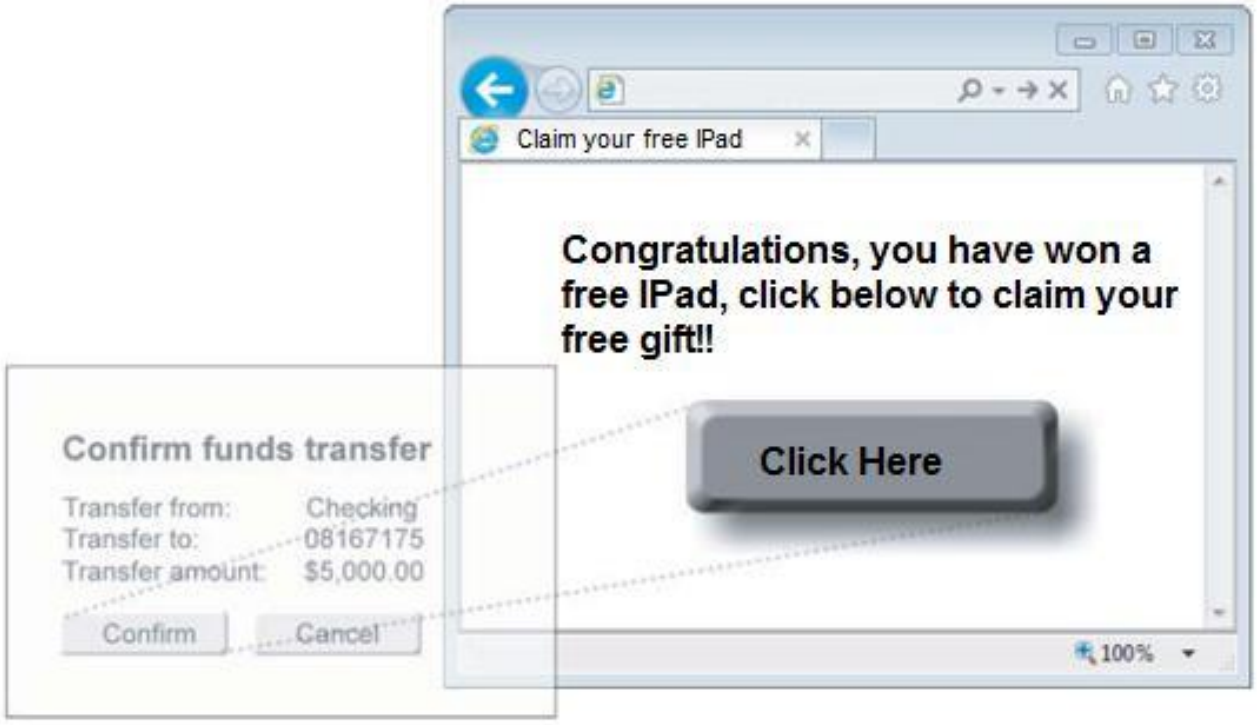
Attacker creates a malicious web page that includes elements of a target website

- typically using iframes (inline frames)
  A frame is a part of a web page, a sub-window in the browser window.
  An internal frame - iframe - allows more flexible nesting and overlapping
- possibly including transparent layers, to make elements invisible
  - this is not needed when the attackers "steals" buttons with non-specific text from the target website, such as

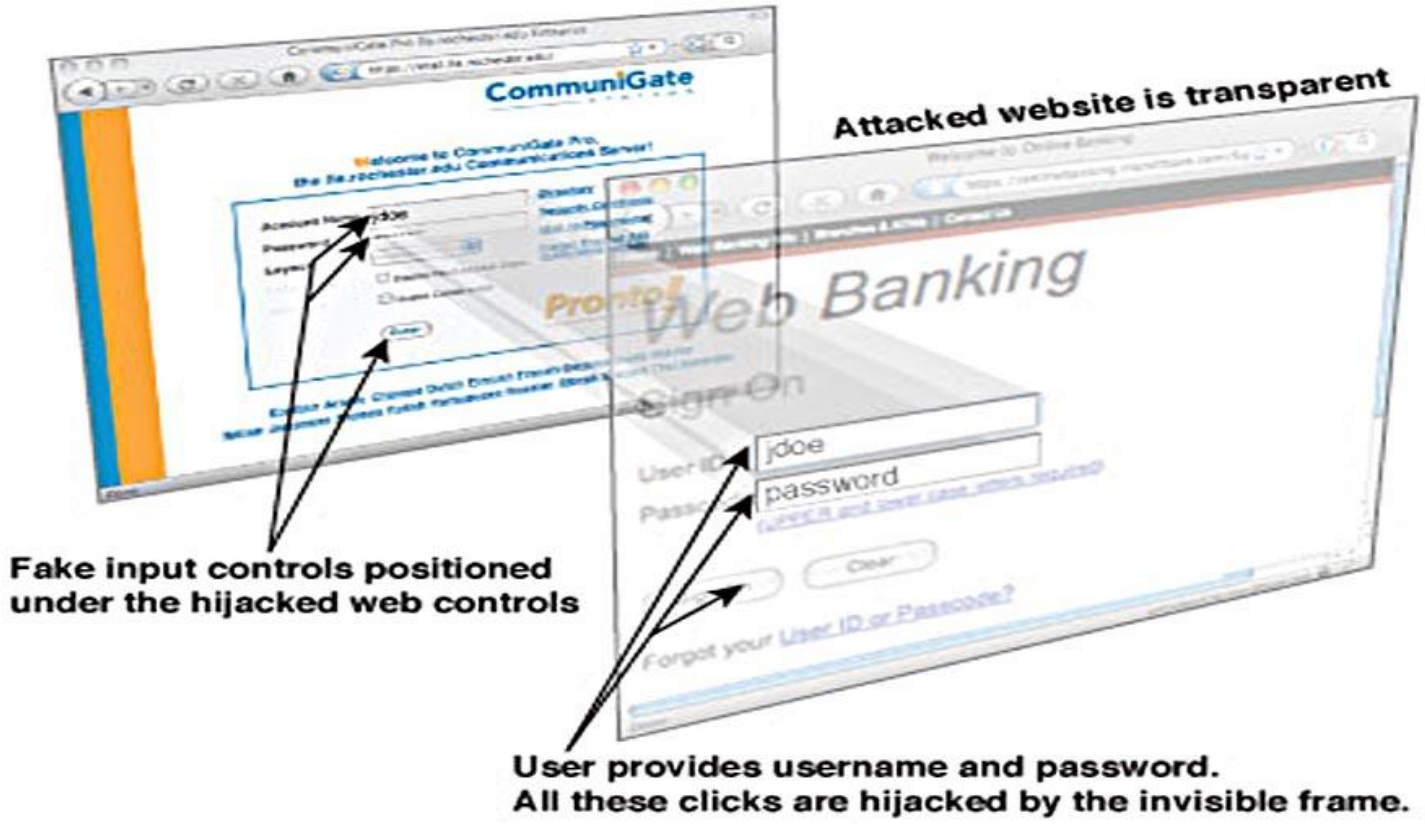
Demos

- http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_some_button.html
- http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_some_button_transparent.html
- http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_radboudnet_using_UI_redressing.html

UI redressing



UI redressing



Clickjacking and UI redressing

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

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

Variations of clickjacking

- Likejacking and sharejacking
- cookiejacking – in old versions of Internet Explorer
- filejacking – unintentional uploads in Google Chrome
- eventjacking
- cursorjacking
- classjacking
- double clickjacking
- content extraction
- pop-up blocker bypassing
- strokejacking
- event recycling
- svg masking
- tapjacking on Android phones
- ...

Countermeasures against Clickjacking & 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 the top or outer window, **`self`** is the current window.

Lots of variations are possible. Some frame busting code is more robust than others.

For an example, you can try the Blackboard webpage, which uses JavaScript to bust frames, eg http://www.cs.ru.nl/~erikpoll/sws2/demo/clickjack_bb_using_UI_redressing.html

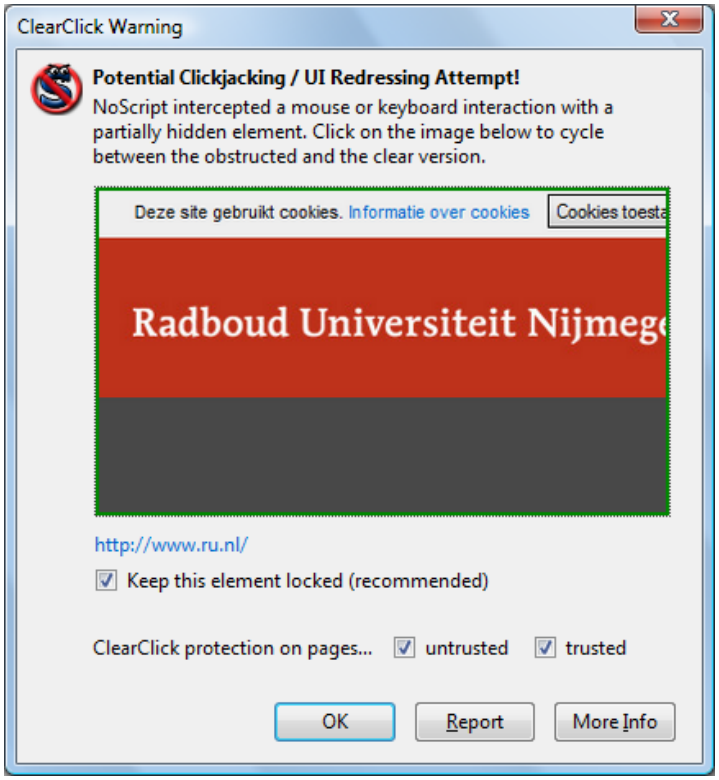
X-Frame options

- Introduced by Microsoft in 2008
- X-Frame-Options in HTTP response header indicate if page 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

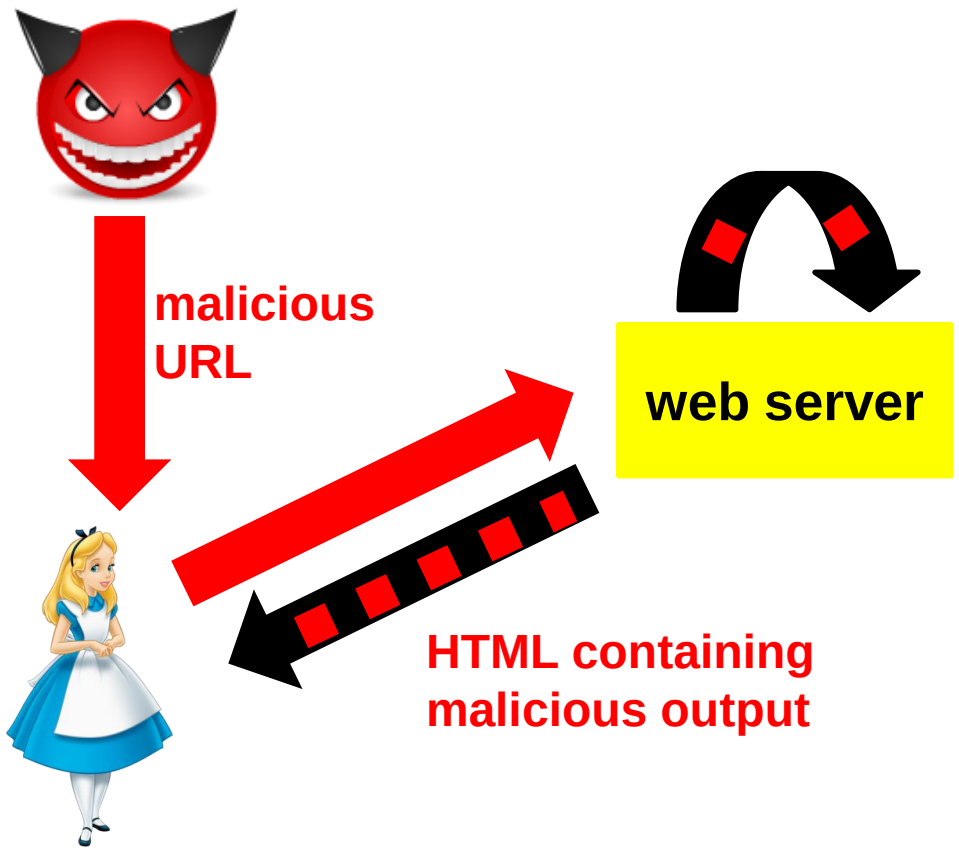
Browser protection against UI redressing

The Firefox extension NoScript extension has a ClearClick option, that warns when clicking or typing on hidden elements



CSRF
(formerly also called XSRF)

Recall: reflected (non-persistent) XSS



Recall: stored (persistent) XSS

attacker storing
malicious content
on website

malicious
input

web server

data
base

HTML containing
malicious output

another user
of the same website

XSS vs CSRF

- XSS exploits the user's trust of a specific website
  - user/client trusts the server

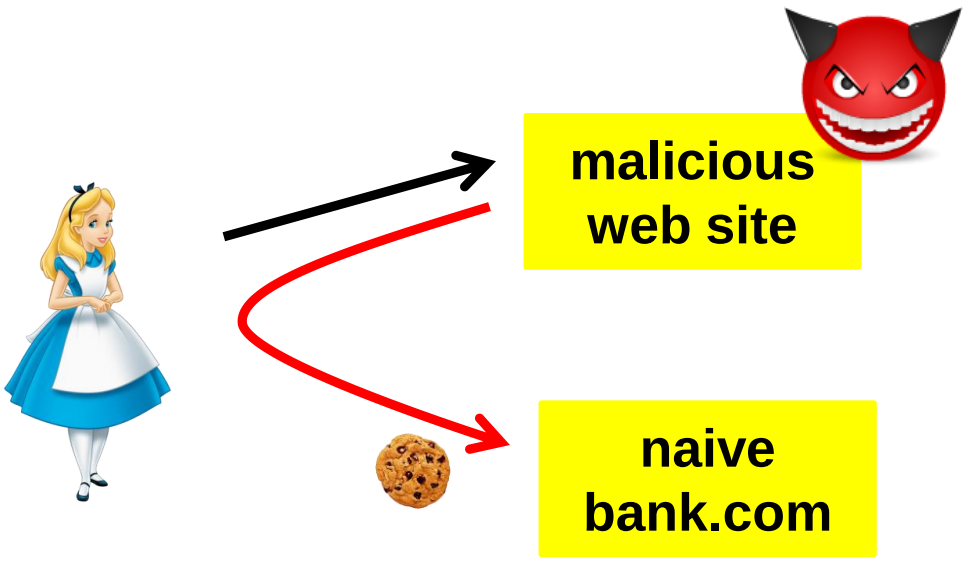
- CSRF exploits the website's trust of a specific user
  - server trusts the user/client

CSRF (Cross-Site Request Forgery)

A malicious website causes a visitor to unwittingly issue a HTTP request on another website, that trusts this user (eg. 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

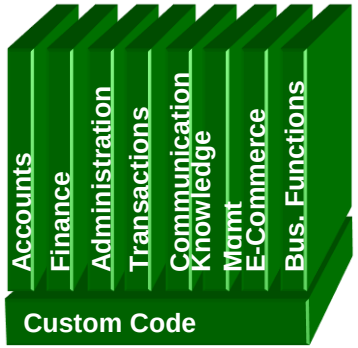
- the victim must have a valid cookie for the attacked website
- that site must have actions which only require a single HTTP request

It's a bit like click-jacking, except that it can be more than just a link,
and it does not involve UI redressing

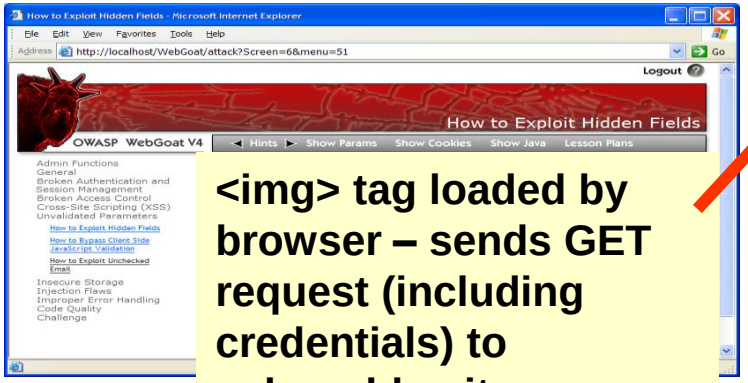
CSRF illustrated

Attacker sets trap on some website (or simply via an e-mail)

Application with CSRF vulnerability



While logged into vulnerable site, victim views attacker site



Vulnerable site sees legitimate request from victim and performs the action requested

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! The attacker can even use an image tag **<img..>** to execute the request

  ```
  <img scr="http://bank.com/transfer?amount=1000
                                  &toAccount=52.12.57.762">
  ```

- For action with a POST request:

  Trickier. The attacker cannot append data in the URL.

  Instead, the 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

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

- log out!

  at security-critical websites before visiting other websites

- don’t visit malicious web sites
  - browser could use a list of known malicious site

Preventing CSRF server-side

Problem: request look just like normal requests. Possible defenses:

- Let users re-confirm any security-critical operation
- Look at the Referrer header in HTTP request

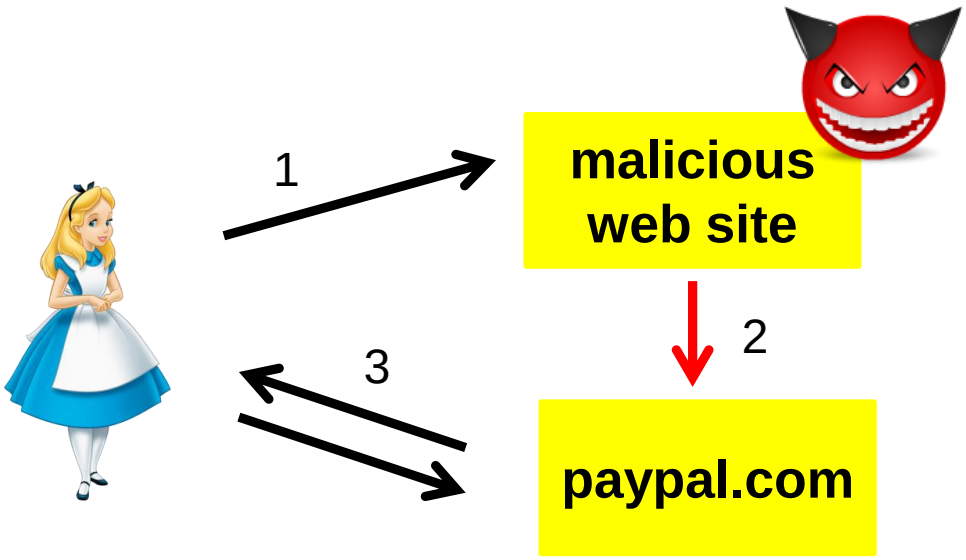
  but this may be spoofed by attacker or surpressed by the victim's browser
- Keep user sessions short

  Expire cookies, by having a short lifetime, or terminate sessions after some period of inactivity
- In addition to a cookie, also have some extra authentication token, eg. as hidden field in HTTP requests
  - 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

Variant: login attack

A malicious website forwards the victim to another website, but authenticated as the attacker instead of herself

Victim may now leak information, say credit card number, to the trusted bank website, eg. in account settings which attacker can later retrieve



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 embeds code on target website, runs code in client's browser
- 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 web site in the client,

  where client = the web browser or its human user:
  - the web site trusts that all actions are actions that the user does willingly & knowingly

- XSS abuses trust of the user in a web site
  - the user trusts that all content of a webpage is really coming from that website
    - 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

  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

Movie on XSS/CSRF at http://www.secure-abap.de/wiki/Movies

Conclusions

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

- Generic browser 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